UNIVERSITY OF CALIFORNIA
SANTA CRUZ

A FAST, PARALLEL, AND MULTI-LANGUAGE HARDWARE COMPILATION FRAMEWORK

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

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

by

Sheng-Hong Wang

December 2022

The Dissertation of
Sheng-Hong Wang is approved:

Professor Jose Renau, Chair

Professor Cormac Flanagan

Professor Scott Beamer

Peter Biehl
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

A Fast, Parallel, and Multi-Language Hardware Compilation Framework

by

Sheng-Hong Wang

A set of new Hardware Description Languages (HDLs) with a higher level of expressiveness has emerged to ease the difficulty of depicting complex hardware design. However, the increased compilation time also becomes a new bottleneck on the designer's productivity and adds more burden to the already lengthy hardware EDA flow. Meanwhile, these new HDLs tend to be developed with a stand-alone compiler, making an HDL compilation innovation hard to share with the compiler community.

I design and implement LiveHD, a new multi-threaded, fast, and generic compilation framework across many HDLs (FIRRTL, Verilog, and Pyrope). Internally, a high-level generic AST-like IR, LNAST, is used to interface the front-end source languages. Then LiveHD translate the LNAST-IR into a low-level LGraph IR, which supports most of the compilation passes and optimizations. I propose new fully and bottom-up parallel passes to handle HDLs. The resulting compiler is able to parallelize all of the compiler steps.

LiveHD can achieve 5.5x speedup scalability when elaborating a multicore RISC-V designed in the FIRRTL HDL. It also gets from 7.7x to 8.4x speedup scalability for a benchmark designed in all three HDLs. This is achieved with a fast single-threaded LiveHD baseline where it has 6x speedup compared to compilers such as Scala-FIRRTL

and 8.6x against Yosys on Verilog. The highly parallelized and generic LiveHD compilation framework will open many exciting opportunities in the HDLs compiler and EDA research domain.

Acknowledgments

First, I sincerely thank my advisor Professor Jose Renau, the kindest person in the world. Jose always opens the gate for students whenever they need a discussion, help, and advice on issues in either research or personal life. Jose's creative research ideas always broaden my mind and encourage me to make breakthroughs beyond the things that seemed impossible initially. Thousands of hours of brain-storming discussion and late-night co-debugging sessions with Jose make this thesis possible.

I would also like to thank Professor Cormac Flanagan and Professor Scott Beamer for accepting to review and providing valuable feedback for both my advancement and thesis work. This thesis wouldn't have been complete without their input.

Special thanks to my labmate Rafael, who gives me a warm hand at the beginning of my Ph.D. I will never forget how he patiently explained the codebase structures, gave valuable suggestions on doing the research, and how he helped on revising our co-authored papers even when he graduated.

Last, I would like to thank my wife, Rachel; she took most of the child-caring during these 5 years; without her push and support, I would definitely need to spend at least 2 more years finishing the PhD. Also, thank my daughter, Julie. You can't imagine how a 1-year baby girl could sit quietly and watch her dad read the paper and type code for the whole night; she's my sweet angel. Thanks to my son, Gary, who lightens my life and brings even more joy and laughs to my home.

Chapter 1

Introduction

A resurgence in hardware accelerators is thriving with the continued power and performance scaling and the emergence of new specialized domains, such as Machine Learning. This trend is shown in the latest systems like Apple M1/M2, Qualcomm Snapdragon, and AMD/Xilinx FPGA. These systems are usually equipped with a dedicated Neural Engine to process AI applications. The enormous potential market for AI processors also encourages hundreds of new hardware startups to design novel AI chips, such as Groq, Tenstorrent, Esperanto, and many others. However, the current hardware development flow contrast with agile methods that became popular in modern software development. In industry, there is a notorious problem that hardware designers have to wait days or even weeks for the time-consuming EDA compilation flow to be finished.

The lengthy EDA compilation consists of source code elaboration, logic synthesis, placement & routing (P&R), and timing analysis and modifications. Each phase

may need to perform circuit simulation to guarantee the behavior works correctly. Usually, designers have to go through several iterations of the entire EDA flow to optimize the circuit and meet timing closure.

The elaboration step translates the source code into the EDA tool's internal intermediate representation (IR). When the input HDL is Verilog, the source code elaboration usually takes a small portion of the time. However, when it comes to modern high-level HDLs, the 'elaboration' step refers to HDL compilation. Due to the more significant level of expressiveness, new HDLs like Chisel3/FIRRTL [15,45], PyRTL [24], and Pyrope [76] have garnered much interest since they make it easier to describe hardware. Each new HDL has a unique compiler that produces Verilog output from the high-level code. Thus, Verilog is frequently referred to as the assembly code output in software language compilers. This also means that the traditional elaboration step now consists of nearly the whole compilation stack as in software languages. The high-level semantic HDL code has to go through a complex compilation stack, which adds non-trivial time to the original EDA flows and makes the EDA flow even lengthier.

To address the long-standing problem of slow EDA compilation time, our group has built several 'live' hardware synthesis techniques, such as LiveSynth [70], SMatch [71], and LiveSim [78] to target logic synthesis, P&R, and simulation, respectively. When a baseline result has firstly been produced, and the designer decides to modify a small portion of front-end source code, these 'live' techniques could help to perform an incremental compilation and get a new design output within a few seconds.

Nevertheless, the designer still has to wait for a slow initial baseline compila-

tion before taking advantage of the 'live' techniques. This prerequisite hinders the designer's productivity harshly and is the core research topic that this dissertation wants to address.

Compilation infrastructures carefully tuned for speed is a crucial step for building a fast and incremental EDA compilation flow. Some facts affect the compilation time. The first is the parallel compilation scheme and scalability to address the sea of modules in a SoC. The second is the IR iteration time which directly affects the algorithm performance of a compiler pass. The third is to avoid unnecessary re-parsing between compilation steps. Besides focusing on the speed metric, some other features are also essential for an excellent hardware IR. For example, the IR iterator should be able to traverse through the whole design hierarchy without flattening the entire design. The IR data model should prevent code duplication among passes. The IR should be generic enough and provide friendly APIs for integrating other third-party tools. In this dissertation, I propose LiveHD, a new hardware development framework, to be the compilation infrastructure. LiveHD provides fast HDL and EDA compilation speed and serves as a common database to integrate with novel hardware compilation passes, such as the 'live' techniques and the other open source tools [43, 44, 80].

LiveHD is built on top of a graph structure called LGraph (Live Graph) [69, 84]. LGraph is a sparse graph representation that serves as a design database and is carefully crafted for incremental live hardware development. LGraph is an LLVM-like infrastructure of hardware design tools, representing large-scale designs in different VLSI phases. It aims to provide incremental elaboration, synthesis, and simulation.

LGraph is carefully crafted with a minimal number of IR node type; the limited node types could reduce the compiler passes design and time complexity.

Though LGraph provides many excellent features as a hardware compiler infrastructure, it is essentially a low-level graph IR, making it hard to interface directly with modern HDLs. Modern HDLs usually define high-level semantics such as aggregate types and circuit attributes. In order to enable multiple HDLs to leverage the fast compilation that LiveHD provides, I introduce a new high-level IR called LNAST (for Language Neutral AST). The LNAST IR is the standard interface that allows easy translation of new languages to LiveHD. By targeting LNAST instead of LGraph, the language designer does not need to worry about SSA, control-flow conversion, variable scopes, and many other constructs shared by most languages. In a way, LNAST is more accessible to translate to because it has a control flow with several operations.

The IR design is a critical component in compiler development [53, 54]. It is common practice for a compiler tool stack to have multiple layers of IR, ranging from high-level to low-level representations. An excellent high-level IR has simple semantics to express the high-level source language and hides details about the language syntax to the back-end compiler stack. It must be independent of the front-end programming language and leverage shared code optimization. In the new LiveHD model, we can view LNAST as a tree-like high-level IR and LGraph as a graph-like low-level IR.

Nowadays, hardware designers need to integrate thousands of modules into a deep design hierarchy that constitutes a modern SoC. Even more challenging, many hardware tools lack scalable or parallel compilation steps that software compilers have.

On the other hand, the enormous module instances in a SoC provide an opportunity for boosting the compilation throughput from a parallelized compiler. In this thesis, I further proposed a new multi-threaded and high scalability compilation scheme in the LiveHD framework. Furthermore, the parallelized mechanism is suitable for multiple HDLs.

In summary, the main contributions in this dissertation are:

- a low-level LGraph IR for fast HDL compilation, synthesis, simulation, and interfacing other tools
- a high-level LNAST IR to bridge different HDLs into LiveHD.
- implementing many hardware-specific compiler passes based on LNAST and LGraph IR, and precious lessons learned about hardware compiler design exploration among passes and IRs.
- a generic and parallelized compilation scheme capable of compiling multiple HDLs front-end.

Multiple communities will benefit from LiveHD. Developers of new HDLs can map to LNAST and leverage the existing compiler infrastructure. Physical design groups can integrate into LGraph to provide support for different front-end languages and to evaluate integration with other steps, moving beyond simple benchmarks regularly used for specific steps in physical design. RTL circuit designers can use the integrated open-source flow, which provides a shallow entry-level barrier, instead of

spending time integrating tools from different domains. We see LiveHD as the LLVM-like system in hardware design since it provides a convergence point for both language developers and back-end engineers. Users can access the open-sourced LiveHD project code base at [41].

The remainder of this dissertation is organized as follows to discuss how the proposed LNAST-IR, LGraph-IR, and parallelized LiveHD framework boost compilation throughput for multi-HDLs:

Chapter 2 takes a quick overview of the internal steps of the LiveHD framework and discusses the language abstractions and IR choices.

Chapter 3 takes a quick overview of the 'live' synthesis and simulation techniques that have been published that will be an excellent complement for our proposed fast multi-threaded HDL compilation framework. I then introduce important related works, including hardware and software IR designs, and other hardware compilation framework that shares the same goal of LiveHD goal to be a generic development framework for hardware language compilation and tool development.

Chapter 4 discusses work done for the low-level LGraph IR. Critical features of LGraph include a unified data model and API, a fast memory mapped library design, integration with third-party tools, and hierarchical design traversal for third-party tools. I further explain some of the compilation pass implementations based on LGraph. Some more depth IR design considerations for hardware are also discussed in this chapter.

Then, Chapter 5 presents our high-level LNAST IR. LNAST offers one main

benefit to the LiveHD framework. It acts as a bridge for the LiveHD flow to interface with different HDLs at the front end. By targeting LNAST instead of LGraph, the language designer does not need to worry about SSA, control flows conversion, variable scopes, and many other constructs shared by most languages. This chapter will also discuss the passes implemented on LNAST.

In Chapter 6, I propose a new design to parallelized LiveHD framework. I demonstrate how LiveHD can extract a dependency tree during the IR lowering process. Then I illustrate how to turn each pass into either a fully-parallelized or bottom-up parallelized mechanism based on the dependency tree relations. I then explain the special care that needs to be taken for different front-end HDLs.

Finally, I provide my final thoughts on Chapter 7 and talk about potential future open research projects that this work could inspire. In my Ph.D. journey, I also have collaboratively work on other projects that are not discussed in the thesis. I co-designed the semantics and syntax of the new Pyrope HDL, along with the LiveHD framework. I also participated in several exciting research prototypes with LiveHD, including leveraging LiveHD's multi-languages code generation ability [40] to increase the compiler testing coverage.

Chapter 2

LiveHD Overview

This chapter gives a concise summary of the steps that occur internally within the LiveHD framework. It also discusses the language abstraction and IR options available.

2.1 HDL Support Choice

Verilog is the language that is considered to be the industry standard, and Chisel3 is the most widely used modern alternative to Verilog. Therefore, it was apparent in the early design process that LiveHD would need to support both languages. LiveHD also supports Pyrope, an HDL still in development that already has features like the global inference that affect the compile design options. Because we want LiveHD to support the full Verilog 2001 language and some SystemVerilog features, the compiler needs to pay attention to many details. LiveHD can directly interface Verilog and Chisel3 generated code at compile time. This allows for optimizations to be performed

across modules.

LiveHD uses Slang [8] to parse Verilog. Even non-synthesizable constructs like classes are no problem for Slang, which can handle most of SystemVerilog. The synthesizable subset is the only one LiveHD will accept. When it comes to Chisel3 [15], LiveHD accepts CHIRRTL, which is comparable to the FIRRTL [45] but is directly produced by Chisel3. The original Scala-FIRRTL [45] compiler will accept the same CHIRRTL before each of the various lowering steps that are performed within FIRRTL. A specialized parser is implemented in LiveHD to handle Pyrope.

2.2 Hardware IR Choice

LiveHD has two internal IRs, LNAST [85] and LGraph [84]. The LNAST and the LGraph internal IRs have their own associated data structures in LiveHD. LNAST is a control flow language-neutral intermediate representation that maintains the control flow. LNAST can be derived from any of the three different HDLs. Compared to IRs produced by non-hardware compilers, LNAST is somewhere between the HIR and MIR produced by Rust [11], or it is closer to the AST than the IR used in LLVM [54]. LGraph is a form of graph representation more analogous to a hardware netlist. There is a lowering or translation step from LNAST to LGraph.

Internally, LNAST is equipped with a Static Single Assignment (SSA) [27] pass that is able to enable its very own compiler optimization steps. However, for the purposes of this particular piece of research, the most crucial function of LNAST is to han-

dle all control flow structures properly before it generates an LGraph.

HDL-specific passes, such as bitwidth inference and code optimization, are included in LGraph. Even though it is possible to complete some of the steps in LNAST, the design process can be made much simpler using LGraph's various traversal algorithms, such as topological sort.

2.3 LiveHD Overview

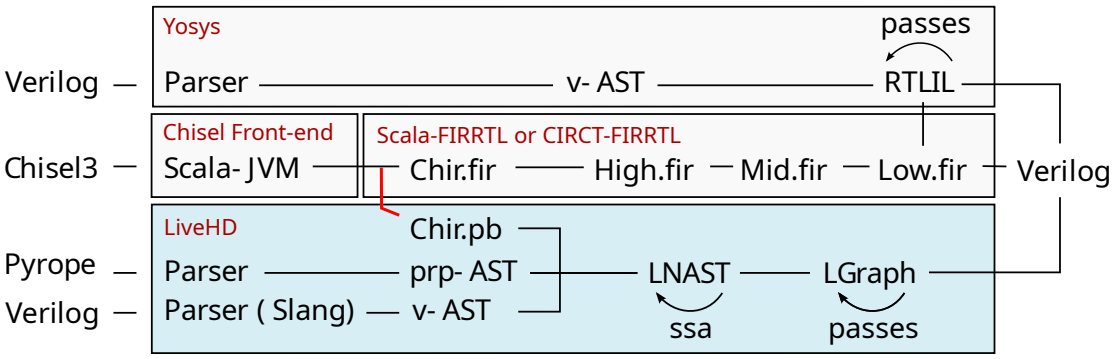


Figure 2.1: Overview of LiveHD compilation flow and comparisons between Yosys and Chisel3/FIRRTL compilers.

The high-level structure of the LiveHD compiler is illustrated in Figure 2.1. Currently, the front end of LiveHD is capable of compiling three different HDLs. These HDLs are FIRRTL, Pyrope, and Verilog. As an example, prp-AST was generated by

our very own Pyrope parser, sv AST was generated using Slang [8], and Chir.pb was generated using Protocol 10 Buffers [4].

Every language has a unique pass responsible for translating its internal AST to the LNAST-IR. After entering LNAST, the internal data for all three languages becomes identical. Because LNAST is the target structure, the language designer does not have to be concerned with SSA, the conversion of control flows, variable scopes, or any of the other numerous constructs common to most languages. The fact that the input is not required to be in SSA form by LNAST is because LNAST itself is internally equipped with an SSA transformation. Thus, targeting LNAST relieves language designers' efforts.

LNAST does not have language-specific nodes. On the contrary, it makes it possible to create function calls to black-boxed modules whenever required. This is taken advantage of by the FIRRTL pass, which transforms a FIRRTL operation into a function call to an LNAST node of blackbox. It is a different approach to MLIR [55] that involves the creation of dialect for custom nodes. The LNAST approach allows custom passes for each language while enabling each compiler pass to handle semantics appropriately.

The front-end design will be represented as LGraph when translated from LNAST. An LNAST node can require multiple LGraph nodes. LGraph makes use of the majority of the LiveHD compilation passes in order to generate optimized Verilog output. These compilation passes include copy-propagation, constant folding, peephole optimization, and bitwidth inference.

A hierarchical graph is the key characteristic of LGraph. We refer to a graph as hierarchical when it can point to other graphs. There are many different constructs, such as hierarchical iterators that can be used to manage graphs.

2.4 Clarification

LiveHD is a large project that many students have collaborated and contributed to together. The specific contributions made in this thesis are summarized in Table 2.1.

Table 2.1 Detailed contributions from the author

item	**comments**
Parallelized LiveHD	main designer
End-to-end Pyrope compilation flow	main designer except the parser [1]
End-to-end FIRRTL compilation flow	main designer
End-to-end Verilog compilation flow	collaboration [2]
LNAST IR infrastructure	main designer with high collaboration [3]
LGraph IR infrastructure	collaboration [4]
FIRRTL to LNAST	main designer with high collaboration [5]
LNAST SSA transformation	main designer
LNAST to LGraph pass	main designer
Cprop pass	main designer with high collaboration [6]
FIRRTL bit analysis pass	main designer
FIRRTL map pass	main designer
Bitwidth inference pass	main designer with high collaboration [7]
LGraph Mockturtle integration	collaboration [8]

[1] Pyrope parser is designed by Kenneth Mayer, a MS student alumni
[2] Verilog parser is maintained by Prof. Renau
[3] co-develop the LNAST interfaces and the tree library with Prof. Renau
[4] co-develop the LGraph interfaces and the graph library with Prof. Renau
[5] the intitial prototype that interfaces hi-FIRRTL is designed by Hunter Coffman, a MS student alumni; I continue his work but re-design most part of the codebase to handle CHIRRTL, the highest format of FIRRTL
[6] co-develop with Prof. Renau to handle tuple structure
[7] the initial prototype is also designed by Hunter Coffman; I and Prof. Renau keep improve the codebase
[8] the initial prototype is designed by Qian Chen; I continue the design and perform the benchmarking

Chapter 3

Related Work

First, this section will discuss the live programming techniques that intrigue our interest in starting the LiveHD research concept. After that, I discuss the IR designs and frameworks relevant to software and hardware communities.

3.1 Live Techniques in LiveHD

Unlike hardware development, software development is much more agile than Hardware design. For example, a trivial change in a large software project like the Linux kernel requires less than one minute to recompile incrementally. More recently, Live programming has received much attention in the software community. In Live programming, developers can see the output from their code change immediately as if the program is always running. Faster feedback leads to a more productive and less frustrating development experience [28]. A couple of seconds is the recommended [28] value to have a *responsive user* interaction.

As a complementary perspective of this thesis, the ultimate goal for LiveHD is building an interactive hardware design experience, much like live programming, by providing feedback within a few seconds. This is possible by applying three principles for incremental hardware design: (1) divide the job into partition regions or checkpoints; (2) incrementally transform these partition regions where the code change happens; and (3) hot-reload the partition regions into a running program without restarting.

Three main live programming techniques in LiveHD have been proposed, including LiveSynth [70], SMatch [71], and LiveSim [78]. These tasks are designed under the three incremental principles. They only handle the sub-jobs related to the code change, then merge results into the background program without re-running from the beginning. LiveSynth focuses on logic synthesis, and SMatch performs P&R for FPGA. LiveSim is our live simulator.

These research prototypes still have two substantial problems. Before moving on to the incremental phase, the baseline synthesis or simulation must be completed for any of these live techniques to even be considered. However, waiting for these protracted baseline compilations eliminates the incremental phase's benefits. In addition, all the prototypes can only accept the low-level Verilog as inputs, despite high-level HDLs becoming increasingly widespread. Even if the high-level HDLs are used, their elaboration/compilation times are no longer negligible. Throughout this thesis, LiveHD will devise a parallelized compilation strategy in addition to two new fast IRs to address these concerns. After that, the brand-new LiveHD framework will be able to

compile new high-level HDLs rapidly and further enable fast-baseline circuit synthesis and simulation for future research.

3.2 Hardware Description Language Compilers and IRs

HDL compilation, hardware IR design, and tools have recently been a research hotspot in the open-source community [2, 3, 10, 12, 16, 24, 29, 45, 47, 50, 58–60, 63, 64, 67, 73, 75, 76, 79, 83, 89]. Nonetheless, most proposed works are tightly knit to their source language, and compilation speed is not the priority. On the other hand, several software languages compiler have evolved toward parallelism or multi-language support.

Yosys

Yosys [89] is a framework for register-transfer-level (RTL) synthesis. It takes Verilog-2005 as the input and converts it to its internal RTLIL [89] IR through a Verilog Abstract Syntax Tree (v-AST in Figure 2.1). RTLIL cannot represent high-level HDL constructs like *tuple* or *vector*. Several front-end passes are needed in Yosys to translate the initial Verilog AST to RTLIL and further down to a more netlist-like construct through the *proc* and *opt* steps. Yosys compilation is sequential, and these front-end passes are not trivial and could easily take Yosys minutes when compiling large designs.

Scala-FIRRTL

FIRRTL is the IR in Chisel3 [15]. The first FIRRTL compiler is implemented in Scala [65](Scala-FIRRTL). A front-end Chisel3 compiler produces CHIRRTL as the input for the Scala-

FIRRTL compiler (Figure 2.1). The Scala-FIRRTL compiler is not designed for compiling languages other than Chisel3/FIRRTL. Moreover, due to the AST-centric and non-SSA representation of FIRRTL, it is not easy to find use-def chains for a variable. Multiple tree iterations must be performed to build the data structures for AST transformations. The long tree traversal time, together with sequential compilation in Scala-FIRRTL, is a problem for large digital designs.

CIRCT-FIRRTL

CIRCT [10,32] is a new experimental hardware IR extended from MLIR [55] and LLVM [54] communities. CIRCT framework shares the same ideas as LiveHD, i.e., to be the unified hardware development center. Theoretically, it is possible to compile multiple languages through interfacing various front-end MLIR dialects designed in CIRCT. Right now, the CIRCT-FIRRTL flow (Figure 2.1) is concurrently developed with LiveHD, with a focus on FIRRTL input.

Other HDL IRs and compilers

LLHD [73] and CoreIR [60] are IRs aiming to be the generic hardware representation for the RTL abstraction level. LLHD is a statically-typed hardware IR designed to capture SystemVerilog. In LLHD, the bitwidth of variables must be explicitly defined. Thus it cannot be easily interfaced with modern HDLs, which only set bitwidth on the modules' I/O. In CoreIR, input HDLs like Halide [72] and Verilog are now supported, and the compilation speed is not the main concern. Furthermore, before mapping to these two IRs, extra bitwidth analysis passes are required to map HDLs' bits-centric operators.

Several works [9, 20, 52, 59, 62, 63, 75] have been proposed to handle the HLS abstraction. Generally, the higher abstraction offers more freedom for expressive syntaxes. However, it usually puts more burden on the compiler to reason about the relationship between high-level code and the generated circuit. Also, in terms of multi-language compilation ability, these HLS projects cannot compile Verilog source. Thus, they lose the opportunity to reuse and optimize Verilog designs.

Some IRs are designed specifically for back-end EDA tool development. In the commercial world, OpenAccess (OA) is an "open" format meant to provide interoperability among IC design tools. Ironically, OA requires signing NDAs that limit its usage. The other commercial option is MilkyWay from SynopsysTM. However, since this is a proprietary format from Synopsys, not much can be said about it. The commercial nature of both options makes them hard to adopt for academic research. The OpenRoad [50] project tries to build a machine-learning-driven automatic RTL-to-GDS flow in the open-source community without human interference. Several open-source projects aim to leverage Python as the host to build new HDL and the corresponding compilers. Representative works are PyMTL and Mamba [47, 58], Magma [3], and MyHDL [29]. Rsyn [34] and Ophidian [35] are two open-source frameworks for physical design. They provide an extensible infrastructure that allows users to leverage existing code and focus on new algorithms and tasks.

3.3 Software Programming Language Compilers

Several software frameworks have similar design concept as LiveHD in terms of parallelism and multi-language support.

Parallel compilation

The two widely used C/C++ compilation frameworks, GCC [81] and LLVM [54] mostly rely on the build system such as Makefile to achieve file-level compilation parallelism. Few of the LLVM internal passes like the linker can also be parallelized.

Several research works have been recently proposed improving the compilation parallelism. The challenges of increasing scalability for Git and GCC applications are discussed in the research work by Bernardino et al. [18]. The parallel GCC project [17] also aims to conduct a multi-threaded compilation on the intra-procedure optimizations in GCC. Researchers in [38,48] have been working on getting higher parallelism in the link time optimization (LTO) stage. In Lighting Bolt [66], the authors discuss how they design the parallel mechanism to improve the performance of the binary optimization pass.

Elixir [33] is a functional language that also focuses on constructing highly scalable applications. The internal framework launches multiple compilers to handle separate files simultaneously. When a function dependency bottleneck occurs, the framework sends *waiting signals* to the dependent caller-compiler and pauses until the dependency is resolved.

Multi-language support

GraalVM [31, 87] is a Java virtual machine framework that bridges multiple languages by using Truffle [39] as the front-end IR. They are analogous to the LiveHD compiler and its front-end IR, LNAST [85]. The AST of several languages is mapped to the common Truffle AST in this framework. A series of back-end GraalVM optimizations like tree rewriting and just-in-time(JIT) compiling are applied to the common Truffle AST. Click and Paleczny [23] present a graph-based SSA intermediate representation to express optimization elegantly. The Common Intermediate Language (CIL) [61] is used in the .NET system, and it is also an IR designed for multiple languages such as C# and Basic.

Chapter 4

LGraph: A Unified Low-level Graph-IR for Productive Hardware Design

In the first part of this chapter, the discussion focuses on the essential aspects of LGraph IR that were created with the development of hardware compilation tools in mind. This chapter also reviews the considerations regarding the IR design of LGraph's hardware. Additionally, some integration prototypes for third-party tools are presented. In the final part of this chapter, a lesson learned from the LiveHD development is reported, a fast but deprecated memory-mapped library. The memory-mapped library and the LGraph-Mockturtle are assessed to know how well the integration works together.

4.1 Introduction

Specialized hardware accelerators provide extra performance, power, and area in multiple areas in the post-Moore's law era. A new momentum of hardware design innovation would come from open-source EDA tools and highly productive hardware design flows. Hardware designers want a fast design flow to iterate between synthesis and its analysis. EDA tool developers, in particular from the open-source community, want to work with a common model and API to focus on the tool's algorithm development.

Ideally, a productive hardware design flow should have a very short design iteration period. This helps designers quickly implement the new design idea based on the feedback from an interactive environment. Whereas, in a traditional design flow, it is common for designers to wait for hours or even days to obtain the design result.

Open-source EDA tools also play a vital role in hardware innovation as fellow researchers and hardware developers could contribute their novelty without facing licensing constraints. In recent years, research work such as DATC [49], qflow [5], VTR [74] and OpenROAD [50] focus on integrating tools of different design stages into a single RTL-to-GDSII flow. Some of the single-stage tools are ABC [19], Mockturtle [80], and Yosys [89] for logic synthesis, OpenTimer [43, 44], OpenSTA [46] for static timing analysis, RePlAce [22] and NTUPlace3 [21] for placement and NCTU-GR [57] and TritonRoute [51] for routing; Verilator [79], LiveSim [78], and Essent [16,67] for simulation.

These works on integrating open-source tools have shown the potential to

build a tapeout-ready flow. Despite the correctness of these design flows, they are still far from ideal. One crucial source of issues is the lack of a common data model and APIs. Individual tools are developed using different data structures, which raises the integration difficulty. Moreover, tools not developed using a common data model end up replicating code and efforts. For example, almost every tool implements its own netlist parser. This code replication further causes a non-negligible portion of the flow execution time. To make matters worse, not all tools implement standards equally, causing compatibility issues.

Several sources for the slow hardware compilation flow include design elaboration, logic synthesis, timing analysis, placement, and routing. State-of-art incremental technique like LiveSynth [70] and SMatch [71] have been applied to provide an interactive experience but are limited to synthesis. However, EDA tools are IO-heavy applications. Re-parsing netlists or libraries to the tools' internal data structure adds much to the flow's run time. As mentioned in [69], it would take Yosys [89] tens of seconds to parse a reasonably large RTL file, which leads to a less productive design experience. The situation worsens when the project goes into debug or optimization phase. Although changes applied in multiple flow iterations are small, designers must repeatedly wait for the same re-parsing time.

Performing design synthesis and static timing analysis in a hierarchical manner is essential for productivity in the hardware design flow. However, as mentioned in [36], most open-source logic synthesis and STA tools such as OpenTimer, Mockturtle, and ABC lack hierarchical design support as compared to industrial tools.

In order to optimize the whole design, the designer must flatten the hierarchical design and then feed it to these tools. However, physically flattening every submodule during the logic synthesis phase would increase the complexity of the back-end physical synthesis. Furthermore even if each tool implements the hierarchical feature, the insidious code replication among these tools still violates the DRY (do not repeat yourself) principle in software development.

In this chapter, I present Live Graph(LGraph), our attempt to build an infrastructure for productive hardware design flow. The following are the highlighted key features of the LGraph-IR:

Unified data model/API

LGraph has a unified data model and API in C++17 for digital circuits. LGraph is meant to represent netlists in different phases of the design flow from RTL to layout, including simulation and code generation. The easy-to-use APIs vastly reduce the design effort of tool developers. More importantly, the nature of the unified data model prevent possible code duplication and avoids parsing and generating the netlist between the internal stages of the RTL-to-GDSII flow.

Hierarchical design traversal

The hierarchical cross-module traversal ability of LGraph empowers the integrated third-party tools to run the core algorithm in a virtually flattened form. Therefore, LGraph could implicitly achieve global optimization without affecting the physical design phase.

Third-party tool integration

LGraph is currently being integrated with some open-source tools such as Mockturtle, OpenTimer, and Yosys. Open-source EDA tool developers could leverage LGraph's succinct API and generic data structures to implement their algorithms. Alternatively, they can leverage other integrated third-party tools to complete the design flow together.

4.2 LGraph-IR Construction and Traversal

4.2.1 Node, Pin, and Edge Construction

A single LGraph represents a single netlist module. LGraph comprises nodes, node_pins, edges, and tables of attributes. An LGraph node is affiliated with a node_type, and each type defines different amounts of input and output node_pins. For example, a node can have three input pins and two output pins. Each of the IO pins can have many edges to other graph nodes. Every node_pin has an affiliated node_pid. A pair of a driver_pin and a sink_pin constitute an edge. In the API example in List 4.1, an edge is connected from a driver_pin (pid1) to a sink_pin (pid3). The bitwidth of the driver_pin determines the edge bitwidth.

```
auto node = lg->create_node(Node_Type_Op);
auto dpin = node.setup_driver_pin(1);
dpin.set_bits(8);
auto spin = node2.setup_sink_pin(3);
dpin.connect(spin);
```

Listing 4.1: Selected API examples for LGraph construction

4.2.2 LGraph Traversal

LGraph is a bidirectional graph representation supporting topological sort traversal in an input-forward and output-backward manner. If the order in which nodes are visited does not matter for the algorithm, developers can choose the fast iterator, which will visit the next node in the cache line. Besides nodes iteration, LGraph also provides an API for visiting each input and output edges of a node.

```
// unordered but very fast traversal
for (const auto &node:lg->fast())     {...}

// propagates forward from each input/constant
for (const auto &node:lg->forward())  {...}

// propagates backward from each output
for (const auto &node:lg->backward()) {...}
```

Listing 4.2: API examples for LGraph traversal

```
for (const auto &inp_edge : node.inp_edges()) {...}

for (const auto &out_edge : node.out_edges()) {...}
```

Listing 4.3: API examples for edge iteration of a node in LGraph

```
for (auto &out : node.out_edges()) {
  auto dpin       = out.driver;
  auto dpin_pid   = dpin.get_pid();
  auto dnode_name = dpin.get_node().debug_name();
  auto snode_name = out.sink.get_node().debug_name();
  auto spin_pid   = out.sink.get_pid();
  auto dpin_name  = dpin.has_name() ? dpin.get_name() : "";
  auto dbits      = dpin.get_bits();

  fmt::print(" {}->{}[label=\"{}b :{} :{} :{}\"];\n"
      , dnode_name, snode_name, dbits, dpin_pid, spin_pid,
    dpin_name);
}
```

Listing 4.4: iterate output edges and get node/pin information from it

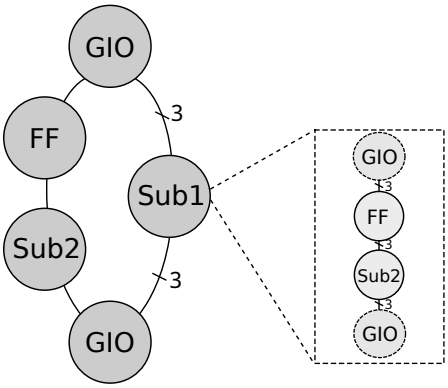


Figure 4.1: A hierarchical LGraph. A sub-graph node is a sub-module instantiation. The hierarchical traversal will walk into the sub-graph structure.

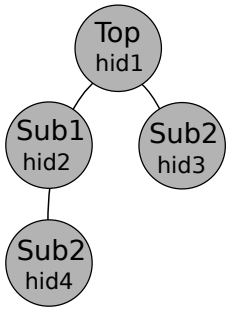


Figure 4.2: The tree hierarchical view for Figure 4.1. Each instantiation has a unique hid and hierarchical attribute table.

LGraph supports hierarchical traversal. Each sub-module of a hierarchical design will be transformed into a new LGraph and represented as a sub-graph node in the parent module, as shown in Figure 4.1. Suppose a hierarchical traversal is used when the iterator encounters a sub-graph node, it will load the persistent tables of the sub-graph to the memory and traverse the sub-graph recursively, ignoring the sub-graph input/outputs. This cross-module traversal treats the hierarchical netlist just like a flattened design. In this way, all integrated third-party tools can automatically achieve global design optimization or analysis by leveraging the LGraph hierarchical traversal feature.

```
1  for (const auto &node:lg->forward_hier()) {...}
```

Listing 4.5: API example for LGraph hierarchical traversal

4.3 LGraph-IR Characteristics

Several important characteristics when designing LGraph-IR are discussed in this section. I also provide a comparison summary between LGraph/LiveHD and other related tools in Table 4.2.

4.3.1 SSA Graph

LGraph has an SSA representation, but others such as Yosys' internals do not. The term "SSA-graph" refers to a wire or network that has a driver from a single source.

The SSA representation has been demonstrated to be very useful in conventional compilers for the purpose of simplifying passes. Pointers do not exist in hardware, and loops are used very infrequently, making the SSA an even more interesting concept.

4.3.2 Reduced Logic Instruction Set (RLIS) Cells

Logic cell need different graph nodes to represent functionality. Most EDA tools have many logic cells or gate options. While some cells are commonly used, such as *add*, *subtract*, *concatenate wires*, *pick bits*, *one_hot_encoding*, many cells are only created for a specific function. Yosys has over 100 types, XLS has 60, and FIRRTL 37. LGraph logic cells have been designed to reduce the number of gates. In a way, it tries to be a Reduced Instruction Set Logic (RLIS). It is similar to that RISC is a reduced instruction set vs. CISC. The goal has been to have the smallest amount of cells covering all the options without the overhead. The reason for choosing the RLIS philosophy is to reduce the design complexity of passes and optimizations on the IR. Table 4.1 summarizes the carefully crafted RLIS LGraph operators.

4.3.3 N-ary Gates

LGraph logic cells are n-ary (see Table 4.1). N-ary means that cells like *or* or *add* can have an unlimited number of inputs. This allows for simpler graphs and easier optimizations because there is no need to have a chain of operators representing the same functionality.

Table 4.1 The reduced number of operator types in LGraph are designed to simplify the compilation complexity

Operator	**Functionality**
Const	constant number or strings
Sum	summation of n-inputs
Mult	multiplication of n-inputs
Div	division of 2-inputs
And	and-gate with n-inputs
Or	or-gate with n-inputs
Xor	xor-gate with n-inputs
Ror	reduced-or-gate with n-inputs
Not	bitwise not operation
Get_mask	get signed value from a given bit position
Set_mask	turn a node from unsigned to signed
Sext	signed extend from a given bit position
LT[1]	less than operation
GT[2]	greater than operation
EQ[3]	equal to operation
SHL	logical shift left
SRA	arithmetic shift right
LUT	look-up table in FPGA
Mux	n-to-1 multiplexer
IO	Graph input or output
Memory	models variant memory types[4]
Flop	models variant flop types[5]
Latch	models Latch gate
Fflop	models fluid-pipeline flop [68]
Sub	models submodule and function-call
TupAdd	add a field to high-level tuple struct
TupGet	get a field from high-level tuple struct
AttrSet	set a attribute to a variable
AttrGet	get a attribute from variable
CompileErr	indicate a compile error during a pass

note: [1] greater-or-equals-to operation could be represented as !LT, [2] less-or-equals-to operation could be represented as !GT, [3] not-equals-to operation could be represented as !EQ [4] provides an interface to model memory like combinational/sequential read, also accepts and generates high-level memory data struct [5] provides an interface to model flops like asynchronous/synchronous reset, positve/negative edge reset, and reset initialization value.

4.3.4 Signed Wires

Most EDA flows have to deal with signed and unsigned wires. In a flow like Yosys, logic cells behave differently with signed and unsigned inputs/outputs. This separation adds even more complexity to the logic cells. In FIRRTL, most of the 37 existing cells also have different behavior regarding cell inputs' signedness. However, signed representation is indeed a superset of unsigned. If the flow supports signed, there is no reason to support unsigned. Choosing a unified signed representation leads to less design overhead (positive values in an unsigned result have the upper bit zero). For example, the bitwidth inference pass no longer needs to differentiate a cell's different input signedness.

Compilations in the proposed signed representation require one extra bit in the LGraph IR. A design module can be broadly categorized into (1) internal logic and (2) IO. LiveHD implements a *bitwidth* pass that calculates each variable's max/min value (gate). Suppose the min value of a variable is always larger or equivalent to zero, and this variable is semantically unsigned. In this case, the *verilog_gen* pass will generate an unsigned Verilog syntax when the variable is an internal logic (case-I). For a design module, if all the internal logic has a positive minimum value, then all the internal logic will generate unsigned Verilog statements. In this scenario, there is no overhead from LiveHD's universal signed representation for the module's internal logic. On the other hand, if a module's internal logic has a negative minimum value from the *bitwidth* pass, there will be an extra signed bit overhead on the generated Verilog, and we have

to rely on the synthesis tools to find pruning opportunities.

The interesting part is the IO. Theoretically, we can also generate unsigned IOs when the min value is positive, which will be the most common case in other CAD tools. Currently, our *verilog_gen* pass will always generate signed IOs. This is because we plan to interface with other LiveHD passes in the future, and those future passes will also be universally signed. This design choice will save us a lot of development effort in the future. These extra signed 1-bits on the IOs produce extra overhead, but they could be potentially optimized away from synthesis tools. In fact, the future LiveHD will integrate synthesis tools, and global analysis and synthesis steps will take care of this overhead from the LiveHD front-end compilation.

4.3.5 Wire Width Semantics

LGraph wires are designed to be always signed, but maybe equally interesting, the cell semantics are independent of the wire widths. This means that a gate-like *concatenate* is not allowed because it is wire width dependent. We make this design choice because even if the designer specifies a 'maximum' number of bits to a wire, the tool should be allowed to optimize the wire's bitwidth. This already happens in synthesis tools that can simplify away wires. LiveHD brings the same concept/ideas to LGraph.

4.3.6 Bitwidth Inference

Verilog specifies the bitwidths for each wire. Other modern HDLs like Chisel [15] or Pyrope [76] have a different bitwidth semantic depending on the operators. For these

Table 4.2 Capability comparison between relevant tools

	Yosys [89]	FIRRTL [45]	XLS [9]	coreIR [60]	LLHD [73]	Verilator [79]	CIRCT [10]	LGraph/LiveHD
HDLs	Verilog	FIRRTL[0]	C++ DSLX[1]	Verilog Halide	SV[2,3]	SV	FIRRTL	Verilog SV[4] FIRRTL[0] Pyrope[5]
Cell Types	>100	~37	~60	~40	~60	>100[6]	>100[a]	31
Aggr. Type[7]	no	yes	yes	yes	yes	yes	yes	yes
Signedness	S/U[8]	S/U	S/U	S/U	S/U	S/U	S/U	S
N-ary Gate	no	no	yes	no	no	no	yes	yes
Global Inf[9]	no	no	no	no	no	no	yes	yes
Formal	yes	yes	yes	yes	no	no	no	no
Simulation	yes	yes	yes	no	yes	yes	yes	yes
Synthesis	yes	no	no	no	todo	no	no	yes
FPGA	yes	no	no	no	no	no	yes	yes

[0] highest format, CHIRRTL
[1] Google's high-level synthesis language
[2] SystemVerilog
[3] At the time of writing, SystemVerilog is implemented
[4] we integrate Slang [8] to bridge SystemVerilog
[5] Pyrope [6] includes many simulation-only constructs
[7] High-level aggregate types like tuple, vector [8] Signed/Unsigned
[9] Global type inference
[a] summation of all dialect CIRCT included

languages, the flow can not know all the bitwidths at the elaboration phase; a bitwidth analysis must be performed to gain knowledge of each gate's bit size before code generation. LiveHD has been designed to allow the generic bitwidth inference for every HDLs.

4.3.7 Global Inference

Another flexibility that LiveHD provides is global inference. Some programming languages have global type inference, others have local type inference, and others like Verilog and FIRRTL have no inference. To support a wider superset of languages, LiveHD is designed to perform both global and local inference. The types, bits, and other attribute fields are propagated through the graph hierarchy. Other tools, except ML-based like Clash [86] and Lava [77], seem to have local type inference or none.

4.3.8 Prototype Inference

Currently, LiveHD does not have a working object model, but the object methods and attributes are built following prototype inheritance. This means that an object/struct can be extended and changed. It does not require specifying a fixed type.

4.4 LGraph Attribute Design

The design attribute stands for the characteristic given to an LGraph node or node_pin. For instance, the characteristic of a node name and node physical placement. Even though a single LGraph represents a particular module, it can be instantiated multiple times, for example, the *sub2* node in Figure 4.1. In this case, the same module could have different attributes in the different hierarchies of the netlist. A good design of attribute structure should be able to represent both non-hierarchical and hierarchical characteristics.

4.4.1 Non-Hierarchical Attribute

Non-hierarchical LGraph attributes include pin_name, node_name, and line of source code. Such properties should be the same across different LGraph instantiations. Two instantiations of the same LGraph module will have the same user-defined node name on every node. For example, in Figure 4.1, instantiations of a subgraph 2 in both top and sub-graph 1 would maintain the same non-hierarchical attribute table.

```
1  node.set_name(std::string_view name);
```

Listing 4.6: API example for LGraph attribute setting

4.4.2 Hierarchical Attribute

I introduced a new hierarchical LGraph attribute design after an inspirational discussion with the author of FIRRTL. LGraph's hierarchical attribute is achieved using a tree data structure to record the design hierarchy. In LGraph, every graph has a unique id (lg_id). Every instantiation of a graph forms some nodes in the tree, and every tree node is indexed by a unique hierarchical id (hid). As shown in Figure 4.2, we can identify a unique instantiation of a graph and generate its hierarchical attribute table. An example of a hierarchical attribute is wire delay.

```
node_pin.set_delay(float delay);
```

Listing 4.7: API example for LGraph hierarchical attribute setting

4.5 3rd Party Tools Integration

The integration of third-party tools into LGraph is intuitive. Most tools have APIs to construct netlists in their internal data structure. Thus, we can first create an object of the tool in the LGraph program, traverse the LGraph netlist and use the tool's API to build an equivalent circuit on the fly inside the object. Then we make the object perform its primary functions, for instance, synthesis. Finally, we map the tool's data structure back into LGraph. Currently, Mockturtle and OpenTimer are being integrated into LGraph as initial prototypes.

4.5.1 Mockturtle

LGraph uses Mockturtle's library for LUT-based synthesis [80]. We first partition combinational groups and map these groups from LGraph to Majority-Inverter Graph (MIG) [14] for synthesis. The synthesized MIG networks are then technology mapped to k-bit Lookup table (KLUT) networks and stitched back to LGraph.

4.5.2 OpenTimer

The synthesized LGraph then uses the integrated OpenTimer to perform timing analysis. Again we traverse the LGraph netlist and build the corresponding OpenTimer structure, compute timing inside the OpenTimer object and return the critical-path information.

4.6 Lesson Learned: The Deprecated but Fast Memory-Mapped Library

There are several important lessons we learned along with the LGraph-IR evolution. The lesson on using the memory-mapped library is the most important one.

LGraph is built with a live interactive design flow in mind. We originally designed a memory-mapped C++17 library for fast netlist load/unload. Modern SoC design usually constitutes hundreds of millions, even billions of logic gates. To quickly load/store such a large netlist, LGraph uses the memory mapping technique for fast persistence. The memory mapping technique maps a disk file directly to the virtual memory space and thus reduce the buffer copy operations. It has a speed advantage for large file processing [56]. We implemented a fast memory-mapped library called *mmap_lib* with basic data structures such as vector, hash map, bi-directional hash map, set, and tree. These fundamental containers form the skeleton of the code base in the first generation of LGraph. They were used extensively for constructing graph networks and attributes. As the program completes, LGraph's database is automatically

synchronized to the disk by the OS.

The memory-mapped LGraph works perfectly in the single-threaded LiveHD compilation and leads to a very fast code base as the evaluation 4.6.2 shows. However, the constant lock checks frequently that happen in the multi-threaded scenario slow down the overall compilation performance. In the end, we deprecated the memory-mapped LGraph library and use Google's abseil library [1,88] instead, which works fine in multi-threaded compilation though it is slower than the LGraph memory-mapped library using single thread.

4.6.1 Evaluation Setup

I compare the LGraph's memory-mapped library *mmap_lib* is compared with the C++17 standard library, Abseil C++ library [88], robin-map [6], and flat-hash map [7]. I randomly generated 100k numbers from a uniform random number generator to insert pairs of *uint32_t* key and value to a hash map; then, I erased the 100k elements randomly and measured the runtime. The vector and hash map flow are performed 100 times and I average the execution times.

The scalability of the alpha LGraph-Mockturtle LUT synthesis flow is also evaluated and is compared with the Yosys-ABC synthesis flow [1] targeting Xilinx 7-series FPGAs. I use simple combinational chains of gates ranging from 1 to 50K serialized concatenations.

All experiments are run on an Intel Core i7-6700K CPU @ 4.20 GHz with 16

[1] Commands include (1) read_verilog (2) proc (3) techmap (4) abc-lut 4, only the time of (3) and (4) are measured

GB of memory, running Manjaro v5.2.8-1. Tools are compiled with gcc v9.1.0.

4.6.2 Memory Mapped Library Results

LGraph Memory Mapping Library

I evaluate the access speeds on various sizes of vectors and hash maps. LGraph's *mmap_lib::vector* is 39.1% faster than *std::vector* in our simple test, and LGraph's runtime scales better. Figure 4.3 shows the average run time for writing and reading the entire vector on both LGraph *mmap_lib::vector* and C++'s standard vector. The comparison result between C++ hash map implementations and *mmap_lib::map* is shown in Figure 4.4.

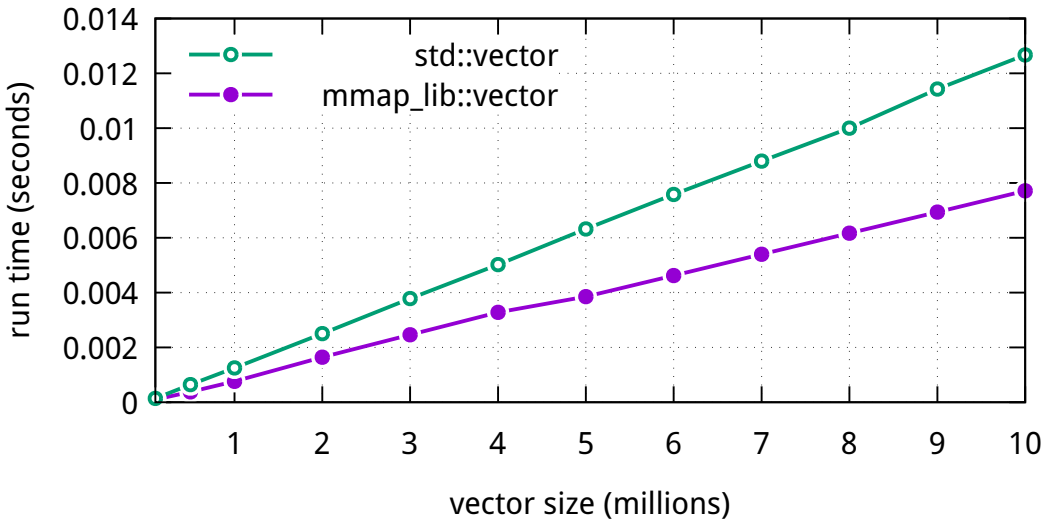


Figure 4.3: LGraph's mmap_lib::vector has faster run-time compared to the std::vector

The *mmap_lib::map* design shows a competitive speed among all competitors when the hash table size is under 10 million and starts to outperform others when the table size is in the order of 10 million, which is typically the size of a modern-day par-

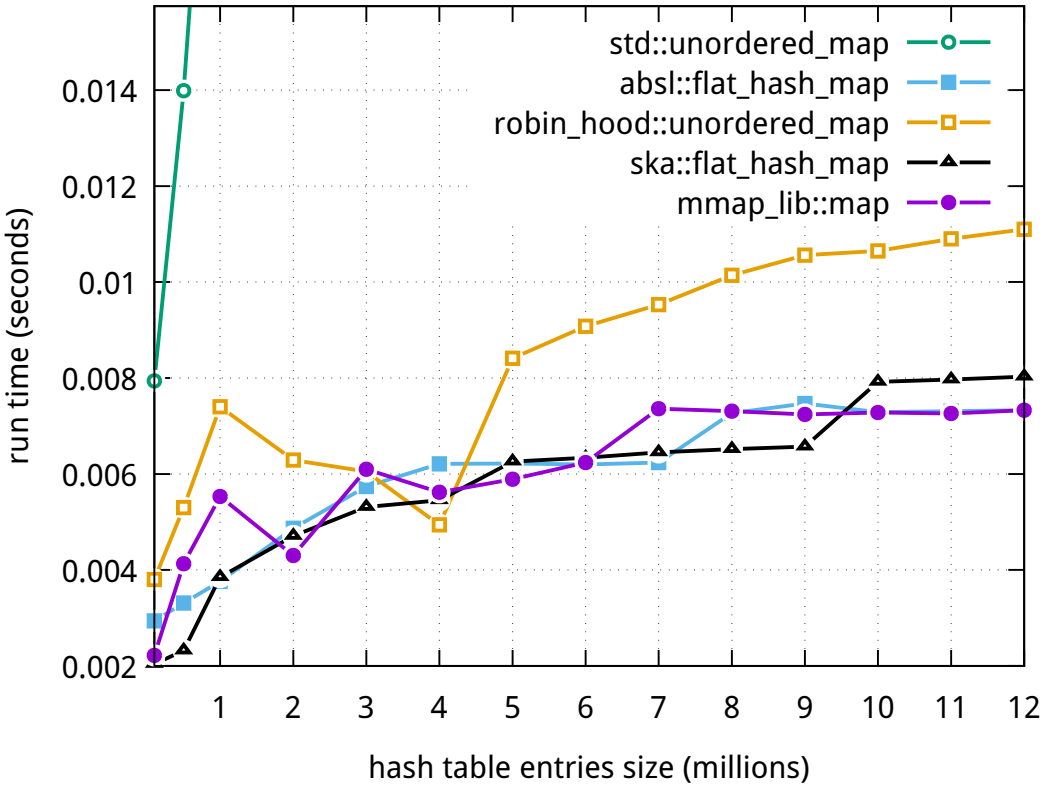


Figure 4.4: LGraph's mmap_lib::map is in par with best-in-class maps for entry sizes less than 10 million but faster for entry sizes in the order of 10 million

titioned VLSI netlist. It has a fast container access time, and LGraph's *mmap_lib* library also provides an extra advantage of data persistence. This would be the key feature when developing a live incremental VLSI flow as we do not have to re-parse the whole data repeatedly.

LGraph-Mockturtle LUT Synthesis Flow

The evaluation also examines the LUT synthesis scalability for flows of LGraph Mockturtle, Mockturtle only, and Yosys-ABC. The prototype of LGraph-Mockturtle LUT

mapping flow starts by converting LGraphs to MIG networks, synthesizing and mapping them to KLUTs, and converting the KLUT networks back into LGraphs. The Mockturtle-only flow is almost the same, but it excludes the conversion steps from and to LGraphs. For Yosys-ABC flow, only the execution time is measured from RTLIL to technology mapping and ABC LUT synthesis.

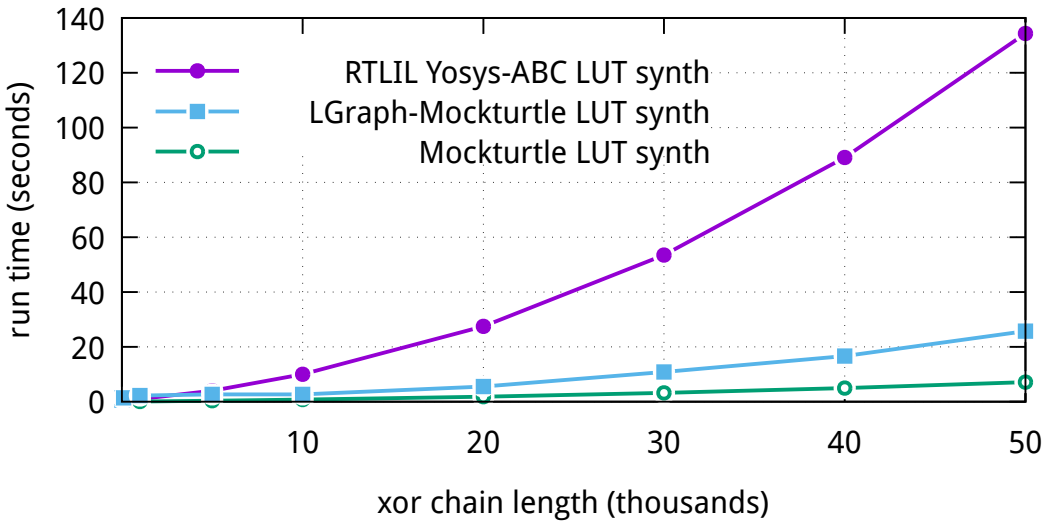


Figure 4.5: The LGraph-Mockturtle flow is faster than Yosys-ABC flow under all tested scenarios

Figure 4.5 compares the scalability of LUT synthesis among various flows such as LGraph-Mockturtle, Mockturtle only, and Yosys-ABC. Though the LGraph-Mockturtle flow is still in its early development stages; the runtime is undoubtedly better than the Yosys ABC flow. For a 50k combinational chain, it takes the Yosys ABC flow 134 seconds to finish, while it only takes our LGraph Mockturtle flow 25.7 seconds, an 80.8% speedup. When compared to the flow of Mockturtle only, there is an integration

overhead in our flow; the main reason lies with the prototype implementation: there are some network copy operations in the integration between LGraph and Mockturtle, which takes quite a bit of time.

4.7 Conclusions

In this chapter, I present the low-level LGraph-IR as a unified infrastructure for open-source EDA developers and an integrated design flow for hardware designers. The proposed features include a fast memory mapped library to avoid netlist reparsing, the hierarchical traversal function to enable the integrated tools to handle hierarchical design support automatically, and prototypes of 3rd party tools to express LGraph's generic integration power. Several LGraph-IR characteristics like the RLIS cells and all signed wires are also presented to address the unique challenges of hardware compilation.

Though being deprecated in the final proposed parallel compilation flow in Chapter 6, our results show that in the single-threaded compilation, LGraph's memory-mapped vector is 39.1% faster than C++ standard library designs. LGraph's memory-mapped hash map design is comparable to the best C++ open-source implementations. A working technology mapping flow with the integration of LGraph and Mockturtle for FPGA LUT synthesis is also shown, which is 80.8% faster than the Yosys-ABC flow and still has room to speed up further.

Chapter 5

LNAST: A High-Level Language Neutral AST-IR for Hardware Description Languages

In this chapter, I first discuss the research question that arises from modern HDLs and the original LGraph design in LiveHD. I then explain the LNAST model and its internal structure and node type design to interact with HDLs. I also describe the LNAST passes to/from the LiveHD framework. Finally, I evaluate the parsing performance of our new Pyrope-LNAST-LiveHD flow.

5.1 Introduction

In the past decades, hardware design complexity increased, and many designers and researchers have sought new hardware description languages to depict the de-

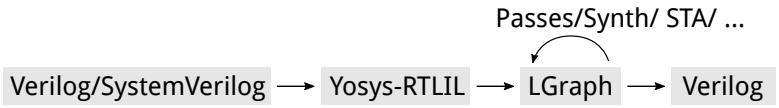


Figure 5.1: The original LiveHD only has LGraph-Yosys interface that handles SystemVerilog/Verilog.

sign structures better. Some representative high-level HDLs are Chisel3/FIRRTL [15, 45], PyRTL [24], MyHDL [83], and our in-house Pyrope HDL. These HDLs come with their unique compilation stack and lead to an isolated HDL compilation codebase. It is difficult to share the innovations and codebase among these HDLs compilers. Clearly, a generic compilation framework for HDLs is needed to build a healthier HDL compilation community.

In the beginning, the original LiveHD framework tried using LGraph directly as the common interface to compile high-level HDLs. Nevertheless, I found three bottlenecks that could potentially hinder our goal of LiveHD, i.e., a fast and generic compilation for HDLs and eventually achieving live compilation feedback.

First, LGraph could use Yosy to interface Verilog/SystemVerilog and generate its internal graph-like representation called RTLIL [89]. A translation layer converts the RTLIL to LGraph. Besides Verilog/SystemVerilog, the translation pass must convert other modern HDLs into LGraph to extend the usability of LiveHD. The reality is that the translations have many similarities across HDLs, and care must be taken to avoid code replication.

Second, the old LGraph-Yosys interface takes several minutes to elaborate a large design with millions of gates. This long translation time impedes our goal of fast HDL compilation. The multi-layer translation in the LGraph-Yosys interface contributes to this delay. To begin with, the Verilog/SystemVerilog AST translates to Yosys RTLIL, followed by the Yosys RTLIL *proc* command [1], and the translation ends with generation of LGraph. A translation interface with minimal overhead is needed to accelerate the front end.

Third, besides HDL compilation (elaboration), the future LiveHD aims to support both synthesis and simulation. To do so, it has to convert the high-level HDL code to support synthesis and generate fast C++ for simulation(similar to what Verilator [79] does). The LGraph model is a low-level graph representation for hardware design. The semantic gap between the high-level HDLs and the low-level LGraph IR must be bridged to ease the generation of human-readable C++ and other HDLs.

I introduce LNAST, a high-level IR, to bridge the gap between LGraph and multiple high-level HDLs. The combination of LNAST and LGraph in the LiveHD framework addresses the concerns raised earlier in this section.

LNAST plus LGraph is similar to Truffle and GraalVM [39], which also maintains a language-neutral AST for dynamically typed languages such as Python and JavaScript. GraalVM also has a low-level IR based on the LLVM IR.

Figure 5.2 depicts the current simplified LiveHD framework. LNAST replaces the original LGraph-Yosys interface, eliminating multi-translations from the previous

[1] Yosys proc is the command to translate high-level RTLIL to flops, muxes…

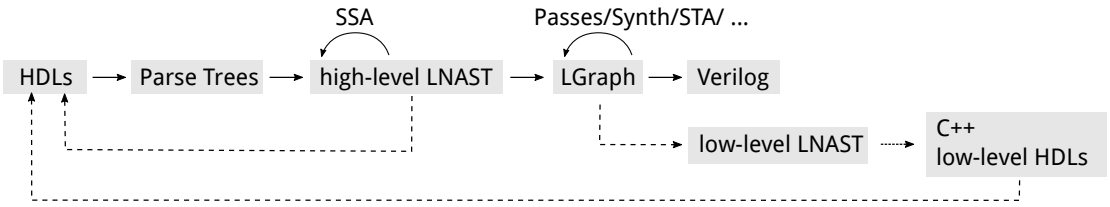


Figure 5.2: The new LiveHD flow with LNAST. The passes with physical lines indicates the contribution of this thesis. The dash-lines indicate the other students' projects that are still under-developed

model. The language-agnostic nature of LNAST IR helps LiveHD target traditional Verilog/SystemVerilog and modern HDLs. The simple but expressive node-type definition in LNAST provides a clear representation of modern HDL semantics. LNAST has a Static Single Assignment(SSA) [27] transformation to enable efficient conversion to LGraph. Though not the main contribution of this thesis, LNAST also facilitates the code generation back from LGraph to the high-level source programs.

The rest of this chapter is organized as follows: Section 2 describes the LNAST model and its internals. Section 3 reports the results. Section 4 reviews the lessons learned from the LNAST development. Section 4 concludes the chapter.

5.2 LNAST Model

5.2.1 Tree Structure

I use the Pyrope code in Listing 5.1 and its LNAST IR in Listing 5.2 to demonstrate LNAST structure. Listing 5.1 generates the most essential LNAST syntax. The code contains a bit width declaration of output %z in the top module (line 1), an *xor* operation of two top-scope inputs $a and $b (line 3), an *adder* function definition with sub module IO $c, $d, and %s (line 5 to 7), a register variable #y with initialization of constant 1 (line 9), an if-else control flow (line 10 to 14), and a function call on *adder* (line 16) with input assignment: variable x to submodule input $c, register #y to submodule input $d, and the function call return value is assigned to the top module output, %z.

```
1  %z.__ubits = 2            //bitwidth declaration
2                            //output %z
3  x = $a ^ $b               //input $a, $b
4
5  adder = ||{               //function definition
6    %s = $c + $d            //input $c, $d and output %s
7  }
8
9  #y = 1                    //register #y
10 if ($a > $b) {            //conditional flow
11   #y = $a & $b
12 } else {
13   #y = $a | $b
14 }
15
16 %z = adder(c=x, d=#y)     //function call
```

Listing 5.1: A sample circuit described in Pyrope

```
top    :
    stmts   :
        tuple_add:
            ref     : %z
            const   : __ubits
            const   : 2
        xor     :
            ref     : x
            ref     : $a
            ref     : $b
        fdef    :
            ref     : adder
            stmts   :
                plus    :
                    ref     : ___t2
                    ref     : $c
                    ref     : $d
                assign  :
                    ref     : %s
                    ref     : ___t2
        assign  :
            ref     : #y
            const   : 1
        gt      :
            ref     : ___t3
            ref     : $a
            ref     : $b
        if      :
            ref     : ___t3
            stmts   :
                and     :
                    ref     : ___t4
                    ref     : $a
                    ref     : $b
                assign  :
                    ref     : #y
                    ref     : ___t4
            stmts   :
                or      :
                    ref     : ___t5
                    ref     : $a
                    ref     : $b
                assign  :
                    ref     : #y
                    ref     : ___t5
        tuple_add:
            ref     : ___t6
            assign  :
                ref     : c
                ref     : x
            assign  :
                ref     : d
                ref     : #y
        fcall   :
            ref     : %z
            ref     : adder
            ref     : ___t6
```

Listing 5.2: An LNAST-IR structure example of the circuit in Listing 5.1

Listing 5.2 illustrates the tree-like structure of LNAST. Multiple modules can be represented in different LNAST files or in a single LNAST tree with function definitions representing submodules. Here I use the representation of the single tree.

The *top* and the *statements*(denoted as *stmts* in line 2) node make up the root scope of the tree. The children of *statements* are generated by following the source code order. A primitive operation generates a sub-tree with the operator as the parent, the first child as the lhs, and the other operands as rhs leaf. The *xor* operation (line 7) is such a primitive operation example. Another case of sub-tree generation comes from the scope hierarchy. For instance, a function definition or an if-else code block generates a new hierarchical sub-tree and defines its own *statements* node (line 13 and line 30). Every *statements* node creates a new program scope in the tree.

5.2.2 Attribute in LNAST Nodes

Each LNAST node has four attributes: *NAME*, *TYPE*, *LOC*, and *SUBS*. *NAME* points to the variable name in the source code, *TYPE* is the node type in LNAST definition, and *LOC* denotes the line of code. *SUBS* is the subscript of a *NAME*, and it aids in SSA transformations.

5.2.3 Neutral Node Types for HDLs

LNAST node types are intentionally designed to capture properties shared across different HDLs and aim to maximize expressibility. Node types are categorized into four groups: *structural*, *variable*, *primitive_op*, and *tuple*.

The *structural* group forms the skeleton of LNAST and presents the program control flow. They define major node types like *top*, *function_definition*, *if*, *uif*,*for*, and *while*. These nodes compose *statements* to declare a new program scope. Two assignment node types: *assign* and *dp_assign* are used to specify values for the lhs variable. The *dp_assign* is quite useful when the lhs is a register where the rhs value has to be constrained to the lhs bitwidth. If the original rhs had larger bits than the lhs defined, the lhs would simply keep its original bitwidth.

The *primitive_op* group includes operations common across different HDLs. For instance, many HDLs have similar logical, arithmetic, and comparison operations. A specific example is the addition operation.

The variables in the source code are classified as constants or references by the *variable* group. Notice that there are no specific type definitions for circuit input, output, and register. Special characters prefix these circuit components. For example, $\$x$ denotes that x is a module input, $\%y$ is a module output, and $\#z$ is a register.

The *tuple* group helps LNAST to construct tuple variables. In LiveHD, the attribute assignment is achieved by a tuple struct. For example, in the first line of the source code in List 5.1, the variable's bit width attribute is expressed as an *tuple_add* with an attribute string of *__ubits* to identify the unsigned bitwidth. This attribute could be retrieved by the *attr_get* operator when needed.

When considering the design choice, such as global type inference and scope in LiveHD, it is almost impossible that create the most generic IRs of semantics and syntax for all HDLs. To overcome this problem, LiveHD allows extensions with at-

tributes in the IRs passes. For example, if a new language wanted to have dependent types [25], LNAST could annotate the types with attributes, and a new compiler pass at the LGraph level could enforce the refining types.

Table 5.1 Four groups of operators in the LNAST IR

group	operator	functionality
structural	*top*	identify the LNAST top node
	stmts	declare a new scope
	if	control flow
	uif	evaluate condition in any order
	for	for loop structure
	while	while loop structure
	func_def	define a new function as a submodule
	func_call	instantiate a submodule instance
	assign	variable assignment
	dp_assign	variable assignment but respect the lhs width
	phi	phi nodes created from SSA algorithm
primitive	*bit_and/or/not/xor*	bitwise logic operation
	reduce_or	or all bit position values together
	logical_and/or/not	logical operations
	plus	n-ary addition
	minus	n-ary subtraction
	mult	n-ary multiplication
	div	n-ary division
	mod	modular operation
	shl	logical shift left
	sha	logical shift right
	set_mask	turn a node from unsigned to signed
	get_mask	get signed value from a given bit position
	is/ne/eq/lt/le/gt/ge	comparison operators
variable	*ref*	referece to a declared variable
	const	reference to a constant number or string
tuple	*tuple_concat*	concatenate 2 tuple struct
	tuple_add	add a field to high-level tuple struct
	tuple_get	get a field to high-level tuple struct
	attr_get	set an attribute to a variable

5.2.4 Function Call in LNAST

In addition to representing the submodule instantiation, the *function call* LNAST node also plays an important role for LNAST in interfacing different HDLs semantics. It is impossilbe for the LNAST node types presented in 5.2.3 can capture all the HDLs semantics. For example, the *tail_op* in FIRRTL HDL extracts LSB n-bits from its input edge. However, without a FIRRTL-specific bitwidth analysis pass, LNAST cannot know the input size of *tail_op*, and therefore cannot map the *tail_op* into an appropriate LNAST primitive op. LNAST circumvents this issue by mapping this kind of operator into black-boxed function calls. Later LiveHD will initiate a mid-end LGraph pass that collects the FIRRTL bitwidth information. Then, all the black boxed functions will be resolved to proper LGraph nodes.

The memory model is another example. There are several different memory models and parameters from different HDLs. It is inefficient to make all these memory types have a one-to-one LNAST memory type. Again, I map all kinds of memory models to the black-boxed function call in LNAST, and then resolve the appropriate memory peripheral parameters at the LGraph phase.

5.2.5 Scope Flexibility

Different HDLs have different scope designs. In general, I try to be sufficiently generic to efficiently cover different HDLs like Verilog, Pyrope, and FIRRTL. For example, Verilog and Pyrope can have local and module/function scopes. The design of LNAST variable scopes resembles Verilog and Pyrope. Statements from the root node

are in the top scope. By creating a new sub-LNAST tree, LNAST could have an additional scope hierarchy inside a module (or a function). In a way, the scope representation in LNAST is a superset of the FIRRTL language. This is because, in FIRRTL, the scope is per module, which needs the variables to be instantiated at the module level. Therefore, LNAST could also handle the FIRRTL module scope naively.

5.3 LNAST Transformations in LiveHD

5.3.1 From HDLs to LNAST

In compiler design, a parser is used to scan the source code tokens and generate the parse tree. In LiveHD, different HDLs' source code is parsed into a language-specific parse tree and translated into LNAST IR, as illustrated in Figure 5.2. The difference between the LNAST and a parse tree is that LNAST focuses more on representing useful abstract information from the components of the source code, such as conditional loop blocks, whereas a parse tree captures low-level details of syntax, for instance, brackets and parentheses.

There are different front-end parsers for each corresponding HDL in LiveHD. To avoid code replications, these parsers and LNAST share a common tree library in LiveHD for building the tree data structures.

A Pyrope parser in JavaScript was implemented and generates the control flow graph (CFG) text in the three addresses format [26]. This is designed by another Ph.D. alumni in Micro-Architecture Lab, Akash Sridhar. Continued from this CFG text, the

current working prototype flow implements a CFG parser using the common tree library to generate the corresponding LNAST.

As speed is the priority for the LiveHD back end, we decide to implement a new C++ Pyrope parser for faster parsing time. The new parser follows the same grammar rules as the JavaScript version. This new parser also uses the tree library in LiveHD for building the parse tree. The transformation from the parse tree to LNAST IR also leverages the same tree library. This is an MS thesis project led by Kenneth Mayer, an MS student alumni in our lab.

I also design the interface between Chisel/FIRRTL and LNAST/LiveHD. This is feasible by taking advantage of Protocol Buffers [4] from Google. Protocol Buffer is a flow to compile a well-defined data format into target language classes with setter and getter methods. The FIRRTL IR design team leverages the Protocol Buffer to make parsing, serialization, and interfacing FIRRTL much more accessible for developers. I transform the FIRRTL IR into LNAST IR by interfacing both the APIs of the protocol buffer FIRRTL APIs and the LNAST APIs.

Currently, we build SystemVerilog parse tree with slang [8], and then transform the parse tree to LiveHD flow through LNAST API. The goal is to make LNAST represent fully synthesizable Verilog code, not just the netlist syntax.

5.3.2 From High-Level LNAST to HDLs

A key motivation for conceiving the idea of LNAST is to support multi HDL code generation. This pass is still under development. In Figure 5.2, notice that high-

level LNAST can generate HDLs, but it does not have to go through the LGraph IR to achieve this. As shown by the dotted arrow, there could be a closed-loop transformation from the HDL to a language-specific parse tree, and then to LNAST, and back to the HDL directly without entering the LGraph stage. This shortcut transformation loop is useful for verifying the correctness between LNAST and HDLs quickly.

5.3.3 From LNAST to LGraph

The graph-based representation of LGraph IR is virtually the same as SSA. SSA is a widely used compiler optimization technique. It assures that every variable in the IR is assigned precisely once and describes the use-def chains explicitly, which in turn helps the LGraph optimization passes. To bridge LNAST to LGraph easily, I perform the SSA transformation on LNAST after building the primitive LNAST. Different SSA definitions from the same variable are represented in the SUBS (subscript) field in different LNAST nodes.

5.3.4 From LGraph to Low-Level LNAST

There is an ongoing project on translating LGraph IR back to LNAST IR. It will be the stepping stone for performing HDL code generation from the synthesized LGraph. However, structural information could be lost when the lower-level LGraph IR expands from higher-level LNAST IR. Take the conditional loop as an example. The loop node in LNAST is flattened in the LGraph Data Flow Graph (DFG) analysis. Therefore, the potential challenges should be identifying the corresponding LGraph region

and tagging the associated LGraph nodes, which would be helpful when folding the loop region back to an LNAST node.

5.4 Lesson Learned

5.4.1 The Deprecated CFG and Yosys to LGraph passes

Before the invention of LNAST-IR, we used to have a JavaScript parser that converted Pyrope HDLs into a control flow graph (CFG) encoded with three-address code [13]. The new scope is defined not only by the submodule, but also by the CFG, which contains hierarchical scopes from if-else or loop statements. Hierarchical LGraph-IR, on the other hand, is only appropriate for representing the submodule scope by creating a hierarchical sub-graph. To capture the statements in the if-else scope, LGraph must first parse the entire CFG to understand the scope hierarchy, record all the def-use chains from the CFG, then perform the SSA for all statements and generate SSA phi-nodes on the fly, which is time-consuming in a low-level IR like LGraph. Furthermore, the JavaScript parser can only handle the Pyrope front-end, so we had to interface Verilog/SystemVerilog with other tools like Yosys. Also, we had no way of dealing with CHIRRTL at the time. After a year of experimenting with various implementations, we realized that a high-level tree IR like LNAST is required to help us easily capture the scope and produce SSA. The old Pyrope JavaScript parser and Verilog Yosys parsing paths are completely deprecated in the new LiveHD.

5.4.2 The Deprecated Persistence

In the first version of LNAST's design, we placed a heavy emphasis on the speed of LNAST's data persistence. LNAST used memory-mapping like LGraph IR to speed up reads and writes. In LNAST, the source code was memory-mapped onto virtual memory. Lexing was done to tokenize the source code, and these tokens were stored in a memory-mapped vector. To record the token of the *name* field in LNAST nodes, we stored the index in the memory-mapped vector instead of the plain string. It ensured that the strings were not manipulated directly and avoided additional memory operations.

This memory-mapped string vector will not work in the multi-threaded LiveHD design. For the same reason discussed in the previous chapter, the lock contention in the memory mapped version slowed the parsing and LNAST construction speed. Therefore, we deprecate the memory-mapped design and use google's tcmalloc [37] library to help allocate memory for the token strings faster in the multi-threaded compilation.

5.5 LNAST Evaluation

5.5.1 Setup

The target circuits are simple adder chains ranging from 20,000 to 200,000 gates. I generated the equivalent Verilog and Pyrope adder chain circuit. Each add operation takes one line of code. LiveHD is implemented with C++17 and compiled

with GCC 11.0.3. I ran our experiments on a server with 32-Core AMD EPYC 7542 Processor at 3.4GHz, 504G memory, and Kali Linux 5.14.0-kali2-amd64 installed.

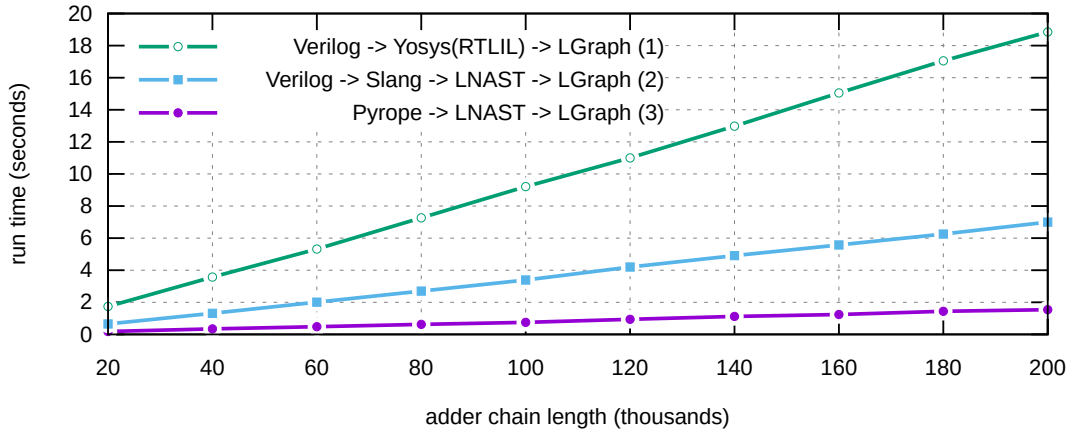


Figure 5.3: LiveHD flow with new LNAST design is significantly faster than the old one for tested circuits.

5.5.2 Results

I compare the old LiveHD flow in Verilog and the new LiveHD flow in Verilog and Pyrope. The old flow uses Yosys to elaborate a Verilog file into RTLIL and then converts the RTLIL to LGraph. The new flow uses slang as the parser to build a parse tree, then converts the parse tree into LNAST-IR, and then translates into LGraph IR. Another new flow uses a Pyrope parser implemented in C++ to create the parse tree, converts the parse tree text to LNAST, and transforms the LNAST to LGraph.

Figure 5.3 presents the runtime comparison among (1) the old Verilog-Yosys-

LGraph flow; (2) the new Verilog-Slang parser-LNAST-LGraph flow, and (3) the new C++ Pyrope-Pyrope parser-LNAST-LGraph flow. For a 200k adder chain circuit, the path with Yosys takes 18.85s in total; the new Slang path completes in 7s, which is 2.7x speedup. Finally, the Pyrope path takes 1.54s to handle an equivalent Pyrope circuit and translate it to LGraph, leading to a 12.4x speedup on parsing and IR-lowering. On further observation, we notice that both new Pyrope and slang interfacing paths demonstrate better scalability with increasing size of the target design.

5.6 Conclusion

This chapter proposes the high-level language neutral AST IR, LNAST, as the new generic interface to bridge Verilog, CHIRRTL, and Pyrope HDLs. Together with other high-level generic operators, black-boxed function definitions, and function calls enable LNAST to capture any new operation semantics from new HDLs. I demonstrate how to use this black-boxed feature to translate the FIRRTL HDL into the LiveHD framework. The tree-like structure eases the effort to represent the hierarchical scopes in the control flow graph. Two precious lessons learned related to the design of LNAST are also discussed. The new LNAST with a proper front-end parser shows as high as 2.7x-12.4x speed up on parsing and LNAST to LGraph IR-lowering time.

Though not the main focus of this thesis, the code generation ability from LNAST can also be leveraged and further open the research gate of HDL compilation test coverage.

Chapter 6

A Fast Parallel Compilation Framework for HDLs

6.1 Introduction

Hardware design uses custom Hardware Description Languages (HDL) and compiler tools. Although Verilog is still the most popular HDL, it shows its age [1] and alternatives like Chisel3/FIRRTL [15,45], PyRTL [24], and Pyrope [76] have gained popularity. Typically, each HDL is bundled with a compiler that converts its high-level code into Verilog output. Verilog is frequently viewed as the assembly code in software compiler stack.

Compilation time is a crucial parameter in any programming language, and HDLs are no different. HDLs are primarily used in ASIC/FPGA fabrication and simulation. Fabrication flows require synthesis, place&route which are inherently slow.

[1] The original Verilog was designed in 1983, and current compiler are semantically compatible with it.

This chapter aims to accelerate HDLs compilation, the elaboration step in fabrication before synthesis, as well as the first step in the simulation.

The ideal way to speed up the HDL elaboration step is to create a fast/parallel compiler flow that can manage multiple HDLs. The compilation flow should be extensible and allow new languages to leverage to construct a fast/parallel compiler automatically. In the open source community, only the concurrently designed CIRCT [10, 32] compiler has some parallelism. The popular open-source HDL compilers (FIRRTL, Yosys [89]) parallelized.

This thesis proposes a new HDL compilation framework to address the issues raised from elaboration and new HDLs. The key contributions of this chapter are as follows:

1. I design a fast parallel multi-HDLs compiler where all the compilation steps can be done in parallel.

2. I implement a proof-of-concept compiler (LiveHD) that supports Verilog, Pyrope, and CHIRRTL (the highest-form of FIRRTL). The compiler is faster than the existing open source alternatives in all cases.

LiveHD is a multi-threaded, multi-HDL fast compiler. Compared with traditional non-HDL compilers, a parallel HDL compiler must address the issue of lacking *import* language feature. The problem is that a file processing order exists in some languages, such as Verilog. A file can be accessed without any declaration modules defined in other files as long as there are no circular dependencies.

This is comparable to C++20 *modules* though not exactly the same. Prior to C++20, the preprocessor would add the contents of included header files to the source code before compiling it. As a result, the compiler would know all function caller-callee relation and compile files in parallel. C++20 necessitates a pre-scan phase before the main compilation to comprehend the dependencies and interfaces. Since HDLs lack an import directive, the pre-scan must handle every file and make inferences across them.

In the proposed LiveHD compiler, there is no additional required pre-compilation scan step. The design relations are dynamically resolved by constructing a dependency tree during the internal IR generation phase.

The dependency tree directs the compiler on how to apply parallelism. Some passes are not embarrassingly parallel [42]. For these passes, LiveHD references the dependency tree to select independent modules and compile with a parallel bottom-up pass.[2] The selection starts from the dependency tree leaves. A parent module can start to be a candidate once all of its children have been processed. This strategy is what we refer to as *bottom-up parallelism*. The input/output connections of a sub-module instantiation is an example that requires correct compilation order from callee to caller. Conversely, for the passes where the caller and callee are independent, LiveHD can compile them parallelly in any order; which I define as *full parallelism* in this thesis.

LiveHD seeks to enable multiple languages to benefit from the parallel compilation. Each HDL has a bitwidth specification in addition to language semantics. It is a challenge that generic multi-HDLs compilers like LiveHD, CoreIR [60], and LLHD [73]

[2] A parallel top-down is also possible, but is is not needed for how the HDLs implemented.

need to address. For instance, Verilog sets bits on every variable, while higher-level HDLs like FIRRTL and Pyrope may only define bitwidth on the I/O, and the compiler must propagate the bit inference globally throughout modules. The front-end IR of LiveHD does not need a bitwidth set on variables. Therefore, LiveHD can directly bridge these languages to its front-end IR before conducting a bitwidth inference process. On the contrary, LLHD and CoreIR require that every variable have its bitwidth explicitly set. To handle high-level languages like FIRRTL and Pyrope, each language needs a custom compiler pass to infer bitwidth because it is part of the language semantics. It is somewhat similar to global type inference in compilers, except it only performs bitwidth inference. Once more, LiveHD makes this pass parallel with a bottom-up mechanism.

Our results show that when compiling the highest level of FIRRTL language, CHIRRTL, LiveHD is 3x to 6x faster than the original FIRRTL compiler in the single-threaded compilation and 16.5x to 46.6x faster in the 16-threaded mode. Compared to Yosys [89] for Verilog parsing and regeneration, LiveHD gains 8.6x and 71.3x speedup, respectively, with 1 and 16 threads. LiveHD achieves high scalability because all the compilation steps are parallel for languages like Pyrope.

6.2 Parallel Compilation

This section discusses the parallel compilation pipeline and how each pass ensures parallel scalability.

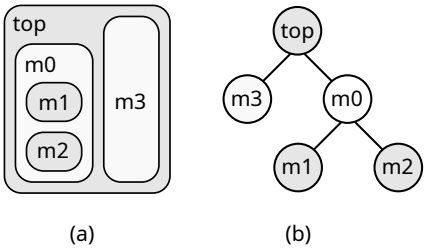


Figure 6.1: The LiveHD parallel compilation example with a hierarchical FIRRTL frontend. Module *m3* is significantly larger than the others. The vertical dashed lines represent the Synchronization barriers.

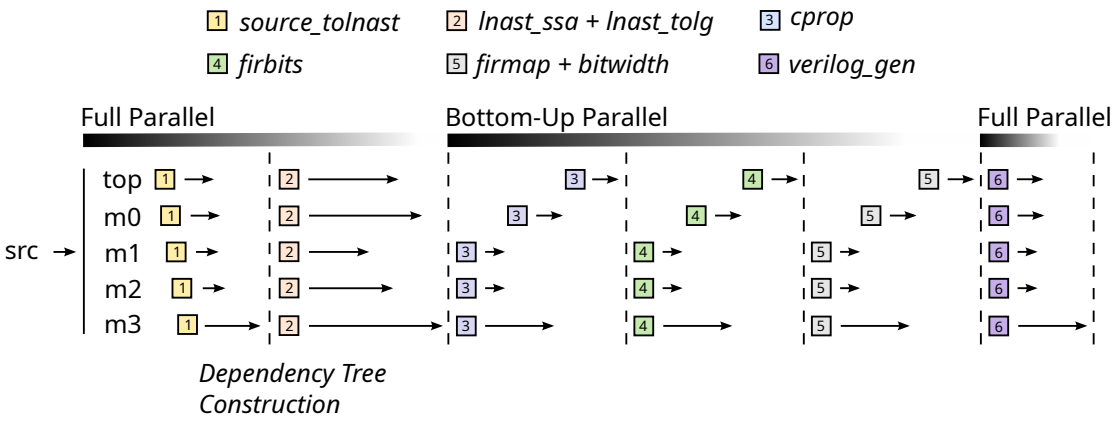


Figure 6.2: The LiveHD parallel compilation example with a hierarchical FIRRTL frontend. Module *m3* is significantly larger than the others. The vertical dashed lines represent the Synchronization barriers.

The unit of parallelism of LiveHD is a module. A finer grain granularity will require many locks shared within module resources, potentially complicating/slowing down single thread performance. Modern large system-on-chip HDL programs have

millions of lines of code spread over hundreds or even thousands of modules. Utilizing parallel compilation is a fruitful opportunity to gain compilation speed.

A compiler pass is fully-parallelizable when all the modules are functional independent. The pass can operate on all modules in parallel in any order, and higher parallel scalability can be attained as long as the compiler allocates more threads. Nevertheless, not all passes can achieve this optimal parallelism. Functionally dependent caller-callee modules in a pass must adhere to a dependency order and cannot be compiled in parallel. To further extract more parallelization from this type of pass, the compiler must examine the dependency relations and select independent modules in order to process in parallel.

6.2.1 Dependency Tree

In HDLs, the hierarchy of all module instantiations can always be represented as a dependency tree structure (Figure 6.1-a, 6.1-b). In LGraph, a sub-module instance is represented as a node with a sub-graph type. The sub-graph could point to the other graph. LiveHD uses depth-first search (DFS) to recursively traverse into the sub-graph nodes and construct the dependency tree from the specified top module. These sub-graph nodes have been recorded separately during LGraph construction. The DFS traversal only visits these sub-graph nodes without traversing all nodes in the LGraph.

In the dependency tree, the leaf module instances must be functionally independent because they have no direct I/O connections. Thus, for a non-fully-parallelizable pass, LiveHD exploits the ***bottom-up parallelization*** mechanism. It starts by compil-

ing the tree leaves in parallel and then immediately allocates a new thread task for the parent module once all its children have been processed.

A pass could benefit from a top-down approach instead of the bottom-up one. In the passes implemented, we only need fully parallel or bottom-up. A top-down approach can be added if a problem is easier to solve.

In an HDL program, a module may be instantiated more than once. In LiveHD, the instantiations of the same module are represented as the same LGraph, but they are viewed as different nodes in the dependency tree. In order to prevent redundant compilation on module instances, LiveHD implements tracks already optimized Lgraphs and avoids redundantly compiling the same module multiple times.

If accessing some global objects is mandatory in a pass, functionally independent modules also necessitate a mutex lock to acquire ownership of global objects, thus guaranteeing access safety. Nonetheless, in parallel programming, passing the mutex lock between threads to protect global data is an expensive action. This is because each thread's critical parts will slow the execution flow in turn. Still, this is a minor issue in the LiveHD compiler because using LGraph IR minimizes such overhead. LGraph IR maintains a graph library to manage basic information like the graph name, graph IOs, and the dependency tree for all the LGraphs. This library is a global object that needs mutually exclusion.

6.2.2 Parallelism in Compilation Passes

This section presents LiveHD's compilation stack in Table 6.1 and discusses why and how each pass exploits either *full* or *bottom-up* parallelism for the circuit modules. One important technique in the LiveHD compiler is to defer the bottleneck from functional dependency as late as possible so that more front-end passes can be fully parallelized.

Since there are several passes with different degrees of parallelism, LiveHD implements a thread pool where tasks are executed independently of each other. To avoid waiting for all the tasks to be completed, LiveHD tracks the call dependency tree. For bottom-up parallelism, when a child node finishes, it checks if all the siblings are done; if so, it calls the parent code. This is achieved with a simple atomic counter per node.

Fully parallelized LNAST construction

Based on the front-end HDL, LiveHD first decides the functionality of *source_tolnast* (see Table 6.1) and converts an HDL program into LNASTs. An HDL program may have many source files, and each file may contain several hardware modules. If more than one module is defined in a single source file, LiveHD will create new *source_tolnast* threads to handle each module separately. Since *source_tolnast* function merely maps the parse tree of a module into the corresponding LNAST, there is no dependency between the executions of the threads. Thus *source_tolnast* is a *full parallelizable* pass.

Table 6.1 The LiveHD passes in the compilation order.

Name	Functionality	Parallelization Type	
		Full	*Bottom-up*
source_tolnast[1]	Verilog to parse tree to LNAST[2]	✓	
	Pyrope to parse tree to LNAST	✓	
	CHIRRTL protobuf to LNAST[3]	✓	
lnast_ssa	SSA transformation for LNAST	✓	
lnast_tolg	LNAST to LGraph translation	✓	
cprop	Copy propagation[4]	(✓)	✓
	Dead code elimination[4]	(✓)	✓
	Constant propagation[4]	(✓)	✓
	Peephole optimization[4]	(✓)	✓
	Attribute resolving		✓
	Tuple struct resolving		✓
	I/O construction		✓
firbits[5]	FIRRTL operator bitwidth analysis		✓
firmap[5]	FIRRTL and LGraph operator mapping		✓
bitwidth	Bitwidth inference and optimization		✓
verilog_gen	Back-end Verilog code generation	✓	

[1] choose one of three functions based on the front-end HDL [2] Verilog has a serial *liveparse pass to split files* [3] *CHIRRTL has a serial protobuf deserialize step* [4] *could be full-parallelized, but here are merged in cprop with a bottom-up* [5] *FIRRTL-only passes*

Fully parallelized LNAST-SSA

After the LNASTs are constructed, LiveHD will spawn *lnast_ssa* thread tasks to translate every LNAST into SSA form. Although there might be sub-module instantiation statements in the LNASTs, since SSA transformation only focuses on the return value and inputs arguments of the sub-module, the internal content of the sub-module does not affect the parent module's SSA. Therefore, modules in the tree hierarchy are inde-

pendent regarding *lnast_ssa* and can be handled full-parallelly.

Novel uIO Techniques for Fully Parallel IR Lowering

In the *lnast_tolg* pass, the functional dependency issue arises when there is a sub-module instantiation in the HDL program. Figure 6.3 shows that in LGraph, a sub-module is shown as a sub-node with inputs and outputs connected to the parent module graph. From the parent point of view, connecting an edge to the corresponding sub-node input requires the knowledge of all sub-module I/O in the graph library. However, when the *lnast_tolg* is multi-threaded, all LNASTs will execute the *lnast_tolg* pass in a random order. In this case, the graph library cannot guarantee that the submodule's I/O information will be ready when the parent needs it.

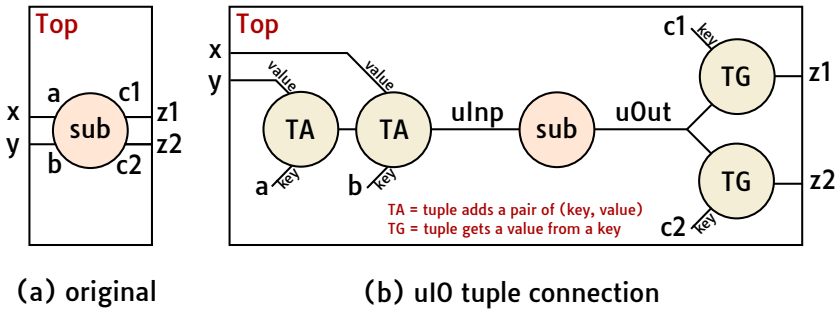


Figure 6.3: A sub-module instantiation in top-module. The sub-module has inputs (a, b) and outputs (c1, c2). LiveHD aggregates these I/O as tuple uInp/uOut to isolate functional dependency while connecting the top and sub at the *lnast_tolg* pass.

LiveHD solves this issue by proposing a novel technique that uses unified input (uInp) and unified output (uOut). An uInp or an uOut is an LGraph tuple structure

used to aggregate inputs or outputs, as shown in Figure 6.3. The uInp and uOut are the only input and output for each module during the *lnast_tolg* step.

As the *lnast_tolg* iterates through the parent LNAST, if there is a sub-module instantiation statement, the LiveHD graph library will check and try to create a sub-graph skeleton with the uIO atomically. After that, regarding the input edges of the sub-module node, the parent first creates tuple-add (TA) operators to collect all the edge driver pins as the tuple fields and connect this tuple to the sub-module uInp. On the other hand, if the parent module tries to connect edges from the sub-module outputs, the parent graph creates tuple-get (TG) operators and fetches fields from the submodule uOut tuple. LiveHD creates these uIO tuple structures around the sub-module to isolate the dependency between parent and child graphs. The uIO resolving process is deferred until the *cprop* pass, where all of the program tuples are handled together in a single graph traversal. So, the *lnast_tolg* pass becomes fully parallelizable.

Merged passes with bottom-up parallelism

LiveHD implements four classical software compiler optimizations currently: copy propagation, dead code elimination, constant propagation, and peephole optimization (*CDCP*). The algorithm starts with the module inputs to traverse the graph locally for each optimization. Theoretically, the four passes in *CDCP* could run with the full parallelism.

However, graph traversal is a time-consuming action. If LiveHD performs the passes of *CDCP* fully parallelized, four individual graph traversals will be required.

I seek the opportunities to merge multiple passes in a single graph iteration to save the iteration time. Therefore, in LiveHD implementation, the four passes of *CDCP* are merged with other bottom-up parallelized passes to minimize the graph traversal to just one iteration. LiveHD currently merges seven functions into a single *cprop* pass as listed in Table 6.1.

In the merged *cprop* pass, attribute resolving, tuple resolving, and I/O construction are three hardware-specific functions required by all HDLs. These functions are all tuple-related and have to be parallelized in a bottom-up manner. This constraint exists because the connection around the sub-module instantiation node needs to be resolved by flattening the uIO tuple. Then the parent module can continue the rest of the algorithm propagation.

Other bottom-up parallelized passes

firbits, *firmap* and *bitwidth* are the other three bottom-up parallelized passes. The reason is that their algorithms require the sub-module outputs attribute to be ready when the parent graph traversal visits them.

After *firbits*, an important optimization is that a single bottom-up task performs the *firmap* and *bitwidth* passes for each module. This increases cache locality and scalability because it guarantees the same lgraph to be mapped to the same CPU.

Verilog code generation

verilog_gen is the final stage of the LiveHD compilation pipeline. Since the functional dependencies between hierarchical modules have been fully resolved from the previous

LiveHD passes, LiveHD can run *verilog_gen* with full parallelization.

6.3 Multi-HDLs Compilation

Currently, LiveHD can compile HDLs of Verilog, CHIRRTL, and Pyrope. All these languages can benefit from LiveHD's fast and parallel compilation. This section discusses the essential characteristic of each HDL and how they are integrated with LiveHD.

6.3.1 Parallel I/O Pass

Ideally, we want to perform each compilation pass with full parallelism for every HDL. However, as discussed in section 1, the type of parallelism that can be achieved is determined by (1) the modules' I/O definition and (2) how a module is instantiated by a caller. This subsection discusses each HDL's instantiation scenario.

Verilog

Listing 6.1 provides an example of Verilog sub-module instantiation. It is important to note that, while the statement (line 6) expresses the instance connections, it does not show the direction of each sub-module I/O. This I/O connection syntax requires Verilog to be compiled in the bottom-up manner to collect sub-module I/O information, which the top-module can then utilize to resolve instance connections.

```
1 module Sub(input inp, output out);
2   assign out = inp | inp;
```

```
endmodule

module Top(input inp_t, output out_t);
  Sub sub(.inp(inp_t), .out(out_t));
endmodule
```

Listing 6.1: A Verilog module instantiation example

CHIRRTL

In CHIRRTL, no I/O information is provided at the instantiation statement (line 7 of listing 6.2), so compilers have to figure it out from the left/right-hand sides of subsequent statements (line 8 and 9 of listing 6.2) that exploit the instantiation.

```
1 module Sub:
2   output io: {flip inp: UInt<1>, out: UInt<1>}
3   node _T = or(io.inp, io.inp)
4   io.out <= _T
5 module Top:
6   output io: {flip inp_t: UInt<1>, out_t: UInt<1>}
7   inst sub of Sub
8   sub.io.inp <= io.inp_t
9   out_t <= sub.io.out
```

Listing 6.2: A CHIRRTL module instantiation example

Interestingly, the Chisel front-end compiler has resolved all of the hierarchical I/O connections from Chisel3 code and it could output a dependency tree before generating the FIRRTL file. Because all hierarchical I/O connections are resolved, Scala-FIRRTL could theoretically use the dependency tree to compile the FIRRTL IR in full parallel.

Pyrope

In Pyrope, the function arguments are the submodule input, and the return value is the sub-module output, as shown in line 7 of listing 6.3. However, the instantiation connections cannot be made as the top module cannot know whether a tuple or a scalar data type is returned when the sub-module is not handled yet. Thus, a bottom-up parallelization is needed to resolve the submodule instance connection.

```
1 //top.prp
2 sub = ||{ //the sub-module syntax in Pyrope
3   %out1.baz = $inp.foo + $inp2 //$ means input
4   %out2     = $inp.bar + $inp2 //% means output
5 }
6 //instantiation
7 ret = sub(inp = (foo = 3, bar = 2), inp2 = 4)
8 %out = ret.out1.baz + ret.out2
```

Listing 6.3: A Pyrope module instantiation example

6.3.2 Bitwidth Pass

The specification of bitwidth representations varies between HDLs. This subsection explains how LiveHD handles these specification variations in Verilog, CHIRRTL, and Pyrope.

Generic Bitwidth Inference Pass

In Verilog, bitwidths are defined for every variable, but this is not necessarily true in CHIRRTL or Pyrope, as the bitwidth may only be defined on module I/O. LiveHD generically handles these HDLs by leveraging the benefits of LNAST and LGraph IR.

Both IRs do not require a HDL variable to have bitwidth defined. Instead, if a module I/O's bitwidth is properly defined, LiveHD will refer to a bitwidth optimization algorithm [82] (*bitwidth* pass in Table 6.1) to initiate a propagation from the module I/O and calculate the optimized bitwidths for each visited wire.

Customization for implicit HDL specification

Language operators may implicitly present part of the bitwidth specification in an HDL. Table 6.2 shows such examples in the FIRRTL language.

Table 6.2 FIRRTL bitwidth management operators

FIRRTL Operator	Functionality
bits_op	extract a value with a specified bit range from the input edge
head_op	extract a value of MSB n-bits from the input edge
tail_op	extract a value of LSB n-bits from the input edge
cat_op	concatenate two input edges

The bitwidth of edges in these FIRRTL operators must be known to be mapped into LGraph cells. For example, a CHIRRTL *head_op* can be mapped to the *shift_right_op* in LGraph, but the exact shift amount depends on the driver operands of *head_op*. This prerequisite raises an interfacing difficulty for hardware IRs because the entire FIRRTL design bitwidth information must be collected somewhere.

Because LiveHD is a pass-modularized framework, these challenges can be addressed easily by plugging-in HDL-specific passes. The CHIRRTL front-end is handled by two CHIRRTL-specific passes, *firbits* and *firmap* (see Table 6.1), to handle the

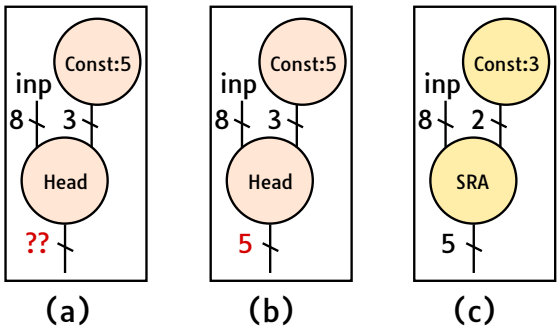


Figure 6.4: Idea illustration of FIRRTL bits analysis and FIRRTL-LGraph mapping passes. The FIRRTL *head_op* extract the MSB 5-bits from the *inp* signal. (a) a FIRRTL-equivalent LGraph with an FIRRTL *head_op*. (b) the LGraph after *firbits* analysis. (c) mapping to LGraph *shift_right_op*.

CHIRRTL front-end. LiveHD first individually translates FIRRTL operators to special LGraph sub-nodes. This graph can be viewed as the LGraph equivalent of the CHIRRTL design. Then *firbits* emulates the *bitwidth* pass to propagate and analyze bitwidths for the whole design from the module I/O. After that, the *firmap* pass then exploits the bitwidths information collected in *firbits* to translate from CHIRRTL operators to LGraph cells. Figure 6.4 demonstrates a high-level view of this flow.

6.4 Setup

LiveHD is implemented with C++17 and compiled with GCC 11.0.3. CIRCT still does not have releases, so the top of the tree on Aug 12th, 2022, is used; Yosys uses the latest release (v0.20+22), and the same for Scala-FIRRTL (v1.5.0-RC2) is chosen for evaluations. For CIRCT, we follow the LLVM benchmarking guidelines to use release

and avoid assertions. All compilers are compared without the *dedup* option because LiveHD has not fully implemented it (see Table 6.3). The experiments are run on a server with AMD EPYC 7542 32-Core Processor at 3.4GHz, 504GB memory, and Kali Linux 5.14.0-kali2-amd64 installed. A frequency scaling effect is measured by turning the turbo option on and off the processor. All experiment data are collected using perf profiler and Perfetto [30].

Table 6.3 Compiler flags or commands for fair evaluations

compiler	**flags/commands**
Scala-FIRRTL	*–no-dedup -X verilog*
CIRCT-FIRRTL	*-inject-dut-hierarchy=false -wire-dft=false -prefix-modules=false -inline=false -emit-metadata=false -emit-omir=false -verify-each=false*
Yosys	*read_verilog; proc; write_verilog*

6.5 Evaluation

The evaluation consists of two main parts: multi-threaded speedup scalability and single-threaded performance. Only open-source tools are compared here because the commercial EDA tools license forbids tool benchmarking.

I use a RISC-V Manycore design in CHIRRTL to compare against CIRCT. This RISC-V Manycore design consisting of 128 RISC-V 32bits integer(rv32i) cores. I only compared the scalability with CIRCT-FIRRTL because there is no other parallel Verilog or Pyrope compiler.

To fairly compare across the three different languages, I design a Balanced

Computational Tree (BCT) benchmark. BCT is a large circuit generated with a randomized script. It consists of 1.3 million gates spread over 3309 modules; The design dependency tree has a depth of 7, and each parent module has an average of 4 children. Each module contains an average of 391 mixed xor and summation operators that are chained together.

6.5.1 Multi-Threaded Scalability

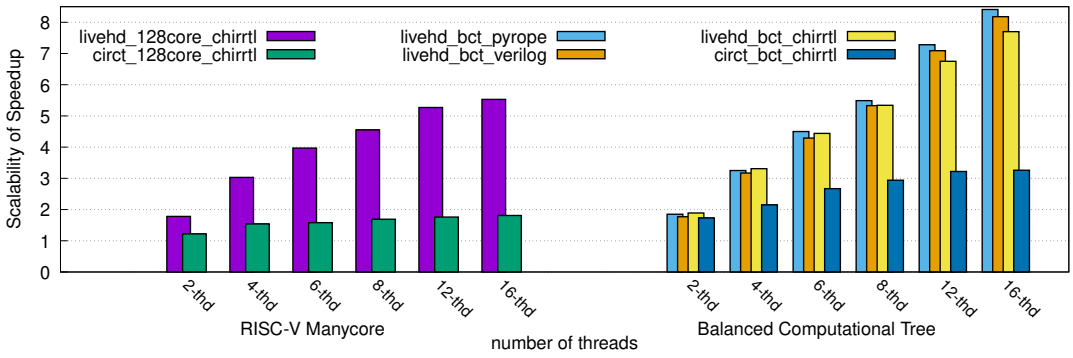


Figure 6.5: LiveHD compiler shows high speedup scalability for a balanced computation tree circuit in Pyrope, Verilog, and CHIRRTL HDLs. LiveHD also scales better for a RISC-V Manycore RISCV processor compared to the Circt-FIRRTL compiler

Figure 6.5 demonstrates the high speedup scalability that LiveHD provides. With 8-thread, LiveHD has 4.55x scalability speedup for the RISC-V Manycore design and 5.34x for the BCT CHIRRTL design. Meanwhile, the CIRCT-FIRRTL compiler only scales 1.7x and 2.9x for the two designs, respectively. When adding more hardware

resources up to 16-threads LiveHD's increasing tendency of scalability slows down but still hits as high as 5.5x speedup for the RISC-V Manycore; The 16-threaded LiveHD also compiles the BCT design and achieves excellent scalabilities of 8.4x in Pyrope, 8.2x in Verilog, and 7.7x in CHIRRTL.

6.5.1.1 Case Analysis: RISC-V Manycore in FIRRTL

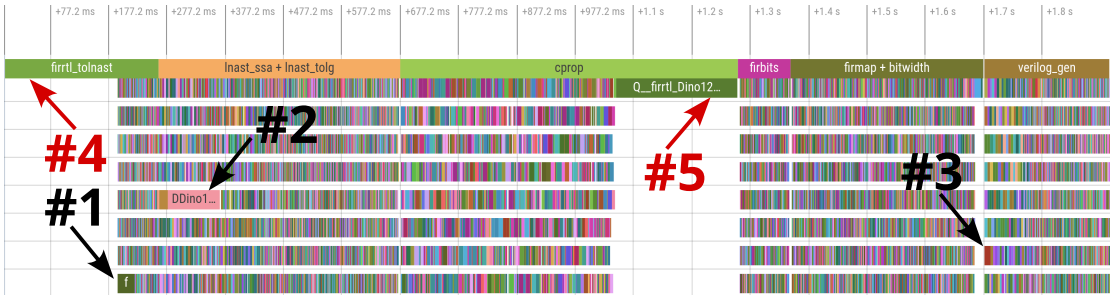


Figure 6.6: LiveHD's parallel schemes establish remarkable thread utilization for an 8-threaded compilation.

Threads Utilization LiveHD gets an overall thread utilization of 76.57% when compiling RISC-V Manycore. This means that under 25% of the CPUs are idle without work. Figure 6.6 presents a visualization of how LiveHD orchestrates the threads. A vertical line means a pass is processing a module. The higher density of colored vertical lines means higher thread utilization. The 8-rows in the figure represent the 8-threads run.

The passes of *lnast_ssa*, *lnast2lg*, and *verilog_gen* deploy full-parallelism, and

thus each module could be compiled in any order. To exploit the max potential of the full-parallelism mechanism, the LNAST and LGraph objects are sorted by their size before piping into these passes. Thus, as arrows #1, #2, and #3 pointed, the most critical path at the beginning of these passes can be hidden.

However, this Perfetto visualization also reveals two facts that detriment the overall scalability. The first facet is the protobuf initialization period as pointed by the arrow#4. LiveHD exploits the Google's protocol buffer package to parse CHIRRTL. At the very beginning of *source_tolnast* pass, LiveHD needs to call a constructor to deserialize and generate the firrtl_protobuf object. This constructor will take 9.2% of the entire execution period.

The second bottleneck is to resolve the top module sub-instances I/O connection issue as pointed by arrow#5. In a FIRRTL design, module I/O usually consists of a deep aggregate data type (listing 6.4), and LiveHD resolves it at the *cprop* pass. Yet, *cprop* deploys bottom-up parallelism, so the top module has to be the last one to be compiled. Moreover, the top module contains 128 instances of RV32i CPUs, and each instance has 28 deep hierarchical I/O connections, which is similar to the example of listing 6.4. The total 3584 deep I/O connections in total introduce a massive hierarchical tuple chain structure in the LGraph IR. Thus, the top module introduces a considerable overhead at the *cprop* pass.

The main reason that LiveHD solves the I/O connection at *cprop* comes from the Pyrope language semantic constraint. Unlike the FIRRTL HDL, where every I/O has been declared explicitly, a Pyrope submodule could infer the tuple I/O field from

the parent module and that needs to be handled at *cprop*.

```
inst  mem of DualPortedCombinMemory
inst imem of ICombinMemPort
mem.io.imem.request.bits.operation <=
    imem.io.bus.request.bits.operation
```

Listing 6.4: CHIRRTL's deep-hierarchical I/O connection adds non-trivial top-module overhead

CHIRRTL's deep-hierarchical I/O connection adds top-module overhead. Once the deep I/Os are resolved by the *cprop* pass, the rest of the bottom-up parallelism passes (*firbits* and *firmap* + *bitwidth*) can benefit from the lowered data structure; Their input workloads are more balanced among each module.

The protobuf initialization and top-module *cprop* (arrow#4 and #5) together take up 18.8% of the time and prevent LiveHD from reaching ideal speed scalability due to Amdahl's law. If excluding these two regions, LiveHD attains high average thread utilization of 97.21%.

Breakdown Analysis Table 6.4 represents the pass breakdown to better understand the source of scalability increment. Besides the *firrtl_tolnast* and the *cprop* discussed in the previous paragraphs, all other passes get excellent utilization of thread resources from 80% to 99%.

Interestingly, these high utilization numbers do not perfectly leads to an ideal speedup. Several reasons like decreased instruction per cycle (IPC) and processor frequency can be observed from the Table 6.4. LiveHD's IPC got affected by mixed reasons like instruction TLB (iTLB) and cache miss rate. For example, at the *firrtl_tolnast* pass, the main slowdown reason in IPC is that the iTLB miss rate of a single thread

Table 6.4 IPC drop and frequency downscaling are two reasons why the high thread utilization from 1 to 8 threaded compilation does not give the ideal speedup.

name of pass	8-threaded		from 1 to 8-threaded			
	fraction	utilization	speedup	#inst.	ipc	freq.
firrtl_tolnast	12.0%	34%	2.5x	+3.3%	-22.2%	-19.0%
lnast_ssa& lnast_tolg	23.8%	97%	4.3x	+0.1%	-21.0%	-13.7%
cprop	30.7%	65%	5.0x	+0.5%	-7.9%	-14.3%
firbits	4.6%	80%	6.8x	+2.8%	-5.2%	-14.7%
firmap & bitwidth	17.6%	95%	5.6x	+0.1%	-14.9%	-13.9%
verilog_gen	11.3%	99%	5.4x	+2.8%	-27.4%	-22.3%
LiveHD: all	100%	76%	4.6x	+1.0%	-15.8%	-15.5%
CIRCT: all	100%	n/a	1.7x	+1.2%	-17.9%	- 1.8%

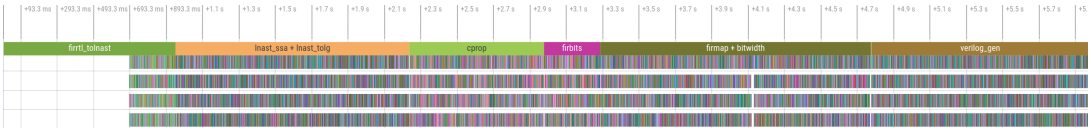
remains the same, but their effects are accumulated in 8-threads and become a burden. At the same time, the cache miss rate increased dramatically in 8-threads, for instance, the *verilog_gen* pass. This is because, in LiveHD, each thread handles different LGraph modules and loses the cache locality from *firmap* and *bitwidth* passes.

LiveHD also has a 15.5% impact from frequency downscaling overall. This is expected due to the turbo (frequency scaling) option enabled on modern CPUs. On the other hand, the CIRCT-FIRRTL compiler only has the parallel speedup of 1.7, and thus not so many threads are used simultaneously during the compilation. Less utilization has less impact on the frequency.

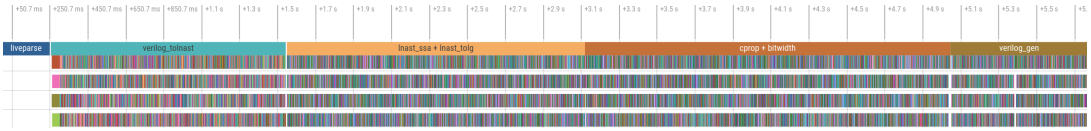
The third reason comes from the implicit mutex contention that is not revealed

to the way that Perfetto counts traces. LiveHD uses a graph_library to manage read-/write module graphs; a mutex protects the overlapped graph writes. Meanwhile, the RISC-V Manycore contains 2945 modules, leading to a high mutex lock activity for the *lnast_tolg* pass because of the high amount of graph creations.

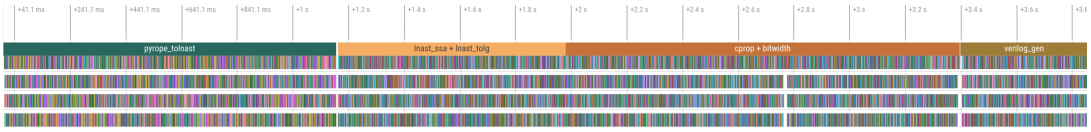
6.5.1.2 Case Analysis: Balanced Computational Tree



(a) 4-threaded LiveHD-FIRRTL completes in 6s, and scales by 3x with 87% of thread utilization



(b) 4-threaded LiveHD-Verilog completes in 5.7s, and scales by 3.2x with 94% of thread utilization



(c) 4-threaded LiveHD-Pyrope parallelizes all passes, completes in 3.9s, and scales by 3.8x with 94% of thread utilization

Figure 6.7: LiveHD exhibits high thread utilization and speedup for all FIRRTL, Verilog, and Pyrope HDLs in the BCT compilation.

The visual traces for the 4-threaded compilation are presented in Figure 6.7. The three sub-graph are displayed with different time scales. While LiveHD-Pyrope

finishes in around 3.9 seconds, the equivalent circuit in LiveHD-FIRRTL requires 6 seconds. Table 6.5 shows the overall speedup, which includes the scalability and single-threaded performance but using the execution times, we can deduce that Verilog and Pyrope have approximately the same scalability between 8.3-8.6x. FIRRTL has 7.7x for a 16-threaded execution. The reason is consistent with the verilog_gen overhead shown in Table 6.4, which has over 20% drop in IPC, over 20% drop in frequency, a small 2.8% instruction count increase, and under 1% lock contention.

Table 6.5 LiveHD provides outstanding compilation speedup in single-threaded and multi-threaded scenarios.

design	**compiler**	**#threads**	**time(s)**	**overall speedup**
RISC-V Manycore-FIRRTL	Scala-FIRRTL	1	27.0	[1]**1.0x**
	CIRCT	1	3.8	7.1x
	LiveHD	1	9.0	3.0x
	CIRCT	16	2.1	12.8x
	LiveHD	16	1.6	16.5x
BCT-FIRRTL	Scala-FIRRTL	1	122.0	[1]**1.0x**
	CIRCT	1	20.6	5.9x
	LiveHD	1	20.2	6.0x
	CIRCT	16	6.3	19.4x
	LiveHD	16	2.6	46.6x
BCT-Verilog	Yosys	1	171.2	[1]**1.0x**
	LiveHD	1	19.9	8.6x
	LiveHD	16	2.4	71.3x
BCT-Pyrope	LiveHD	1	13.7	1.0x
	LiveHD	16	1.6	8.4x

note:[1] the baselines

Source HDLs Parsing Using BCT allows us to compare different languages' scalability and performance. A significant difference happens during parsing. LiveHD-FIRRTL has the non-parallel protobuf serialization previously mentioned. LiveHD calls the Verilog Slang parser in parallel for each file, but it still requires splitting the Verilog file in multiple files. Pyrope parsing is fully parallel.

Balanced Workload of BCT Unlike RISC-V Manycore, BCT does not have a significantly larger top module. The result is a higher balance in the bottom-up passes. Therefore, as shown in Figure 6.7, it leads to better thread utilization because no parent needs to wait for any giant child. Further, Verilog has no aggregate data types, so the flattened BCT I/O is generated in all languages. That means no deep hierarchical I/O as in the List 6.4, and it leads to a faster *cprop* result in all three HDLs.

6.5.2 Single-Threaded Performance

Different compilers' performance are measured in the Table 6.5. The single-threaded LiveHD-FIRRTL achieves good speedups of 3x for RISC-V Manycore and 6x for BCT compared to the baseline Scala-FIRRTL compiler.

Since there is no parallel Verilog compiler in the open-source community, Yosys is chosen as the comparator. For the BCT design, the LiveHD is 8.6x faster than Yosys in single-thread, respectively. LiveHD is the only compiler for Pyrope, but the performance is faster than Verilog and CHISEL for an equivalent circuit.

6.6 Conclusions

Compilation time is a key bottleneck on hardware productivity only exacerbated by new HDLs. These modern HDLs' long compilation adds more burden to the current lengthy hardware design flow.

The main contribution of this chapter is LiveHD, a new multi-HDL, parallelized, and fast compilation framework. The chapter goes over the main challenges of parallelizing all the compiler passes. We pick the FIRRTL, Verilog, and Pyrope HDLs to demonstrate LiveHD's ability of generic compilation.

The presented LiveHD compiler can achieve as high multi-threading scalability as 5.5x for the RISC-V Manycore FIRRTL. We also create BCT as the benchmark that resembles large designs to allow comparing across different HDLs compile times. BCT has between 7.7x to 8.4x speedup scalability for a 16-threaded compilation.

The parallel scalability results are on top of a fast single-threaded LiveHD compiler. Compared to the Scala-FIRRTL compiler, single-threaded LiveHD has over 6 times speedup. Compared to the popular Yosys, single-threaded LiveHD is 8.6 times faster.

Chapter 7

Conclusion and Future Opportunities

Compiling hardware description languages has been a hot topic in the academic and open-source community. Together with the new proposed HDLs, many hardware-specific IR framworks are also created to support compiler development. However, each HDL compiler's isolated ecosystem results in many code duplicates and prevents novelty sharing. A carefully designed generic compilation framework could solve this issue by allowing different HDL developers to leverage it easily. Furthermore, the compilation speed has become the new important metric that requires much more attention as the hardware EDA flow is already lengthy.

In this thesis, I have proposed a fast and parallel open-source framework to be the unified hardware compilation infrastructure for HDLs development. The generic power of the high-level LNAST-IR eases the difficulty of bridging a new HDL into the LiveHD framework. The versatile APIs and traversal ability from the low-level LGraph-IR empowers the developers to implement the algorithm easily. More importantly, the

newly invented parallel LiveHD framework could boost hardware designers' productivity because of the fast compilation throughput.

7.1 Future work

Through the development of the LiveHD, we discover that some design decisions dramatically impact compilation speed. Here I discuss some of the issues and suggest future research directions to improve the LiveHD framework further. Furthermore, I present several exciting research opportunities opened by the new LiveHD framework.

7.1.1 Resolving High-level Program Structural at LNAST

Currently, LiveHD solves the high-level tuple structure at the *cprop* pass in LGraph-IR. However, as noted by Section 6.5.1.1, there is deep hierarchical IO syntax from the CHIRRTL HDL. The low-level netlist properties of LGraph-IR is not the ideal stage to resolve these tuple structures, because LiveHD take a considerable effort to construct another internal data structure to flatten the tuple hierarchy fields at runtime.

We found this problem and launched another research project called *lnast_opt* led by another Ph.D. student. The goal of *lnast_opt* is to resolve all the tuple structures at the LNAST stage. LiveHD can benefit from *lnast_opt* in several aspect. First, the number of *tuple_add* and *tuple_get* operations can be largely reduced in the final LNAST and thus reduce the work of *lnast_tolg* to translate these tuple operations into LGraph

nodes. Secondly, the tuple resolving step in the original *cprop* pass is no longer needed and thus could largely reduce the compilation time.

7.1.2 A Finer-granularity of Parallelization

There are several synchronization barriers in LiveHD as demonstrated in 6.2. Theoretically, if two consecutive passes are all full-parallel, a module could ignore the synchronization barriers and directly execute the next pass once the first pass is finished. This can be achieved by properly re-designing the LiveHD shell, which arranges all the pass executions. Furthermore, if we can successfully develop the *lnast_opt* pass that resolves all of the IO tuples, we can turn the *cprop* pass into a full-parallel pass. With the proposed solution, we anticipate LiveHD could achieve finer-granularity parallelism and get a much better compilation speed.

7.1.3 A Verifiable Pyrope Compilation

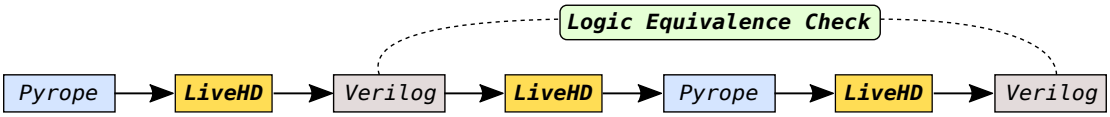


Figure 7.1: A potential verification flow to check the compiler correctness of Pyrope HDL

The ability of LiveHD to take in HDL source code and generate HDL brings out an exciting path to verify the correctness of the Pyrope compiler stack, as shown in Fig-

ure 7.1. The idea is that the Pyrope source code (P1) describes a hardware architecture using high-level syntax, and the generated Verilog (V1) is the low-level representation of the hardware. Though we cannot directly verify the equivalence between P1 and V1, we could make the generated V1 go into LiveHD to get a new Pyrope code (P2), and go one step more to feed P2 to LiveHD to get the final Verilog (V2). Now we can perform a logic equivalent check on V1 and V2. If V1 equals V2, it means the compile process from P1 to V1 also produces consistent result.

7.1.4 An Ene-to-End Parallel and Incremental Hardware Compilation

LiveHD is developed with live programming techniques in mind. Several research papers regarding incremental hardware synthesis [70], placement and routing [71], and simulation [78] have also been proposed in our group. These techniques are complementary to this thesis. In an ideal world, the parallel and multi-HDLs compilation presented in this thesis could provide a fast baseline compilation. The live techniques can further provide an incremental phase whenever the designer makes a small design change. Together with the incremental compilation, the fast baseline compilation that LiveHD provides will boost hardware designers' productivity.

7.2 Concusion

The highly parallelized and generic LiveHD compilation framework opens many exciting opportunities for HDLs and EDA research. A hardware designer could

enjoy LiveHD's high compilation speed to improve productivity. A developer for a new HDL could interface with LiveHD's generic LNAST IR. Similarly, an EDA research project could exploit LiveHD's ability to interface with the Verilog front-end, then use the provided parallelization framework to develop a parallelized EDA tool. We plan to release the LiveHD compiler as open-source to enhance the impact on the community.

Bibliography

[1] Abseil. https://abseil.io/. Online; accessed on 20 April 2022.

[2] Database and Tool Framework for EDA. https://github.com/The-OpenROAD-Project/OpenDB. Online; accessed on 10 April 2020.

[3] magma. https://github.com/phanrahan/magma. Online; accessed on 10 August 2022.

[4] Protocol Buffers - Google's data interchange format. https://github.com/protocolbuffers/protobuf. Online; accessed on 10 August 2021.

[5] Qflow. http://opencircuitdesign.com/qflow/. Online; accessed on 20 April 2022.

[6] robin-map. https://github.com/Tessil/robin-map. Online; accessed on 28 August 2019.

[7] ska-map. https://github.com/skarupke/flat_hash_map. Online; accessed on 28 August 2019.

[8] slang - SystemVerilog Language Services. https://github.com/MikePopoloski/slang. Online; accessed on 5 August 2021.

[9] XLS: Accelerated HW Synthesis. https://github.com/google/xls, 2021. Online; accessed on 9 August 2021.

[10] CIRCT: Circuit IR Compilers and Tools. https://github.com/llvm/circt, 2022. Online; accessed on 12 August 2022.

[11] Guide to Rustc Development. https://rustc-dev-guide.rust-lang.org/, 2022. Online; accessed on 12 August 2022.

[12] Rapid Open Hardware Development (ROHD) Framework. https://github.com/intel/rohd, 2022. Online; accessed on 9 August 2022.

[13] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.

[14] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6. ACM, 2014.

[15] 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 *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.

[16] Scott Beamer and David Donofrio. Efficiently exploiting low activity factors to accelerate rtl simulation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[17] Giuliano Belinassi. The Parallel GCC. `https://gcc.gnu.org/wiki/ParallelGcc`, 2021. Online; accessed on 28 July 2021.

[18] Matheus Tavares Bernardino, Giuliano Belinassi, Paulo Meirelles, Eduardo Martins Guerra, and Alfredo Goldman. Improving parallelism in git and gcc: Strategies, difficulties, and lessons learned. *IEEE Software*, 2020.

[19] Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*, pages 24–40. Springer, 2010.

[20] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D Brown, and Jason H Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):1–27, 2013.

[21] Tung-Chieh Chen, Zhe-Wei Jiang, Tien-Chang Hsu, Hsin-Chen Chen, and Yao-Wen Chang. Ntuplace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1228–1240, 2008.

[22] Chung-Kuan Cheng, Andrew B Kahng, Ilgweon Kang, and Lutong Wang. Replace:

Advancing solution quality and routability validation in global placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[23] Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. *ACM Sigplan Notices*, 30(3):35–49, 1995.

[24] 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.

[25] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C Necula. Dependent types for low-level programming. In *European Symposium on Programming*, pages 520–535. Springer, 2007.

[26] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[27] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[28] 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.

[29] Jan Decaluwe. Myhdl manual, 2019.

[30] Google Developers. Perfetto: System profiling, app tracing and trace analysis. `https://perfetto.dev/`, 2022. Online; accessed on 10 August 2022.

[31] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*, pages 1–10, 2013.

[32] Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth, George Leontiev, Fabian Schuiki, Ram Sunder, et al. Mlir as hardware compiler infrastructure. In *Workshop on Open-Source EDA Technology (WOSET)*, 2021.

[33] Geovane Fedrecheski, Laisa CP Costa, and Marcelo K Zuffo. Elixir programming language evaluation for iot. In *2016 IEEE International Symposium on Consumer Electronics (ISCE)*, pages 105–106. IEEE, 2016.

[34] Guilherme Flach, Mateus Fogaça, Jucemar Monteiro, Marcelo Johann, and Ricardo Reis. Rsyn: An extensible physical synthesis framework. In *Proceedings of the 2017 ACM on International Symposium on Physical Design*, ISPD '17, pages 33–40, New York, NY, USA, 2017. ACM.

[35] Tiago Fontana, Renan Netto, Vinicius Livramento, Chrystian Guth, Sheiny Almeida, Laércio Pilla, and José Luís Güntzel. How game engines can inspire eda tools development: A use case for an open-source physical design library. In

Proceedings of the 2017 ACM on International Symposium on Physical Design, ISPD '17, pages 25–31, New York, NY, USA, 2017. ACM.

[36] Russell Friesenhahn and Johnathan York. Arl: Ut's experiences in the free open-source vlsi eda landscape. Oct. 2018.

[37] Sanjay Ghemawat and Paul Menage. Tcmalloc: Thread-caching malloc, 2009.

[38] Taras Glek and Jan Hubicka. Optimizing real world applications with gcc link time optimization. *arXiv preprint arXiv:1010.2196*, 2010.

[39] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. High-performance cross-language interoperability in a multi-language runtime. In *ACM SIGPLAN Notices*, volume 51, pages 78–90. ACM, 2015.

[40] UCSC MASC Group. LiveHD Code Generation. `https://github.com/masc-ucsc/livehd/tree/master/inou/code_gen`, 2022. Online; accessed on 29 Oct 2022.

[41] UCSC MASC Group. LiveHD: Live Hardware Development. `https://github.com/masc-ucsc/livehd`, 2022. Online; accessed on 30 Oct 2022.

[42] Michael T Heath et al. *Hypercube Multiprocessors 1986*. Siam, 1986.

[43] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin DF Wong. Opentimer v2: A new parallel incremental timing analysis engine. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(4):776–789, 2020.

[44] Tsung-Wei Huang and Martin D. F. Wong. OpenTimer: A high-performance timing analysis tool. In *Computer-Aided Design, Proceedings of the IEEE/ACM International Conference on*, ICCAD'15, pages 895–902, Piscataway, NJ, USA, Nov. 2015. IEEE Press.

[45] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, et al. Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In *Proceedings of the 36th International Conference on Computer-Aided Design*, pages 209–216. IEEE Press, 2017.

[46] James Cherry. OpenSTA. `https://github.com/abk-openroad/OpenSTA`. Online; accessed on 5 September 2019.

[47] Shunning Jiang, Berkin Ilbeyi, and Christopher Batten. Mamba: closing the performance gap in productive hardware development frameworks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[48] Teresa Johnson, Mehdi Amini, and Xinliang David Li. Thinlto: scalable and incremental lto. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 111–121. IEEE, 2017.

[49] Jinwook Jung, Pei-Yu Lee, Yan-Shiun Wu, Nima Karimpour Darav, Iris Hui-Ru Jiang, Victor N Kravets, Laleh Behjat, Yih-Lang Li, and Gi-Joon Nam. Datc rdf: Ro-

bust design flow database. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 872–873. IEEE, 2017.

[50] Andrew B Kahng and Tom Spyrou. The openroad project: Unleashing hardware innovation. In *Proc. GOMAC*, 2021.

[51] Andrew B Kahng, Lutong Wang, and Bangqi Xu. Tritonroute: an initial detailed router for advanced vlsi technologies. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[52] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–311, 2018.

[53] Rainer Koschke, J-F Girard, and Martin Wurthner. An intermediate representation for integrating reverse engineering analyses. In *Proceedings Fifth Working Conference on Reverse Engineering (Cat. No. 98TB100261)*, pages 241–250. IEEE, 1998.

[54] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[55] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Olek-

sandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[56] Z. Lin, M. Kahng, K. M. Sabrin, D. H. P. Chau, H. Lee, and U. Kang. Mmap: Fast billion-scale graph computation on a pc via memory mapping. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 159–164, Oct 2014.

[57] Wen-Hao Liu, Wei-Chun Kao, Yih-Lang Li, and Kai-Yuan Chao. Nctu-gr 2.0: Multithreaded collision-aware global routing with bounded-length maze routing. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 32(5):709–722, 2013.

[58] Derek Lockhart and Christopher Zibrat, Garyd Batten. PyMTL: A unified framework for vertically integrated computer architecture research. In *Microarchitecture, Proceedings of the 47th Annual IEEE/ACM International Symposium on*, MICRO'14, pages 280–292, Washington, DC, USA, Dec. 2014. IEEE Computer Society.

[59] Kingshuk Majumder and Uday Bondhugula. HIR: An MLIR-based Intermediate Representation for Hardware Accelerator Description. *arXiv preprint arXiv:2103.00194*, 2021.

[60] Cristian Mattarei, Makai Mann, Clark Barrett, Ross G Daly, Dillon Huff, and Pat Hanrahan. CoSA: Integrated Verification for Agile Hardware Design. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 2–5. IEEE, 2018.

[61] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction*, pages 213–228. Springer, 2002.

[62] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. Predictable accelerator design with time-sensitive affine types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 393–407, 2020.

[63] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. A compiler infrastructure for accelerator generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 804–817, 2021.

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

[65] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. 2004.

[66] Maksim Panchenko, Rafael Auler, Laith Sakka, and Guilherme Ottoni. Lightning bolt: powerful, fast, and scalable binary optimization. In *Proceedings of the 30th*

ACM SIGPLAN International Conference on Compiler Construction, pages 119–130, 2021.

[67] Scott Beamer Thomas Nijssen Krishna Pandian and Kyle Zhang. Essent: A high-performance rtl simulator.

[68] Rafael T. Possignolo, Elnaz Ebrahimi, Haven Skinner, and Jose Renau. Fluid-Pipelines: 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.

[69] Rafael T. Possignolo, Sheng H. Wang, Haven Skinner, and Jose Renau. LGraph: A multilanguage open-source database. In *Open-Source EDA Technology, Proceedings of the First Workshop on*, WOSET'18, Oct. 2018.

[70] Rafael Trapani Possignolo and Jose Renau. LiveSynth: Towards an interactive synthesis flow. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 74. ACM, 2017.

[71] Rafael Trapani Possignolo and Jose Renau. SMatch: Structural matching for fast resynthesis in fpgas. In *Proceedings of the 56th Annual Design Automation Conference 2019*, page 75. ACM, 2019.

[72] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image

processing dsl. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):1–25, 2017.

[73] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. LLHD: A multi-level intermediate representation for hardware description languages. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–271, 2020.

[74] David Shah, Eddie Hung, Clifford Wolf, Serge Bazanski, Dan Gisselquist, and Miodrag Milanovic. Yosys+ nextpnr: an open source framework from verilog to bitstream for commercial fpgas. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–4. IEEE, 2019.

[75] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. μir-an intermediate representation for transforming and optimizing the microarchitecture of application accelerators. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 940–953, 2019.

[76] Sheng-Hong Wang, Haven Skinner, Sakshi Garg, Hunter Coffman, Kenneth Mayer, Akash Sridhar, Rafael T. Possignolo, and Jose Renau. Pyrope. `http://masc.soe.ucsc.edu/livehd/pyrope/`. Online; accessed on 16 August 2021.

[77] Satnam Singh and Philip James-Roxby. Lava and jbits: From hdl to bitstream in

seconds. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, pages 91–100. IEEE, 2001.

[78] Haven Skinner, Rafael Trapani Possignolo, Sheng-Hong Wang, and Jose Renau. LiveSim: A fast hotreload simulator. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2020.

[79] Wilson Snyder. Verilator: Speedy reference models, direct from rtl. *Presentation to University of Massachusetts Amherst*, 2017.

[80] Mathias Soeken, Heinz Riener, Winston Haaswijk, and Giovanni De Micheli. The EPFL logic synthesis libraries, May. 2018. arXiv:1805.05121.

[81] Richard M Stallman. *Using the gnu compiler collection: a gnu manual for gcc version 4.3. 3*. CreateSpace, 2009.

[82] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bidwidth analysis with application to silicon compilation. *ACM SIGPLAN Notices*, 35(5):108–120, 2000.

[83] 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.

[84] Sheng-Hong Wang, Rafael Trapani Possignolo, Qian Chen, Rohan Ganpati, and Jose Renau. LGraph: a unified data model and api for productive open-source

hardware design. In *Open-Source EDA Technology, Proceedings of the Second Workshop on*, WOSET'19, Nov. 2019.

[85] Sheng-Hong Wang, Akash Sridhar, and Jose Renau. LNAST: a language neutral intermediate representation for hardware description languages. In *Open-Source EDA Technology, Proceedings of the Second Workshop on*, WOSET'19, Nov. 2019.

[86] Rinse Wester, Christiaan Baaij, and Jan Kuper. A two step hardware design method using cλash. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 181–188. IEEE, 2012.

[87] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B Kessler, Oleg Pliss, and Thomas Würthinger. Initialize once, start fast: application initialization at build time. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[88] Titus Winters. Non-atomic refactoring and software sustainability. In *2018 IEEE/ACM 2nd International Workshop on API Usage and Evolution (WAPI)*, pages 2–5. IEEE, 2018.

[89] Clifford Wolf. Yosys Open SYnthesis Suite. `http://www.clifford.at/yosys/`, 2022. Online; accessed on 5 August 2022.