A Parallel HDL Compilation Framework

Sheng-Hong Wang*, Sakshi Garg†, Hunter James Coffman‡, Kenneth Mayer§, Jose Renau¶
Dept. of Computer Science and Engineering
University of California, Santa Cruz
Email: {*swang203, †sgarg3, ‡hcoffman, §krmayer, ¶renau}@ucsc.edu

***Abstract*—We present LiveHD, a parallel HDL compiler to boost the HDL compilation throughput. The hierarchical dependency of the design is resolved internally in LiveHD without any pre-scan step on source files. We identify why and how LiveHD pass is compiled with full parallelism or a bottom-up scheme by referencing the design dependency relation. Our results show that when compiling the highest level of FIRRTL language, CHIRRTL, LiveHD is 1.7x faster than the FIRRTL compiler with single-threaded; and 3.1 to 5.48X faster when exploiting 2-5 threads.**

I. Introduction

Modern HDLs like Chisel3/FIRRTL [1], [2], PyRTL [3], and Pyrope [4] have attracted extensive interest because their higher level of expressiveness eases the difficulty of depicting hardware.

However, when programming in modern HDLs, the compilation throughput becomes a critical criterion. Conventionally, HDL compilation is classified as the *elaboration phase* in EDA flows. When the input language is Verilog, the elaboration time takes only a tiny portion compared to the logical and physical synthesis flow [5]. Notwithstanding, it is not the case in modern HDLs where a deep compiler stack is needed to handle the complex semantics, leading to a low compilation throughput.

Modern SoC usually contains thousands of module instantiations. Designing a parallel and scalable HDL compiler is the key to increase compilation throughput. However, heretofore, there is no HDL compiler designed to focus on high parallelization and scalability.

This paper proposes a new concept-proof framework, LiveHD, as the first HDL compiler that achieves high thread scalability. We chose FIRRTL as the target HDL to evaluate how LiveHD addresses the issue of low compilation throughput.

HDLs lack the *#include directive* and preprocessor mechanism as in C/C++. Therefore, resolving both dependency relations and interfaces of the design hierarchy is a challenge for HDL parallel compilation. In the LiveHD, no extra pre-compilation scan step is needed. The design relations are dynamically resolved by constructing a dependency tree during the internal IR generation phase.

Dependency tree guides LiveHD to apply parallelism. For the passes where the caller and callee are independent, LiveHD can compile them parallelly in any order; this is usually called embarrassingly parallel [6], or ***full parallelism*** in this paper. For the passes that are not full parallelism, LiveHD references the dependency tree to select independent modules and compile them parallelly. The selection starts from the dependency tree leaves; A parent module can only be a candidate after all its children have been processed. We define this approach as ***bottom-up parallelism***. One of the major effort in LiveHD is to make as many full-parallelized passes as possible.

Our results show that when compiling the highest level of FIRRTL language, CHIRRTL, LiveHD is 1.7x faster than the FIRRTL compiler in the single-threaded compilation; and 3.1-5.48X faster when exploiting 2-5 threads.

II. Related HDL Compilers and IRs

Recently, HDL compilation and hardware IR designs [2], [7]–[10], [10]–[19] have been a research hotspot for both HLS and RTL abstractions. Nonetheless, most of these works do not apply parallelized compilation, and the compilation throughput is always not the priority.

a) Scala-FIRRTL: FIRRTL is an IR used internally in the Chisel3 [1]. The first FIRRTL compiler is implemented in Scala [20] (Scala-FIRRTL). As shown in Figure 1, a front-end Chisel3 compiler produces CHIRRTL as the input for the Scala-FIRRTL compiler. 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 in order 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.

b) Circt-FIRRTL: Circt [9] is a new experimental hardware IR extended from MLIR [21] and LLVM [22] 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 design in Circt. Right now, the Circt-FIRRTL flow (Figure 1) is being actively developed , and only the FIRRTL input is supported.

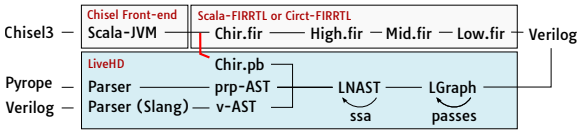

Fig. 1. Overview of LiveHD compilation flow and comparisons between Chisel3/FIRRTL compiler.

III. High-Level Overview of LiveHD

In LiveHD, we focus on handling the RTL abstraction. Figure 1 demonstrates the high-level overview of LiveHD. It uses two levels of IRs, a tree-like language-neutral AST, LNAST [23], and a hierarchical graph, LGraph [24]. LiveHD currently compiles three HDLs: FIRRTL, Pyrope, and Verilog. Source codes are first translated into language-specific parse trees and then transformed to LNAST IR for executing the SSA transformation. The LNAST IR will be lowered into LGraph IR. In LGraph, most of the LiveHD compilation passes are executed to generate the optimized Verilog output.

IV. Parallel Compilation

A. Functional independence w.r.t Parallelism

Functional independence is the key factor to evaluate the parallel ability of an HDL compilation pass. Modules are functionally independent when the pass executions of the modules do not interfere with one another. In most HDLs, a callee's inputs and attributes are usually self-defined and do not depend on the caller's information. That means as long as processing the caller does not rely on the outputs of its callees, the independence property is still valid for a caller-callees pair.

A pass is fully parallelizable when all the modules are functional independent; The pass can operate all modules parallelly in any order. Higher parallel scalability could be achieved as long as the compiler puts more threads resources. Yet, not all passes could achieve this optimal parallelism. Caller-callee modules functionally dependent in a pass have to follow a dependency order and cannot be compiled parallelly. The compiler has to explore the dependency relations and select independent modules to process parallelly to extract further parallelization levels from this kind of passes.

B. Dependency Tree Design

In HDLs, the hierarchy of all module instantiations can always be represented as a dependency tree structure (Figure 2-a, 2 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 employs a depth-first search (DFS) algorithm from the specified top module to recursively traverse into the sub-graph nodes and build the dependency tree. The sub-graph nodes have been recorded previously during LGraph construction. The DFS algorithm could just visit these sub-graph nodes without traversing all nodes in an LGraph.

In the dependency tree, the leaf module instances must be functionally independent of each other because they have no direct IO connections. Thus, LiveHD could exploit the ***bottom-up-parallelization*** mechanism for a non-fully-parallelizable pass: start parallelizing the tree leaves, and then create a new thread job for the parent module once all of its children have been processed.

A module may have multiple instantiations in an HDL program. 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. LiveHD excludes the redundant compilation on module instances by avoiding them entering the internal thread pool.

When accessing global objects, handing over the mutex lock ownership between threads is costly in parallel programming. The execution flow will be bottlenecked sequentially by the critical sections of

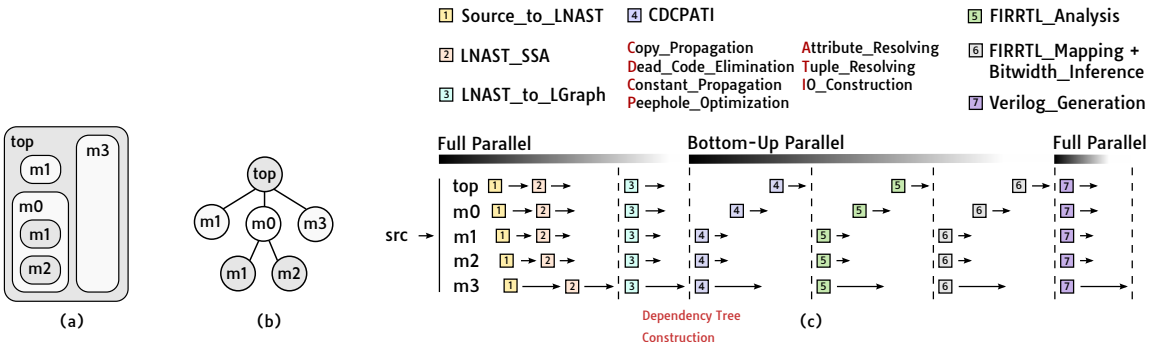

Fig. 2. The LiveHD parallel compilation example with a hierarchical FIRRTL front-end. Module-1 has multiple instances but only gets compiled once. Module-3 has much more lines of code than others. The vertical dotted lines are the synchronization barriers.

each thread. Nonetheless, using LGraph IR minimizes such overhead in LiveHD. LGraph IR maintains a graph library to manage basic information like the graph name, graph IOs, and the dependency tree for all LGraphs. This library is the only global object that needs lock protection. Whenever there is a request to create a new LGraph, the graph library will atomically generate a new graph skeleton. The skeleton generation is just a minor C++ constructor procedure and could be neglected compared to an entire compiler pass.

C. Parallelism of Compilation Passes

1) Fully parallelized LNAST construction: If an HDL source file has multiple modules defined, LiveHD will spawn new threads of *Source_to_LNAST* to handle each of the modules. Since *Source_to_LNAST* 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_to_LNAST* is a fully parallelizable pass.

Whenever a new LNAST is constructed, LiveHD will immediately spawn an *LNAST_SSA* thread task to translate this LNAST into SSA form. Even though there might be sub-module instantiation statements in the LNASTs, since SSA transformation only focuses on the sub-modules return value and inputs arguments, the internal content of the sub-module does not affect the parent module's SSA. Therefore, modules in the dependency tree are independent as regards *LNAST_SSA* and can be handled full parallelly.

2) Fully parallelized LNAST_to_LGraph: In the *LNAST_to_LGraph* pass, the functional dependency issue arises when there is a sub-module instantiation in the HDL program. In LGraph, a sub-module is represented as a sub-node with inputs and outputs connected inside the parent module graph as shown in Figure 3-(a). From the parent point of view, connecting an edge to the corresponding sub-node input requires the knowledge of all sub-module IO. However, suppose that the *LNAST_to_LGraph* is multi-threaded, and all LNASTs start the pass execution with a random order. In such a scenario, the graph library cannot guarantee that the sub-modules IO information is ready for the parent when needed.

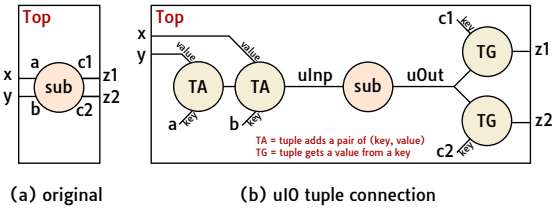

Fig. 3. A sub-module instantiation in top-module. The sub-module has inputs (a, b) and outputs (c1, c2). LiveHD aggregates these IOs as tuple uInp/uOut to isolate functional dependency while connecting the top and sub at the *LNAST_to_LGraph* pass.

LiveHD addresses 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 3. The uInp and uOut serve as

the single input and output per module during the *LNAST_to_LGraph* step.

As the *LNAST_to_LGraph* 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 a uIO atomically. After that, regarding edges connected toward the sub-module node, the parent first creates tuple-add (TA) operators, collects all the edge driver pins as the tuple fields, and connects 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 fetch field from sub-module uOut tuple. LiveHD creates these uIO tuples around the sub-module to break the dependency between parent and child graphs. The uIO resolving process is deferred until the *CDCPATI* pass, where all the program tuples are handled together in a single graph traversal. In this way, the *LNAST_to_LGraph* pass becomes fully parallelizable.

3) Merged bottom-up parallelized passes: LiveHD implements four classical compiler optimizations on the day of writing: *Copy_Propagation, Dead_Code_Elimination, Constant_Propagation*, and *Peephole_Optimization*(*CDCP*). For each optimization, the algorithm starts from the module inputs to traverse the graph locally. Theoretically, the four passes in *CDCP* could be fully parallelized.

However, graph traversal is a time-consuming action. If LiveHD performs the passes of *CDCP* fully parallelized, four individual graph-traversals will be required. We seek the opportunities to combine 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. Thus the graph traversal is minimized to just one iteration. LiveHD currently merges seven functions into the *CDCPATI* passes as shown in Figure 2-c.

Attribute_Resolving, Tuple_Resolving, and *IO_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 connections around the sub-module instance node need to be resolved by flattening the uIO tuple. Then the parent module could continue the rest of the algorithm propagation.

4) Other bottom-up parallelized passes: There are other three bottom-up parallelized passes, *FIRRTL_Analysis*, *FIRRTL_Mapping* and *Bitwidth_Inference*. The algorithm in these passes requires the sub-module outputs attribute to be ready when the parent graph traversal visits them.

5) Verilog generation: This is the final stage of the LiveHD compilation pipeline. Since the functional dependencies between hierarchical modules have been resolved from the previous LiveHD passes, LiveHD can run *Verilog_Generation* with full parallelization.

V. Evaluation

The experiments were run on an AMD EPYC 7542 CPU @ 2.9 GHz with 512 GB of memory, Kali3-AMD64 5.4.0 OS, and compiled with GCC v10.2.1.

We compared the compilation speed on a synthetic hierarchical design between Scala-FIRRTL and LiveHD compilers. The design has a total of 130k lines of code (LoC) and 317 modules. The sizes of modules are uniformly random distributed from 1 to 800 with an average of 400. The design hierarchy could be modeled as a balanced 4-ary dependency tree with a height of 4.

Figure 4 shows that even when running with single-threaded, LiveHD is 1.71x faster than Scala-FIRRTL. When exploits 2 to 4 threads, the speed-up is 3.1x to 4.99x and shows good scalability. When applying 5 threads, since the synthetic design hierarchy is a 4-ary tree and the bottom-up parallelism constrains the parents to wait for their children, the speed-up gaining trend is slow-down to 5.48x.

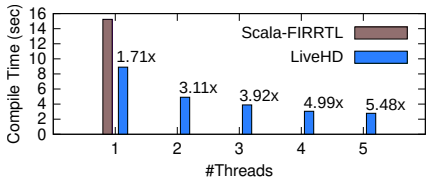

Fig. 4. Multi-threaded LiveHD v.s. Single-threaded Scala-FIRRTL compiler

VI. Conclusion

This paper identifies the fundamental challenge and solutions for designing a parallel HDL compiler. The proposed LiveHD is the first parallelized

HDL compilation framework and could compile FIRRTL HDL with up to 5.48x speed-up compared to the original FIRRTL compiler.

ACKNOWLEDGMENTS

This material is based upon work supported by, or in part by, the Research in Open Source Software (CROSS) at UC Santa Cruz, Google, and Army Research Laboratory and the Army Research Office under contract/grant W911NF1910466.

REFERENCES

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*. IEEE, 2012, pp. 1212–1221.

[2] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. 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*. IEEE Press, 2017, pp. 209–216.

[3] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood, "A pythonic approach for rapid hardware prototyping and instrumentation," in *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE, 2017, pp. 1–7.

[4] 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.

[5] S.-H. Wang, R. T. Possignolo, H. B. Skinner, and J. Renau, "LiveHD: A Productive Live Hardware Development Flow," *IEEE Micro*, vol. 40, no. 4, pp. 67–75, 2020.

[6] M. T. Heath *et al.*, *Hypercube Multiprocessors 1986*. Siam, 1986.

[7] F. Schuiki, A. Kurth, T. Grosser, and L. 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*, 2020, pp. 258–271.

[8] C. Mattarei, M. Mann, C. Barrett, R. G. Daly, D. Huff, and P. Hanrahan, "Cosa: Integrated verification for agile hardware design," in *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2018, pp. 2–5.

[9] "CIRCT: Circuit IR Compilers and Tools," https://github.com/llvm/circt, 2021, Online; accessed on 9 August 2021.

[10] C. Wolf, "Yosys Open SYnthesis Suite," http://www.clifford.at/yosys/, 2021, Online; accessed on 10 February 2021.

[11] K. Majumder and U. Bondhugula, "Hir: An mlir-based intermediate representation for hardware accelerator description," *arXiv preprint arXiv:2103.00194*, 2021.

[12] A. Sharifian, R. Hojabr, N. Rahimi, S. Liu, A. Guha, T. Nowatzki, and A. 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*, 2019, pp. 940–953.

[13] R. Nigam, S. Thomas, Z. Li, and A. Sampson, "A compiler infrastructure for accelerator generators," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 804–817.

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

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

[16] R. 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.* IEEE, 2004, pp. 69–70.

[17] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. 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*, 2018, pp. 296–311.

[18] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming heterogeneous systems from an image processing dsl," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 3, pp. 1–25, 2017.

[19] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, pp. 1–27, 2013.

[20] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the scala programming language," 2004.

[21] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 2–14.

[22] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.

[23] S.-H. Wang, A. Sridhar, and J. Renau, "LNAST: a language neutral intermediate representation for hardware description languages," in *Open-Source EDA Technology, Proceedings of the Second Workshop on*, ser. WOSET'19, Nov. 2019.

[24] S.-H. Wang, R. T. Possignolo, Q. Chen, R. Ganpati, and J. 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*, ser. WOSET'19, Nov. 2019.