WIP: A Flexible Intermediate Language for High-Level Synthesis

Anonymous Author(s)

Abstract

High-level synthesis (HLS) compilers transform high-level, untimed programs into synthesizable RTL designs and have the potential to drastically improve the productivity of accelerator design. However, research on HLS is hindered by existing tools' monolithic integration: existing toolchains tightly couple a specific input language with large, mandatory transformation passes and target-specific output. We posit that the fundamental problem is the lack of a self-contained intermediate language (IL) that can capture both computational semantics and hardware-level resource constraints.

Futil is an in-progress IL that can represent programs at every stage in the chain of transformations from high-level specification to the low-level implementation. The key idea in Futil is a dual representation that captures both the *structure*, consisting of physical hardware resources and their interconnection graph, and the *control*, which orchestrates the structure over time to run a computation. Futil includes a framework for implementing modular compiler passes that transform higher-level control constructs into hardware structure. Through composition of passes, Futil can offer flexible compilation strategies suited to different input languages and different reconfigurable hardware targets. Futil aims to provide a robust and expressive foundation for experimenting with HLS in the same way LLVM does for traditional software compilers.

1 Introduction

High-level synthesis (HLS) compilers transform high-level, untimed programs into synthesizable RTL designs. Widespread adoption and research into HLS tools is a crucial ingredient in the development of reconfigurable accelerators to counteract the stagnation resulting from the wane of Moore's Law. However, research in HLS is hindered by the monolithic integration of the compilers and tooling. Standard HLS tools are intertwined with the semantics of one or two input languages like C++. A given HLS compiler typically targets only a single vendor's FPGA or a single ASIC toolflow. Finally, monolithic HLS toolchains prevent the development of modular, reusable passes that manipulate or optimize accelerator programs.

A key innovation that enables the shared infrastructure of software tools today is the development of intermediate languages (ILs) such as as LLVM [13] that can concisely represent the constraints of target ISAs while also being a flexible frontend for many different languages. We posit that an IL for HLS compilation can enable similar reuse of tooling and infrastructure across many languages and backends.

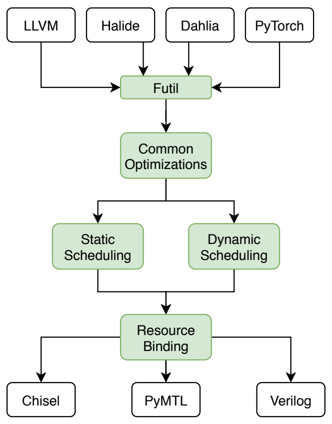

Figure 1. Futil (highlighted) separates microarchitectural decisions from source and target languages.

We see the fundamental responsibilities of an HLS compiler as threefold: parallelization, resource binding, and cycle insertion. A traditional, monolithic HLS compiler intertwines all three responsibilities in a composite heuristic framework. *Parallelization* transforms a sequential program into a parallel schedule. HLS tools typically rely on standard conservative automatic parallelization techniques. During *resource binding*, the compiler maps logical instructions, such as `add32`, onto physical resources on the target fabric, such as adders. Finally, *cycle insertion* generates a finite state machine (FSM) that realizes the logical parallel schedule as physical, cycle-by-cycle timing. The challenging part of building a modular and extensible HLS compiler is cleanly separating these concerns without adversely affecting performance and area. For example, if the parallelization step attempts to maximize the throughput, the resource binding phase has to work harder to minimize resource conflicts.

Our main goal in the design of an IL for HLS is to enable modular passes to do parallelization, resource binding, and cycle insertion by transforming programs in the IL. We propose that such an IL should meet these criteria:

Self contained. An IL is a programming language that should capture the meaning of a program at any stage of compilation. Therefore, it should be possible to rigorously define a program's semantics without reference to the original input code, details of the passes, or *ad hoc* compiler data structures. A self-contained IL allows for modular compiler design because the responsibility for a given pass can be completely defined in terms of its input and output programs.

Hardware aware. An IL for HLS must represent the physical constraints on timing and resources. This way, modular passes can optimize the timing behavior and resource usage of an accelerator design by transforming the IL program. When an IL is both self contained and hardware aware, the semantics of the IL can answer questions about the performance and area of a given program.

Expressive. An IL for HLS must be expressive enough to capture both the computational semantics of different high-level frontend languages as well as the resource and timing constraints of different targets, including commercial FPGAs and various flavors of CGRAs. An expressive IL can enable innovation in new high-level languages for accelerator design that shed the legacy baggage of C [16, 6, 11], and it can facilitate novel reconfigurable hardware designs [17] by providing a starting point for their compiler toolchains.

Futil is an intermediate language for building extensible HLS compilers. Futil explicitly represents resource and timing decisions using a split representation that includes both static hardware *structure* and dynamic logical *control*. In addition to allowing common software optimizations used by HLS compilers, Futil toolchains can easily swap out hardware-focused cycle insertion and resource binding passes.

2 Related Work

Other HLS compilers also rely on intermediate representations. The key difference in Futil is the explicit representation of hardware resources as a complement to imperative control flow. This section contrasts Futil with traditional software IRs and more recent languages that specifically target reconfigurable accelerators.

LLVM and software IRs. xPilot [4] and LegUp [2] are commercially successful HLS toolchains built on LLVM [13]. Using LLVM allowed these tools to reuse complex software optimizations and generate timed RTL by writing monolithic compiler passes to perform cycle insertion and resource binding. Some passes work by adding metadata to the LLVM program to encode hardware-level concerns like timing and resource binding. These metadata formats are undocumented internal data structures, however, and do not allow modular passes to experiment with new mapping strategies. The goal of Futil is to expose these concerns in a language with self-contained semantics, making it easy to inspect and manipulate the compiler's hardware-level decisions.

μIR. μIR [19] is a recent proposal for a C-to-RTL compiler that relies on a parallel extension to LLVM as an input. The μIR compiler represents programs as a graph of asynchronously communicating tasks which let it represent forms of parallelism not manifest in traditional IRs like LLVM. Unlike Futil, it does not attempt to represent physical resources. Frontends therefore cannot control resource mappings, and passes cannot manipulate the allocation of hardware resources to computations.

HPVM. The Heterogeneous Parallel Virtual Machine [12] is a new intermediate representation that targets a wide variety of novel hardware targets, from multicores to GPUs and FPGAs. The key idea is to represent many forms of parallelism in the IR to enable efficient code generation on platforms that exploit parallelism in different ways. We see Futil as a potential *backend* for HPVM when targeting reconfigurable hardware specifically. Unlike HPVM, Futil adds a mechanism for reasoning about the allocation of physical resources to exploit area–parallelism trade-offs.

IRs for HDLs. Modern hardware description languages such as Chisel [1, 9], PyMTL [15], and Magma [7, 5] include IRs for building pass-based hardware optimization frameworks, and LLHD [18] is a standalone IR designed to capture the semantics of traditional HDLs. These IRs are lower level than Futil and target optimization at the bit and wire level. We view these as target backends for Futil.

3 The Futil Language

This section introduces Futil, an IL that enables the design of extensible and modular HLS compilers. Futil programs are composed of *components*. Every component consists of a *structure* part and a *control* part. Structure instantiates subcomponents and the data-flow connections between them, and control describes how the structure behaves over time.

The separation of structure and control is a key idea in Futil's design. Structure lets Futil represent hardware-level concerns such as resource sharing and control enables reasoning about a program's computational semantics. Passes in a Futil-based compiler shift parts of the program from software-like control to hardware-like structure, eventually producing a mostly-structural program that closely corresponds to a hardware implementation.

We next describe the Futil language in more detail. Sections 3.1 and 3.2 then describe how lowering and optimization passes work in a compiler based on Futil.

Components. A Futil program defines a component with the `define/component` syntax form. A component consists of a name, a list of named input ports and their bitwidths, a list of output ports, a structure list, and a control expression. The syntax looks like this:

```
(define/component component_name
  ([inputA 32] [inputB 1])
  ([output 32])
  ( /* structure */ )
    /* control */ )
```

Futil backends provide implementations for primitive components that other components can instantiate. Primitive components include adders, multiplexers, registers, memories, etc. Backends also supply area, energy, and timing information for the primitives that can be used in passes.

Structure sub-language. Futil components describe their static hardware structure as a graph where nodes are subcomponents and edges are wired connections. Subcomponents are declared with `new`. The `->` statement connects ports between component instances. The syntax `(@ comp portA)` references the port named `portA` on the component named `comp`. This example shows a structure with two subcomponents and two connections:

```
[new B (comp/memory 8)] // component instantiations
[new dot my/register]
[-> (@ B out) (@ dot in)] // port connections
[-> (@ dot out) (@ this out)]
```

The `new` statements can optionally provide parameters to components to specify properties like bitwidths or memory sizes. The keyword `this` in the last statement refers to the component currently being defined.

Control sub-language. The control sub-language in Futil orchestrates the behavior of the components instantiated in the structure. It resembles an ordinary imperative programming language augmented with parallelism.

The central statement that Futil control can execute is `enable`, which *activates* one or more structural components, running their respective computations:

```
(enable A reg0) // Execute A and allow writes to reg0
```

Futil provides two composition operators: `par` to execute components in parallel, and `seq` to execute components in sequence. A `par` or `seq` statement finishes executing when all sub-components are done.

```
(seq (enable A) (enable B) (enable C))
(par (enable A) (enable B) (enable C))
```

`if` and `while` statements allow expressing more complex control-flow.

```
(if (@ comp port)
  (seq (enable A) ...))
```

```
(while (@ comp port)
  (seq (enable A) ...))
```

The composition primitives (`seq` and `par`) in Futil give frontend compilers the ability to concisely express a rich class of *program schedules*, while the control-flow primitives (`if` and `while`) allow programmers to express high-level control in a similar fashion to high-level programming languages, making it easier to compile frontend languages into Futil. These high-level control statements are compiled away for Futil toolchain.

3.1 Compilation

Futil aims to enable a compiler to translate high-level programs to low-level hardware implementations. High-level programs, early in the compiler pipeline, are control heavy while lower-level programs consist of more structure and less control. A purely structural program has Verilog-like semantics and admits straightforward translation to RTL. In this section, we demonstrate how Futil represents the traditional scheduling and binding phases of an HLS compiler.

Scheduling. In an HLS compiler, the scheduling phase assigns each logical operation of a program to a specific clock cycle. The *control* language of each component in Futil represents a coarse-grained schedule; it describes a happens-before ordering of operations rather than a strict assignment of operations to clock cycles. Scheduling in Futil is the task of generating cycle-level timing for a component that implements its control description. Futil represents cycle-level timing with a *global schedule* with the following form:

```
(seq (enable A B) (enable C D) ...)
```

In a global schedule, operations in each `enable` correspond to actions for that clock cycle. Futil assigns each `enable` statement a precise latency by recursively computing the timing information of each sub-component.

While scheduling in traditional HLS compilers happens in a monolithic phase, Futil allows for the process of generating the global schedule to be broken up into several modular passes. For example, one pass could be responsible for replacing `while` loops with equivalent structure and another pass could flatten nested `seq` / `par` constructs. This makes it easier to experiment with small changes to the scheduler.

Binding. The binding phase of an HLS compiler assigns physical resources to each logical resource, possibly reusing physical resources multiple times. Replacing this phase is challenging because it typically uses target-specific heuristics. The compiler implicitly maintains timing information and target specification to enable binding.

Since Futil directly represents resources and timing information, binding is *just another optimization pass.* It can be implemented using small, modular passes that remove duplicate components, insert multiplexing logic, and modify the control. For example, consider a program that uses multipliers A and B at two different times:

```
([new A (comp/mult 32)] // structure
 [new B (comp/mult 32)] ...)
(seq (enable A)         // control
     (enable B))
```

Since the multipliers execute in sequence, a compiler pass may decide to reuse the multiplier `A` and reduce the area of the final design by multiplexing the inputs and outputs of A:

```
([new A (comp/mult 32)] // structure
 [new M (comp/mux  32)] ...) // define new multiplexer
(seq (enable A M)       // control
     (enable A M))
```

Because resource binding is decoupled from the rest of the compiler, experimenting with different binding strategies for different targets is straightforward.

3.2 Optimization Passes

Through its explicit representation, Futil can represent both traditional HLS optimizations such as loop unrolling as well as timing and resource-directed optimizations.

Loop unrolling. Area–performance trade-offs such as loop unrolling are common in HLS programming. HLS loop unrolling (distinct from software loop unrolling) duplicates hardware to execute independent loop iterations in parallel, increasing throughput. Traditional HLS tools represent unrolling using `#pragma` annotations on C loops.

Futil can easily represent loop unrolling by explicitly making copies of the loop structure and parallelizing the loop control. Consider this Futil program:

```
([new A  (comp/memory   8)] // structure
 [new m0 (comp/mult    32)]
 [new i0 (comp/iter 0 1 8)])
(while (@ i0 stop)          // control
  (enable A m0))
```

Unrolling the loop once results in code like this:

```
([new A0 (comp/memory 4)]   [new A1 (comp/memory 4)]
 [new m0 (comp/mult  32)]   [new m1 (comp/mult  32)]
 [new i0 (comp/iter 0 2 8)] [new i1 (comp/iter 1 2 9)])
(par (while (@ i0 stop) (enable A0 m0))
     (while (@ i1 stop) (enable A1 m1)))
```

Operator chaining. *Operator chaining* is an optimization that improves the overall latency of a design by scheduling sequences of operations into a single clock cycle if the latency of the sequence of operations is shorter than the estimated cycle length. In Futil, this can be expressed as a control transformation. Programs of the following form:

```
(seq (enable A) (enable B) (enable C) ...)
```

could be translated into:

```
(seq (enable A B) (enable C) ...)
```

Software-style optimizations. Futil enables classical compiler optimizations to be performed on the control language. Performing these optimizations in Futil, rather than in a software IR, allows these optimizations to compose cleanly with hardware optimizations. For example, classic loop-invariant code motion (LICM) lifts statements out of loop when their behavior is the same on every iteration, as in this Futil loop:

```
(while (@ i0 out)
  (seq (enable A B mult c) // c = A * B
       (enable x c)))      // x = x * c
```

Here, c is recomputed every loop iteration but its value never changes. A Futil LICM pass results in code like this:

```
(seq (enable A B mult c)  // c = A * B
     (while (@ io out)
       (seq (enable x c)  /* x = x * c */ )))
```

4 Future Directions

Futil aims to make the development of HLS compilers flexible and modular. Rapid iteration of compiler technologies is a critical ingredient in widespread adoption of reconfigurable accelerators. We enumerate opportunities to build on Futil to enable future research.

Latency-insensitive design. Dynamic scheduling [10] is a scheduling strategy that leverages latency-insensitive interfaces to improve the execution time of designs that extensively use data-dependent control. Currently, Futil can represent static schedules but not dynamic ones. We plan to augment Futil with variants of `enable` that can wait asynchronously for a component to signal its completion.

Verified compilation. The last decade has produced breakthroughs in formal verification of software compilers. Futil's pass-based design will enable easier verification of HLS compilers. CompCert [14] and similar verified compilers use refinement to build up a proof of correctness for the compiler from modular proofs that individual passes preserve the semantics of the program. A self-contained IL semantics is a critical first step toward formulating a correctness theorem for individual compiler passes. Because Futil modularizes complex passes such as scheduling and binding, it will allow verification efforts to scale.

Hardware backends. Futil currently generates accelerators for commercial FPGAs. We want to extend Futil to target other hardware backends such as emerging coarse-grained reconfigurable arrays (CGRAs) and real silicon via ASIC toolchains. The challenge in targeting CGRAs is that, unlike FPGAs, their design and capabilities can vary wildly: some use static scheduling while some are purely dynamically scheduled [8]; each CGRA bakes different logic into its processing elements [3]; and each CGRA can distribute on-chip memories differently [17]. Meanwhile, ASIC design offers total flexibility in the instantiation of structural resources. We expect Futil's modular pass framework to enable us to add ASIC- and CGRA-specific optimization and binding passes, making it easier to develop toolchains for these technologies.

New design languages. Traditional HLS relies on C and C-like input languages, but repurposing a legacy software language introduces a semantic gap between the programmer's view and the compiler's output. We see an opportunity to design new HLS languages that more faithfully reflect the constraints of accelerator implementation while still offering high-level algorithmic semantics. Futil's representation of structural resources and hardware timing will let novel language frontends exert more control over the hardware they generate without resorting to generating Verilog.

References

[1] 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 *Design Automation Conference (DAC)*.

[2] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[3] S. Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Helge Anderson. 2017. CGRA-ME: A unified framework for CGRA modelling and exploration. *International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (2017).

[4] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. 2006. Platform-Based Behavior-Level and System-Level Synthesis. In *International SoC Conference*.

[5] Ross Daly, Lenny Truong, and Pat Hanrahan. 2018. Invoking and Linking Generators from Multiple Hardware Languages using CoreIR. In *Workshop on Open-Source EDA Technology (WOSET)*.

[6] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-Directed Scheduling of Streaming Accelerators. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[7] Pat Hanrahan. [n.d.]. Magma. https://github.com/phanrahan/magma.

[8] Yuanjie Huang, Paolo Ienne, Olivier Temam, Yunji Chen, and Chengyong Wu. 2013. Elastic CGRAs. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[9] Adam M. Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *International Conference on Computer-Aided Design (ICCAD)*.

[10] Lana Josipoviundefined, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-Level Synthesis. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[11] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: a language and compiler for application accelerators. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[12] Maria Kotsifakou, Prakalp Srivastava, Matthew D. Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. 2018. HPVM: Heterogeneous Parallel Virtual Machine. In *PPoPP*.

[13] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization (CGO)*.

[14] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Communications of the ACM (CACM)* (July 2009), 107–115.

[15] Derek Lockhart, Gary Zibrat, and Christopher Batten. 2014. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[16] 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[17] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matthew Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christoforos E. Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for parallel patterns. In *International Symposium on Computer Architecture (ISCA)*.

[18] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. 2020. LLHD: A Multi-Level Intermediate Representation for Hardware Description Languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[19] 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 *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.