Compile-Time RTL Interpreters

Sahand Kashani
EPFL
Switzerland

James R. Larus
EPFL
Switzerland

ABSTRACT

There is no doubt that high-level synthesis (HLS) tools have greatly eased accelerator development. However, many accelerators have already been written in a RTL language and cannot benefit from the automated techniques commonly used by HLS tools. Pipelining is one such technique and must painstakingly be performed by a human when a RTL accelerator needs to be migrated to a higher-speed FPGA for example. This paper provides a first step to bringing HLS-like automated pipelining directly to accelerators written in a RTL language.

1 INTRODUCTION

Pipelining is the main mechanism for creating high-frequency accelerators. All designs—written directly in a RTL language or generated through HLS—exhibit some form of pipelined structure to achieve their performance goals. HLS tools, in particular, are especially good at generating highly-pipelined accelerators and are the subject of most research in techniques for accelerator design today [1, 2, 4]. However, HLS only applies to *new* code and does not help improve the design of *existing* accelerators written in RTL languages. These pre-existing RTL accelerators must currently be modified manually if increased performance is required.

To say RTL design is tedious is an understatement:

(1) RTL languages are verbose and even small design changes that look simple on paper may require modifying hundreds of lines of code [5].
(2) RTL design requires hardware expertise to handle concerns such as explicit clocking, concurrent execution, scheduling, and timing issues (to name a few).
(3) The RTL design process is based on a compile-edit cycle that involves reading textual reports generated by a CAD tool to identify the performance-limiting region of a circuit and act on the design accordingly.

Manually pipelining a RTL design is therefore long and error-prone. This work attempts to answer the question of how one can automatically pipeline existing RTL designs by eliminating the human-in-the-loop in point 3 to help open up a path for future CAD optimizations beyond what is currently possible.

2 LIMITATIONS OF THE STATE OF THE ART

Pipelining has been studied extensively and can in theory be performed automatically, but CAD tools cannot do this in user RTL without annotations. The reason is that RTL languages explicitly encode the precise timing and physical structure of a circuit. Compilers must preserve the semantics of their input programs, but pipelining changes the input-to-output latency of a circuit and thus its original timing.

Retiming is a transparent transformation that decreases a circuit's clock period by moving existing registers around to balance delays. Current FPGA toolchains perform retiming under the hood, but its applicability highly depends on the structure of the input circuit. For example, Intel's HyperFlex FPGA architecture has first-class support for retiming by including bypassable registers on every routing segment of the device to help balance long routing paths. However, these special registers do not support asynchronous resets, clock enables, or initial values. Design registers which use such features can therefore not be retimed to a routing segment register to decrease the clock period.

Intel's CAD tool tries to help designers pipeline their designs to reach performance goals by using a post-route analysis wizard to virtually modify performance-limiting paths, measure the incremental benefit of the modification, and report suggestions back to the designer. However, the timing-driven nature of such analyses is only concerned with design performance, not functionality. The suggestions guide designers to regions of the RTL design that require more pipelining, but do not point to the paths that will need more registers to maintain data synchronization and functional correctness. Similarly, no timing-driven analysis can tell a designer if *less* pipelining is enough to achieve their clock period targets, for example.

3 KEY INSIGHT

If we truly want to eliminate the human-in-the-loop, CAD tools need to know the properties of communication interfaces to determine boundaries at which they are free to manipulate circuits. Fortunately hardware accelerators are not arbitrary circuits and often interact with other components through *elastic* interfaces. In fact, FPGA toolchains automatically perform datapath pipelining of an elastic interface when they are in charge of generating it, but not within user code behind the same interface to preserve the RTL's precise timing semantics (c.f. Figure 1).

A special case exists when a all a design's interfaces are elastic as the whole design becomes latency-insensitive (though it may internally have structures where relative timing matters). The key insight is that accelerators often satisfy this property and their behavior thus only depends on the order of events that occur at their interfaces, and not on the precise timing at which they arrive. Such accelerators can be represented by an algorithm and no longer require the cycle-level modeling of a RTL language to be described. We propose leveraging this fact to lift the gate-level representation of user RTL to a higher-level that captures the essence of what the RTL code is trying to do without modeling the precise timing between which different functional units receive their data. This

LATTE '21, April 15, 2021, Virtual, Earth

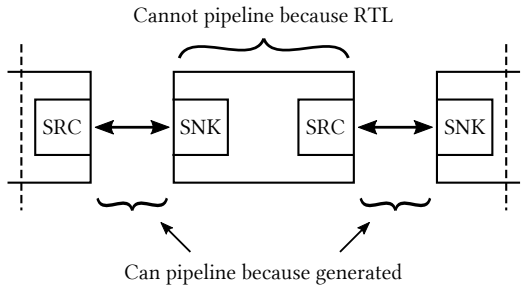

Figure 1: CAD tools can pipeline across elastic interfaces they generate, not within user RTL with the same interfaces.

higher-level representation can then be used to reason about how to restructure, pipeline, and schedule the circuit to achieve better performance.

4 OUR APPROACH

Our goal is to pipeline a RTL circuit towards a given target clock period (or to the best extent possible if the requested clock period is infeasible). Note that this is a work-in-progress and we limit ourselves to processing pipelined DAG circuits for now.

Figure 2 outlines our CAD flow. Our tool builds on top of FIRRTL [3]—an IR and compiler infrastructure for hardware description languages—as its IR supports all operations and abstractions in classical RTL languages and can be extended with custom IR nodes to further ease programmatic manipulation of RTL designs. FIRRTL was originally designed as a backend for the Chisel HDL, but traditional HDLs can be ingested following a pre-processing step through Yosys [6] and custom tools to convert them to FIRRTL.

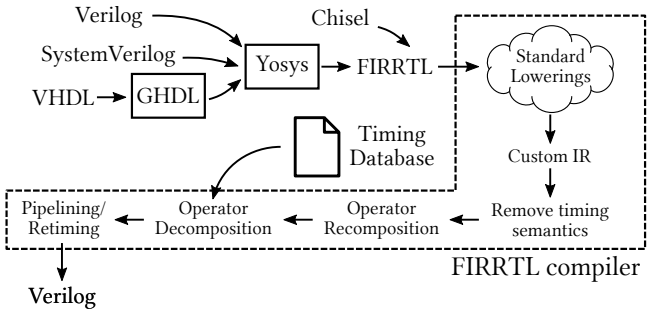

Figure 2: CAD flow.

Pipelining requires knowledge of the latencies introduced by various operations in a circuit. We capture these delays in a timing database which we create by enumerating all of FIRRTL's arithmetic/logic IR operators at various data widths, emitting verilog for each, and running P&R/STA through Quartus. FIRRTL's arithmetic/logic IR operators match those of existing RTLs and their delays are representative. Note that we use post-fit delays instead of post-map delays when characterizing operators as inter-LUT routing makes up a large part of the overall operator latency and these delays are absent in post-map reports. We assume all wiring-related IR operators have a delay of zero as such operators do not need LUTs to be implemented and can be handled directly by the FPGA's routing fabric.

Input RTL circuits may already be sufficiently pipelined to achieve the target clock period. However, new FPGAs may not need as much pipelining for the same results and it may be possible to reduce the latency of an accelerator by removing excessive pipeline stages. We therefore start by dropping all registers in the circuit so as to reason on the functionality of the circuit rather than on its sequential behavior. This transformation is safe as pipeline stages in a DAG only have connections with the stages that immediately precede/follow it and the absence of loops means registers' initial values do not affect successive invocations of the accelerator. A sea of operators remains after this step.

At high clock speeds the delay of IR operators may already exceed the target clock period. High-frequency pipelining therefore requires reasoning on LUT-level circuit representations to allow for more fine-grained register placement. Due to the way Quartus places LUTs when characterizing them in isolation, the delay of a wide operator is typically *smaller* than the sum of narrower operators left behind by the original RTL circuit's pipeline. In order to have accurate timings for an operation, it is important to perform operator recomposition to lift clusters of gates into wider operators. We can then use the timing database to decide how to better decompose operators at a fine granularity suitable for pipelining.

We use retiming as the mechanism for automatic pipelining throughout the compiler. Retiming is a global optimization algorithm and can be expensive on large circuits, so we avoid running it on full circuit graphs until the exact number of registers needed to satisfy the target clock period is determined. This number is found efficiently through the critical path of the design on which a binary search is performed to add registers at the end of the graph and retime them into the circuit. The last successful retiming determines the minimum number of registers needed to satisfy the clock period. Retiming is economical in runtime here as the subset of vertices formed by the critical path represents a short line graph. Finally, we add the required registers to the end of the full circuit graph and perform a single global retiming call to pipeline the circuit.

Note that eliminating registers and reintroducing them, as performed by our compiler, removes structural limitations on these registers that may have prevented effective retiming in the original RTL circuit. Retiming in the compiler can therefore go deeper than that possible if the original RTL circuit's semantics were kept as-is.

5 LIMITATIONS AND FUTURE WORK

This work attempts to raise the level of abstraction of existing RTL circuits to then automatically pipeline the designs without requiring a human in the compile cycle. This is a work-in-progress and we limited our discussion to fully-pipelined input circuits. Future work will extend the ideas presented here to support circuits with loops, unbalanced register paths, and memories.

Our approach is akin to considering RTL as a description to be *interpreted* rather than be used as-is. Our goal is to enable better design automation for pre-existing RTL designs by exposing CAD optimization opportunities beyond those considered by current FPGA CAD tools that need to preserve all RTL semantics.

REFERENCES

[1] 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* 13, 2, Article 24 (sep 2013). https://doi.org/10.1145/2514740

[2] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (apr 2011), 473–491. https://doi.org/10.1109/TCAD.2011.2110592

[3] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 209–216. https://doi.org/10.1109/ICCAD.2017.8203780

[4] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-Level Synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA). ACM, 127–136. https://doi.org/10.1145/3174243.3174264

[5] Tian Tan, Eriko Nurvitadhi, David Shih, and Derek Chiou. 2018. Evaluating the Highly-Pipelined Intel® Stratix® 10 FPGA Architecture Using Open-Source Benchmarks. In *Proceedings of the 2018 International Conference on Field-Programmable Technology* (Naha, Ikinawa, Japan). IEEE, 206–213. https://doi.org/10.1109/FPT.2018.00038

[6] Clifford Wolf. [n.d.]. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/.