A Multi-threaded Fast Hardware Compiler for HDLs

Sheng-Hong Wang
UC Santa Cruz
Santa Cruz, USA
swang203@ucsc.edu

Hunter James Coffman
UC Santa Cruz
Santa Cruz, USA
hcoffman@ucsc.edu

Kenneth Mayer
UC Santa Cruz
Santa Cruz, USA
krmayer@ucsc.edu

Sakshi Garg
UC Santa Cruz
Santa Cruz, USA
sgarg3@ucsc.edu

Jose Renau
UC Santa Cruz
Santa Cruz, USA
renau@ucsc.edu

Abstract

A set of new Hardware Description Languages (HDLs) are emerging to ease hardware design. HDL compilation time is a major bottleneck in the designer's productivity. Moreover, as the HDLs are developed independently, the possibility to share innovations in compilation technology is limited.

We design and implement LiveHD, a new multi-threaded, fast, and generic compilation framework across many HDLs (FIRRTL, Verilog, and Pyrope). We propose new parallel full and bottom-up passes to handle HDLs. The resulting compiler can parallelize all the compiler steps.

LiveHD can achieve 5.5x scalability speedup when elaborating a CHISEL RISC-V Manycore. It also gets from 7.7x to 8.4x scalability speedup for a benchmark designed in all three HDLs. This is achieved with a fast single-threaded LiveHD baseline with 6x speedup compared to compilers such as Scala-FIRRTL and 8.6x against Yosys on Verilog.

CCS Concepts: • **Hardware → Hardware description languages and compilation**.

Keywords: HDL, Compiler Design, Parallel Compilation

ACM Reference Format:
Sheng-Hong Wang, Hunter James Coffman, Kenneth Mayer, Sakshi Garg, and Jose Renau. 2023. A Multi-threaded Fast Hardware Compiler for HDLs. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction (CC '23), February 25–26, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3578360.3580254

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 [8, 22], PyRTL [13], and Pyrope [41] have gained popularity. Typically, each HDL is bundled with a compiler that converts its high-level code into Verilog output. Verilog is frequently regarded as the assembly code in the 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. This paper aims to accelerate HDLs compilation, the elaboration step in fabrication before synthesis, as well as the main step in the simulation compilation.

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 [5, 17] compiler has some parallelism. Other popular HDL compilers like FIRRTL and Yosys [49] are not parallelized.

This paper proposes a new HDL compilation framework with the following key contributions: (1) We design a fast parallel multi-HDLs compiler where all the compilation steps can be done in parallel; (2) We implement a proof-of-concept compiler, LiveHD. It supports Verilog, Pyrope, and CHIRRTL (the highest form of FIRRTL). The compiler is faster than the existing open-source alternatives.

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. One key problem is that a module can be accessed without any module declaration. It is an equivalent problem as if a non-HDL language allows function calls without requiring an "include" or "import". Languages like Verilog leverage the specified file processing order to solve the lack of declaration problem.

A pre-scan pass would have problems in some HDLs because the input/outputs depend on the called modules. We propose to dynamically resolved the IOs 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 [21].

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

CC '23, February 25–26, 2023, Montréal, QC, Canada

ACM ISBN 979-8-4007-0088-0/23/02…$15.00
https://doi.org/10.1145/3578360.3580254

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 we define as *full parallelism* in this paper.

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 [31], and LLHD [39] 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 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 [49] 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.

2 Related Work

2.1 HDL Compilers and IRs

HDL compilation and hardware IR design have recently been a research hotspot in the open-source community [1, 5, 7, 13, 22, 23, 29–31, 34, 35, 39–41, 45, 49]. The three main compilers/IRs related are Yosys, Scala-FIRRTL, and CIRCT-FIRRTL.

Yosys: Yosys [49] is a framework for register-transfer-level (RTL) synthesis. The main front end takes Verilog-2005 and converts it to the internal RTLIL [49] IR through a Verilog Abstract Syntax Tree (v-AST in Figure 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 without parallel passes.

Scala-FIRRTL: FIRRTL is the IR in the Chisel3 [8]. The first FIRRTL compiler is implemented in Scala [36](Scala-FIRRTL). A front-end Chisel3 compiler produces CHIRRTL as the input for the Scala-FIRRTL compiler (Figure 1). The Scala-FIRRTL compiler is not designed for compiling languages other than Chisel3/FIRRTL. The FIRRTL compiler is sequential without parallel passes.

CIRCT-FIRRTL: CIRCT [5, 17] is a new experimental hardware IR extended from MLIR [28] and LLVM [27] 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 but right now, only the CHIRRTL input exists. The CIRCT-FIRRTL flow (Figure 1) is concurrently developed with LiveHD and leverages the MLIR pass manager to handle parallelism. The pass manager only allows some passes to be parallel after the module interfaces are known. Only passes with full parallelism will be executed parallelly, otherwise, CIRCT will execute the pass sequentially.

Other HDL IRs and compilers: LLHD [39] and CoreIR [31] 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 like CHIRRTL or Pyrope, which only set bitwidth on the modules' I/O. In CoreIR, input HDLs like Halide [38] 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. Those two compilers are sequential without parallel passes.

Several works [4, 11, 26, 30, 33, 34, 40] 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 sources; thus, they lose the opportunity to reuse and optimize Verilog designs.

2.2 Software Programming Language Compilers

Several software frameworks have partially the same design concept as LiveHD in terms of parallelism and multi-languages support.

Parallel compilation: The two widely used C/C++ compilation frameworks, GCC [42] and LLVM [27] rely on a build system such as Makefile to achieve compilation parallelism at the file-level granularity. Several research works have been recently proposed to improve compilation parallelism. The

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

challenges of increasing scalability for Git and GCC applications are discussed in the research work by Bernardino et al. [10]. The parallel GCC project [9] also aims to conduct a multi-threaded compilation on the intra-procedure optimizations in GCC. Researchers in [19, 24] have been working on getting higher parallelism in the link time optimization (LTO) stage. In Lighting Bolt [37], the authors discuss how they design the parallel mechanism to improve the performance of the binary optimization pass.

Elixir [18] 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. LiveHD follows a dependency tree which avoids the need to suspend/wait for the dependency to be resolved within a running pass.

Multi-languages support: GraalVM [16, 48] is a Java virtual machine framework that bridges multiple languages by using Truffle [20] as the front-end IR. They are analogous to the LiveHD compiler and its front-end IR, LNAST [47]. 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 [12] present a graph-based SSA intermediate representation to express optimization elegantly. The Common Intermediate Language (CIL) [32] is used in the .NET system; it is also an IR designed for multiple languages such as C# and Basic.

3 LiveHD Overview

This section provides insights on LiveHD HDL support, internal IRs, and overall organization that is needed to understand how to parallelize the LiveHD compiler in the following section.

3.1 HDL Supported

Verilog is the de-facto language in industry, and Chisel3 is the most widely used modern alternative to Verilog. As such, it became clear early in the design that LiveHD would need to support both. In addition, LiveHD also supports Pyrope, an HDL that is still under development but with features like global inference that affect the compile design options. By supporting the full Verilog 2001 and some SystemVerilog features, the compiler must address many details. By supporting both languages directly, LiveHD can directly interface Verilog and Chisel3 generated code at compile time, performing optimizations across modules.

For Verilog, we use Slang [3] as the parser. Slang can handle most of SystemVerilog, including non-synthesizable constructs like classes. LiveHD only accepts the synthesizable subset. For Chisel3 [8], we handle the CHIRRTL, which is close to FIRRTL but directly generated by Chisel3. The Scala-FIRRTL [22] compiler accepts the same CHIRRTL before the different lowering steps inside FIRRTL. For Pyrope, we implement a custom parser.

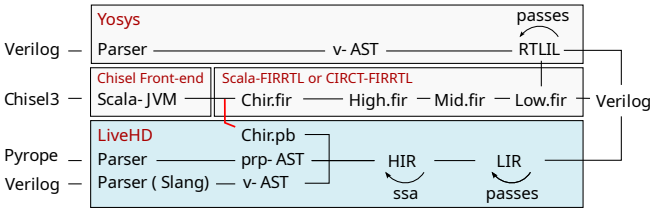


Figure 1. Overview of LiveHD compilation flow and comparisons between Yosys and Chisel3/FIRRTL compilers.

3.2 LiveHD IRs: HIR and LIR

LiveHD is constructed with two internal IRs infrastructures: A High-Level IR (HIR) with a tree-like structure that follows control flow, and a Low-Level IR (LIR) that uses a graph-like structure that resembles a hardware netlist. The 3 different supported languages translate to HIR. There are no major optimizations in HIR, it is used as a common bridge before translating to LIR. The LIR has a graph representation and includes the main time-consuming passes like copy-propagation.

Compared against non-hardware compiler IRs, HIR is something between the HIR and MIR from Rust [6], or closer to the AST than LLVM IR. The HIR resembles the LNAST [47] and the high-level FIRRTL [22]. Internally, HIR has a Static Single Assignment (SSA) [14] pass that can enable its compiler optimization steps. However, in this paper, the primary function of HIR is to remove all control flow structures prior to generating an LIR.

LIR is a bi-directional hypergraph representation closer to a hardware netlist. Each graph node corresponds to a cell like an AND gate, and it can have multiple cell pins as sinks or drivers. LIR resembles Lgraph [46] and Yosys RTLIR [49], but it restricts to have a single driver per cell pin, and many other implementation differences like the number of cells. For each LIR node, there is an equivalent HIR node, but not vice-versa.

LIR has HDL-specific passes like bitwidth inference and code optimization. Although possible to do some of the steps in HIR, LIR has several traversal algorithms like the topological sort that simplify the design.

3.3 LiveHD Overview

Figure 1 shows the high-level overview of the LiveHD compiler. At the front-end, LiveHD currently compiles three HDLs: FIRRTL, Pyrope, and Verilog. Language source codes

are first translated into language-specific parse trees. As an illustration, prp-AST by our own Pyrope parser, v-AST via Slang [3], and Chir.pb via Protocol Buffers [2].

Each language has a custom pass that translates from its internal AST to HIR. Once in HIR, the three languages share the same data structure. By targeting HIR, the language designer does not need to worry about SSA, control flow conversion, variable scopes, and many other constructs shared by most languages. Thus, targeting HIR relieves language designers' efforts.

Unlike MLIR [28], each language does not require explicit nodes. Instead, HIR allows for creating function calls to blackboxed modules as needed. This is used by the FIRRTL pass, which converts a FIRRTL operation to an HIR blackbox function call. This enables per-language custom passes while still allowing each compiler pass to handle the semantics correctly.

LiveHD translates from HIR to LIR. An HIR node can require multiple LIR nodes. LIR employs most of the LiveHD compilation passes, including copy-propagation, constant folding, peephole optimization, and bitwidth inference, to produce the optimized output.

LIR is a hierarchical hypergraph. By hierarchical, we mean that a graph can point to other graphs, and there are many constructs like hierarchical iterators to handle graphs.

4 Parallel Compilation

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

The unit of parallelism of LiveHD is a module, and each file can have several modules. 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 of modules.

A compiler pass is fully parallelizable when a pass can operate on all modules in parallel in any order. 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 tree.

4.1 Dependency Tree

In HDLs, the hierarchy of all module instantiations can always be represented as a dependency tree structure (Figure 2-a, 2-b). In LIR, 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 LIR construction. The DFS traversal only visits these sub-graph nodes without traversing all nodes in the LIR.

In the dependency tree, the leaf module instances must be functionally independent because they have no direct I/O connections. Thus, for a not fully parallelizable pass, LiveHD exploits the ***bottom-up parallelization*** mechanism. It starts by compiling 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 LIR, but they are viewed as different nodes in the dependency tree. In order to prevent redundant compilation on module instances, LiveHD tracks the already optimized LIR 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 to acquire ownership of global objects, thus avoiding data races. LIR IR maintains a graph library to manage basic information like the graph name, graph IOs, and the dependency tree for all the LIRs. This library is a global object that needs mutual exclusion for entry field updates, but the bottom-up traversal guarantees that updates and reads can not happen at the same time.

4.2 Parallelism in Compilation Passes

This section presents LiveHD's compilation stack in Table 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 cluster passes that are bottom-up and fully parallel. The alternative would be to schedule a task for each pass, but it will create smaller tasks with more overhead and lower cache locality due to scheduling the same module in multiple cores.

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 complete, 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.

Full parallelized HIR construction Based on the front-end HDL, LiveHD first decides the functionality of *source2HIR* (see Table 1) and converts an HDL program into HIRs. 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 *source2HIR* threads to handle each module separately.

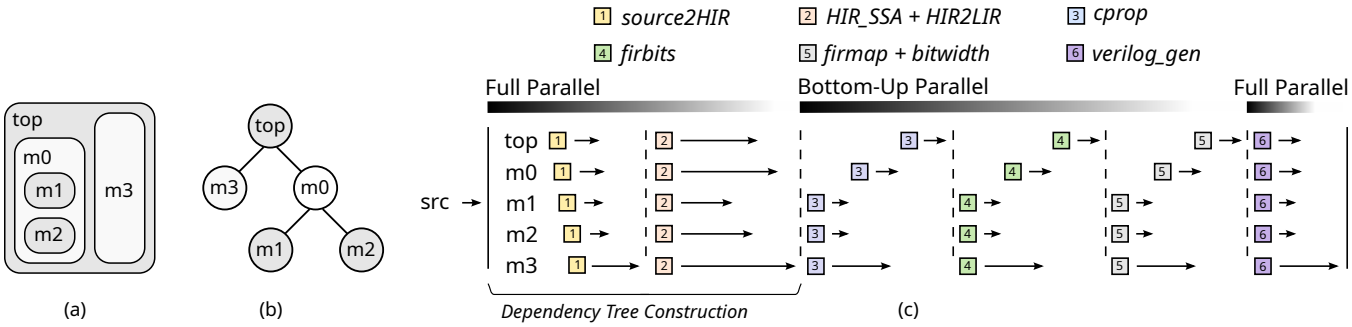


Figure 2. The LiveHD parallel compilation example with a hierarchical FIRRTL front-end. Module *m3* is significantly larger than the others. The vertical dashed lines represent the Synchronization barriers.

Table 1. The LiveHD passes in the compilation order.

Name	Functionality	Parallelization Type	
		Full	*Bottom-up*
source2HIR[1]	Verilog to parse tree to HIR[2]	✓	
	Pyrope to parse tree to HIR	✓	
	CHIRRTL protobuf to HIR[3]	✓	
HIR_SSA	SSA transformation for HIR	✓	
HIR2LIR	HIR to LIR 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 LIR 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

Since the *source2HIR* function merely maps the parse tree of a module into the corresponding HIR, there is no dependency between the executions of the threads. Thus *source2HIR* is a *full parallelizable* pass.

Full parallelized *HIR_SSA* After the HIRs are constructed, LiveHD tasks perform *HIR_SSA* to translate every HIR into SSA form. Although there might be sub-module instantiation statements in the HIRs, 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 independent regarding *HIR_SSA* and can be handled full-parallelly.

Novel uIO Techniques for Fully Parallel IR Lowering In the *HIR2LIR* pass, the functional dependency issue arises when there is a sub-module instantiation in the HDL program. Figure 3 shows that in LIR, 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 *HIR2LIR* is multi-threaded, all HIRs will execute the *HIR2LIR* 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.

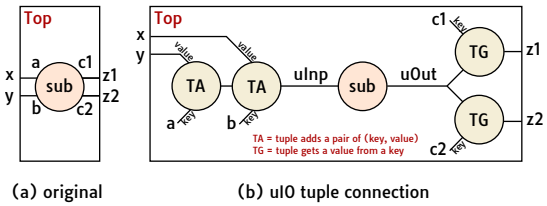


Figure 3. A sub-module instantiation in top-module. The sub-module has inputs (a, b) and outputs (c1, c2). LiveHD aggregates these I/O as tuple uInp/uOut to isolate functional dependency while connecting the top and sub at the *HIR2LIR* pass.

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 LIR tuple structure used to aggregate inputs or outputs, as shown in Figure 3. The uInp and uOut are the only input and output for each module during the *HIR2LIR* step.

As the *HIR2LIR* iterates through the parent HIR, 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 *HIR2LIR* 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. LiveHD combines the passes like a nanopass compiler [25] but the nanopasses are manually integrated. This avoids redundant graph traversal and improves cache locality because all the passes operate over the same CPU not in different tasks. LiveHD currently merges seven functions into a single *cprop* pass as listed in Table 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 LIR 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.

5 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 some of the challenges for each HDL.

5.1 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 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 a 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;
3 endmodule
4
5 module Top(input inp_t, output out_t);
6   Sub sub(.inp(inp_t), .out(out_t));
7 endmodule
```

Listing 1. A Verilog module instantiation example

CHIRRTL In CHIRRTL, no I/O information is provided at the instantiation statement (line 7 of listing 2), so compilers have to figure it out from the left/right-hand sides of subsequent statements (line 8 and 9 of listing 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 2. A CHIRRTL module instantiation example

Interestingly, the Chisel front-end compiler (Figure 1) 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 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 3. A Pyrope module instantiation example

5.2 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 HIR and LIR IR. Both IRs do not require an 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 [43] (*bitwidth* pass in Table 1) to initiate a propagation from the module I/O and calculate the optimized bitwidth for each visited wire.

Customization for implicit HDL specification. Language operators may implicitly present part of the bitwidth specification in an HDL. Table 2 shows such examples in the FIRRTL language.

Table 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 LIR cells. For example, the *head_op* in CHIRRTL can be mapped to the *shift_right_op* in LIR, but the exact shift amount depends on the number of bits. This prerequisite raises an interfacing difficulty for hardware IRs because the entire FIRRTL design bitwidth information must be collected somewhere. It is important to notice that the bitwidth is part of the language specification because performing a different bitwidth can change the generated code semantics.

Because LiveHD is a pass-modularized framework, these challenges can be addressed easily by plugging-in HDL-specific bitwidth inference passes. The CHIRRTL front-end is handled by two CHIRRTL-specific passes, *firbits* and *firmap* (see Table 1), to handle the CHIRRTL front-end. LiveHD first individually translates FIRRTL operators to special LIR sub-nodes. Then *firbits* follows the FIRRTL specification to compute the bits for each node. After that, the *firmap* pass leverages the bit information to translate to LIR cells. Figure 4 demonstrates a high-level view of this flow.

6 Setup

LiveHD is implemented with C++17 and compiled with GCC 11.0.3. CIRCT still does not have releases, so we use the top of the tree on Aug 12th, 2022; 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 and checks. All compilers are compared without the *dedup* option because LiveHD has not fully implemented it (see Table 3). The experiments are run on a server with AMD EPYC 7542 32-Core Processor at 3.4GHz, 504GB memory, and Linux 5.14. We measure the frequency scaling effect by turning the turbo option on and off of the processor. All experiment data are collected using perf profiler and Perfetto [15]. The Verilog output generated by the compilers is equivalent and correct. This is confirmed by doing formal logic equivalence [44, 49] checks between their Verilog outputs.

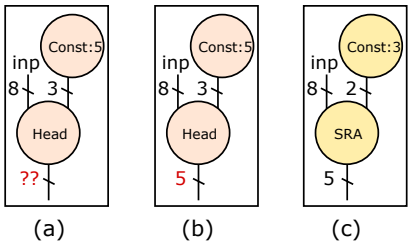


Figure 4. Idea illustration of FIRRTL bits analysis and FIRRTL-LIR mapping passes. The FIRRTL *head_op* extract the MSB 5-bits from the *inp* signal. (a) a FIRRTL-equivalent LIR with an FIRRTL *head_op*. (b) the LIR after *firbits* analysis. (c) mapping to LIR *shift_right_op*.

Table 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*

7 Evaluation

The evaluation consists of two main parts: multi-threaded speedup scalability and single-threaded performance. We compare only against open-source tools because the commercial EDA tools license forbids tool benchmarking.

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

To fairly compare the three different languages, we create 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.

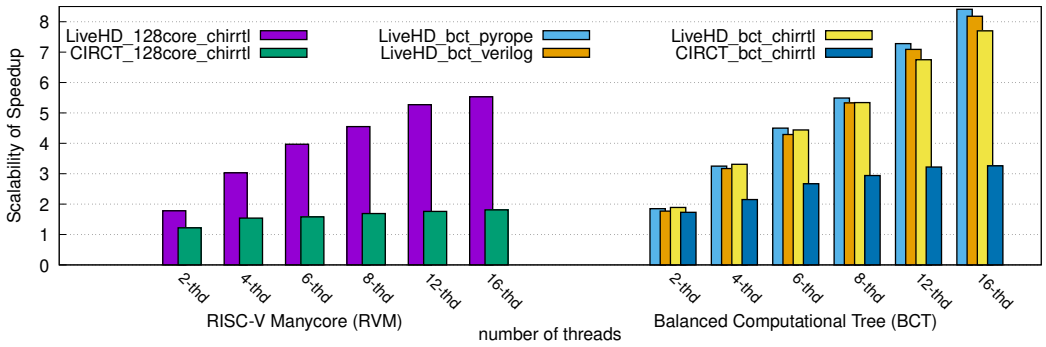


Figure 5. LiveHD compiler shows high speedup scalability for a balanced computation tree circuit in Pyrope, Verilog, and CHIRRTL HDLs. LiveHD also scales better for a 128-core RISCV processor compared to the CIRCT-FIRRTL compiler

7.1 Multi-Threaded Scalability

Figure 5 demonstrates the high speedup scalability that LiveHD provides. With 8-thread, LiveHD has 4.55x scalability speedup for the RVM 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 RVM; 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.

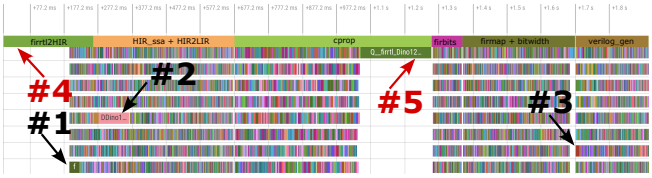


Figure 6. LiveHD's parallel schemes establish remarkable thread utilization for an 8-threaded compilation.

7.1.1 Case Analysis: RVM in FIRRTL. Threads Utilization. LiveHD gets an overall thread utilization of 76.57% when compiling RVM. This means that under 25% of the CPUs are idle without work. Figure 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 *HIR_ssa*, *HIR2LIR*, 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, we sort the HIR and LIR objects by their size before piping into these passes. Thus, as arrows #1, #2, and #3 pointed, we could hide the most critical path at the beginning of these passes.

However, this Perfetto visualization also reveals two facts that detriment the overall scalability. The first facet is the protobuf initialization period as pointed out by the arrow#4. LiveHD exploits Google's protocol buffer package to parse CHIRRTL. At the very beginning of *source2HIR* 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 out by arrow#5. In a FIRRTL design, module I/O usually consists of a deep aggregate data type (listing 4), and LiveHD resolves it at the *cprop* pass. Yet, *cprop* deploys bottom-up parallelism, so the top module must be the last 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 4. The total 3584 deep I/O connections in total introduce a massive hierarchical tuple chain structure in the LIR 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*.

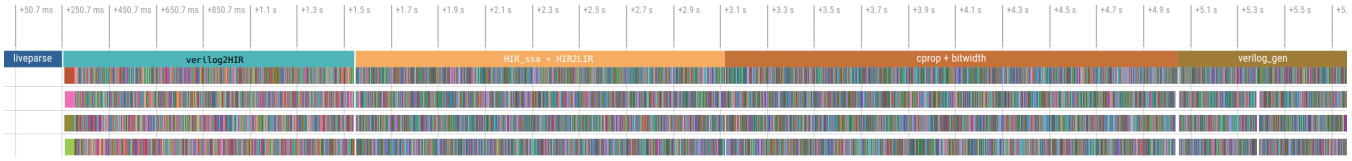
```
1 inst  mem of DualPortedCombinMemory
2 inst imem of ICombinMemPort
3 mem.io.imem.request.bits.operation <=
4     imem.io.bus.request.bits.operation
```

Listing 4. CHIRRTL's deep-hierarchical I/O connection adds non-trivial top-module overhead

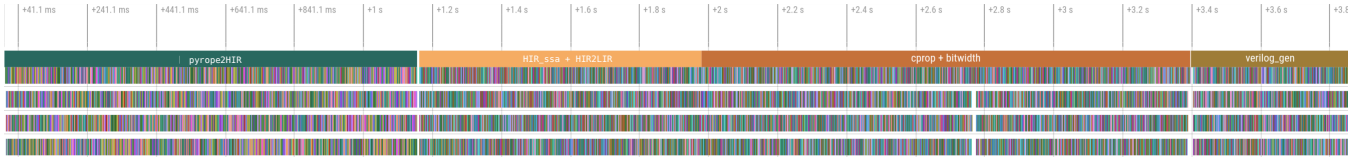
CHIRRTL's deep-hierarchical I/O connection adds top-module overhead. Once the deep I/Os are resolved by the *cprop* pass, the rest of the bottom-up parallelism passes (*firbits* and *firmap + bitwidth*) can benefit from the lowered data structure. Their input workloads are more balanced among each module.

The protobuf initialization and top-module *cprop* (arrow#4 and #5) together take up 18.8% of the time and prevent

(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 7. LiveHD exhibits high thread utilization and speedup for all FIRRTL, Verilog, and Pyrope HDLs in the BCT compilation.

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%.

Table 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.
firrtl2HIR	12.0%	34%	2.5x	+3.3%	-22.2%	-19.0%
HIR_SSA & HIR2LIR	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%

Breakdown Analysis Table 4 represents the pass breakdown to understand the source of scalability increment better. Besides the *firrtl2HIR* 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 lead to an ideal speedup. Several reasons, like decreased instruction per cycle (IPC) and processor frequency, can be observed from the table 4. LiveHD's IPC got affected by mixed reasons like instruction TLB (iTLB) and cache miss rate. For example, at the *firrtl2HIR* pass, the main slowdown reason in IPC is that the iTLB miss rate of a single thread 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 LIR 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 a 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 in the way that Perfetto counts traces. LiveHD protects the graph database for updates with a single-write multiple-read mutex. Meanwhile, the RVM contains 2945 modules, leading to high mutex lock activity for the *HIR2LIR* pass because of the high amount of graph creations. This time is added to the thread utilization and extra instructions. The BCT benchmark allows to see this overhead better, but it is less than 3% in all the tests evaluated.

7.1.2 Case Analysis: Balanced Computational Tree.

We present the visual traces for the 4-threaded compilation in Figure 7. The three sub-graphs 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 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 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 5. LiveHD provides outstanding compilation speedup in single-threaded and multi-threaded scenarios.

design	compiler	[2]#thd	time(s)	overall speedup
RVM-FIRRTL	[1]Scala	1	27.0	[3]**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	[1]Scala	1	122.0	[3]**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	[3]**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] Scala-FIRRTL [2] thread numbers [3] 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 RVM, 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 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 we generated flattened BCT I/O in all languages. That means no deep hierarchical I/O as in the List 4, and it leads to a faster *cprop* result in all three HDLs.

7.2 Single-Threaded Performance

We measure different compilers' performance in the table 5. The single-threaded LiveHD-FIRRTL achieves good speedups of 3x for RVM and 6x for BCT compared to the baseline Scala-FIRRTL compiler.

Since there is no parallel Verilog compiler in the open-source community, we chose Yosys 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.

7.3 Fairness in Comparing the Compiler Result

The comparison between compilers is difficult because the compilers are rarely identical. The LiveHD-FIRRTL and CIRCT compilers have roughly the same compilation passes. The main difference is that CIRCT has common subexpression elimination, whereas LiveHD conducts additional bitwidth optimizations [43] together with bitwidth inference. Aside from that, both compilers have passes of similar goals along with the FIRRTL lowering process, including (1) FIRRTL parsers, (2) IR metadata analysis, (3) IR optimizations, (4) memory structure lowering, and (5) Verilog code generation. Both compilers will infer critical hardware characteristics like clock, reset, memory interface, and bitwidth. Combinational loop detection and black-boxed modules are handled by both compilers. We perform a formal logic equivalence check between outputs to verify the correctness.

When comparing with Yosys on Verilog compilation, we disable all Yosys optimization passes (no opt), and only the Verilog to RTLIR and Verilog generation are executed.

8 Conclusions

Compilation time is a key bottleneck in hardware productivity only exacerbated by new HDLs. We propose LiveHD, a fast multi-HDL compiler. The main novelty of this paper is the HDL parallel compiler. The paper 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 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.

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 to improve productivity. A developer for a new HDL could interface with LiveHD's generic HIR. 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.

Acknowledgments

We would like to thank the reviewers for their feedback on the paper. This work was supported in part by grants from Google and CROSS. This material is based upon work supported by, or in part by, the Army Research Laboratory and the Army Research Office under contract/grant W911NF1910466.

References

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

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

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

[4] 2021. XLS: Accelerated HW Synthesis. https://github.com/google/xls. Online; accessed on 9 August 2021.

[5] 2022. CIRCT: Circuit IR Compilers and Tools. https://github.com/llvm/circt. Online; accessed on 12 August 2022.

[6] 2022. Guide to Rustc Development. https://rustc-dev-guide.rust-lang.org/. Online; accessed on 12 August 2022.

[7] 2022. Rapid Open Hardware Development (ROHD) Framework. https://github.com/intel/rohd. Online; accessed on 9 August 2022.

[8] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*. IEEE, 1212–1221.

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

[10] Matheus Tavares Bernardino, Giuliano Belinassi, Paulo Meirelles, Eduardo Martins Guerra, and Alfredo Goldman. 2020. Improving Parallelism in Git and GCC: Strategies, Difficulties, and Lessons Learned. *IEEE Software* (2020).

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

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

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

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

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

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

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

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

[19] Taras Glek and Jan Hubicka. 2010. Optimizing real world applications with GCC link time optimization. *arXiv preprint arXiv:1010.2196* (2010).

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

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

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

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

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

[25] Andrew W Keep and R Kent Dybvig. 2013. A nanopass framework for commercial compiler development. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. 343–350.

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

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

[28] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.

[29] Derek Lockhart and Christopher Zibrat, Garyd Batten. 2014. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. In *Microarchitecture, Proceedings of the 47th Annual IEEE/ACM International Symposium on (MICRO'14)*. IEEE Computer Society, Washington, DC, USA, 280–292. https://doi.org/10.1109/MICRO.2014.50

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

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

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

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

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

[35] Rishiyur Nikhil. 2004. 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.* IEEE, 69–70.

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

[37] Maksim Panchenko, Rafael Auler, Laith Sakka, and Guilherme Ottoni. 2021. Lightning bolt: powerful, fast, and scalable binary optimization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. 119–130.

[38] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2017. Programming heterogeneous systems from an image processing DSL. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 3 (2017), 1–25.

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

[40] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. 2019. 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*. 940–953.

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

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

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

[44] Synopsys Inc. [n. d.]. Formality User Guide.

[45] Jose I. Villar, Jorge Juan, Manual J. Bellido, Julian Viejo, David Guerrero, and J. Decaluwe. 2011. Python as a hardware description language: A case study. In *2011 VII Southern Conference on Programmable Logic (SPL)*. 117–122. https://doi.org/10.1109/SPL.2011.5782635

[46] Sheng-Hong Wang, Rafael Trapani Possignolo, Qian Chen, Rohan Ganpati, and Jose Renau. 2019. 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)*.

[47] Sheng-Hong Wang, Akash Sridhar, and Jose Renau. 2019. LNAST: A Language Neutral Intermediate Representation for Hardware Description Languages. In *Open-Source EDA Technology, Proceedings of the Second Workshop on (WOSET'19)*.

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

[49] Clifford Wolf. 2022. Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/. Online; accessed on 5 August 2022.

Received 2022-11-10; accepted 2022-12-19