Case Study: Software and Tool Challenges Encountered in Parameterizing a Domain-Specific Accelerator

Radhika Ghosal
Harvard University
Cambridge, MA, USA

Sabrina M. Neuman
Harvard University
Cambridge, MA, USA

ABSTRACT

For application domains with a range of different deployment scenarios, such as robotics, it can be advantageous to parameterize accelerator designs so that they can be easily re-designed and customized per-deployment. Previous work prescribed a systematic methodology to parameterize a class of robotics accelerators according to the physical features of the robot, and manually implemented an accelerator for robot rigid body dynamics gradients for a manipulator arm. However, to *automate* the process of accelerator re-deployment for future work, it has been necessary to elaborate on the tool infrastructure to create a fully parameterized flow. Without delving into the details of our particular architectural framework, this case study informally provides a retrospective of some of the challenges with tools and software that we have encountered while parameterizing our domain-specific accelerator design.

1 INTRODUCTION

Systematic domain-specific hardware design methodologies are necessary for rapid re-targeting of accelerator designs across different deployment scenarios. In the robotics application domain, for example, previous work introducing *robomorphic computing* has demonstrated that there are substantial performance gains to be had from systematically customizing hardware accelerators based on the physical structure of the robot model [4]. While the robomorphic computing methodology applies across a broad class of robots, however, the evaluation in that work only manually implemented a single accelerator for a single robot model.

To realize the goal of generalizable re-deployment of accelerators across diverse target scenarios (e.g., different robot models), systematic hardware design methodologies like robomorphic computing must be *extended* and built upon to create *parameterized, automated* domain-specific accelerator design frameworks. However, building parameterized accelerator design flows is a challenging process that can involve stitching together combinations of different software tools, hardware compilers, and programming languages.

In this work, we describe challenges encountered in parameterizing a domain-specific accelerator for robotics applications. Abstracting away the details of our particular application and architectural framework, in this case study we provide an informal retrospective of our experiences with hardware design tools and software, noting challenges and opportunities for future hardware design work.

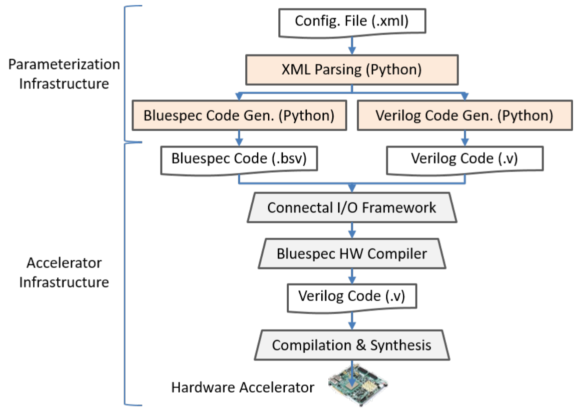

Figure 1: Accelerator parameterization infrastructure.

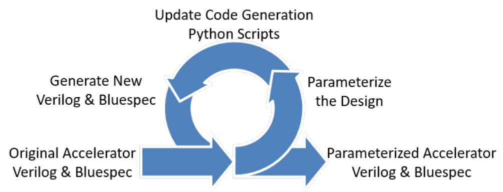
Update Code Generation Python Scripts
Generate New Verilog & Bluespec
Parameterize the Design
Original Accelerator Verilog & Bluespec
Parameterized Accelerator Verilog & Bluespec

Figure 2: Overview of development workflow.

2 HIGH-LEVEL INFRASTRUCTURE & FLOW

Original Accelerator. Our starting point was a domain-specific accelerator [4] written in Verilog and Bluespec System Verilog [6], with most of the design in Verilog. Bluespec was used for CPU-FPGA communication, and queues for pipelining accelerator stages. Using Bluespec enabled the use of the Connectal framework [3] for I/O with a host CPU. Connectal sets up CPU-side direct memory accesses in C++ and FPGA-side buffers for handling PCIe I/O.

Parameterization & Workflow. To automate re-deployment of the original accelerator, we built a parameterization infrastructure that takes an XML configuration file as input; parses out parameters that impact the accelerator; and feeds those parameters into our accelerator template, implemented as Bluespec and Verilog code generation scripts written in Python (Fig. 1). Our resulting development workflow was that each time we parameterized a feature of the accelerator, we edited our code generation Python scripts to reflect that change, and generated new code to test (Fig. 2).

Design Decisions. For designing our parameterization infrastructure (Fig. 1), we narrowed down to several options: (a) rewrite the Verilog in a more metaprogramming-friendly HDL, e.g., Chisel [1]; (b) use a hardware compiler framework, such as Calyx [5] or CIRCT [2], to generate hardware from higher-level specifications; or (c) make our own generation infrastructure. We did not consider HLS tools such as Vivado HLS because we started with an existing Verilog accelerator, and wanted to avoid massaging HLS input to re-produce the same processor and risking poorly-optimized output compared to hand-tuned RTL [8]. SystemVerilog [7] would let us us readily parameterize some features (e.g., instantiating variable numbers of modules), but was not suitable for others (e.g., generating matrices of wires and registers with specific sparsity patterns). Finally, none of the tools we were considering would let us easily re-generate the original Bluespec portions of the accelerator.

We ultimately chose to use a scripting language for generating hardware because we were starting with a relatively simple existing accelerator written in Verilog and Bluespec, and had specific plans for the parameterization methodology. We weighed this against the upfront cost of adopting more fully-featured hardware generation tools, and decided to avoid the overhead of learning new languages (e.g., Scala, Chisel). We were not making any program transformations on the original algorithm or addressing complicated new algorithms, so a full compiler framework was not needed; it was not worth incurring the upfront cost to have access to abstract syntax trees or control and data-flow graphs. Since we were generalizing an already-existing accelerator template, our focus was not on making optimizations *in* the template itself, but simply parsing the template arguments from our configuration file, and substituting them into the template in a systematic manner. This could be done in any easy-to-use scripting language; we chose to use Python to directly write out Verilog and Bluespec code.

3 CODE CASE STUDY: DYNAMIC INDEXING

In this section, we examine a specific code case study example (Alg. 1) to illustrate some of the challenges we have faced in developing, debugging, and scaling our accelerator design. This code was implemented in Bluespec (which then generates Verilog) in our design, but the issues we discuss are tool-agnostic and are applicable to any stack consisting of multiple layers of code generation.

```
Original Code: Dynamically indexing a 2D array.
CONST i_max=2; CONST num_PEs=2; REG i=0;
// { {i0: PE1,PE2}, {i1: PE1,PE2} }
CONST d1_table = { {0,1}, {1,0} };
CONST d2_table = { {1,0}, {0,1} };

rule getResult(proc.output_ready() && i < i_max);
  d1_PE1=d1_table[i][0]; d1_PE2=d1_table[i][1];
  d2_PE1=d2_table[i][0]; d2_PE2=d2_table[i][1];
  out[d1_PE1][d2_PE1] = proc.PE1_out();
  out[d1_PE2][d2_PE2] = proc.PE2_out();
  ...
Attempt #2: Declare a function.
(* noinline *) function mux_update(old[2][2],d1,d2,out)
  new=old; new[d1][d2]=out; return old; endfunction;

rule getResult(proc.output_ready() && i < i_max);
  out=mux_update(out,d1_PE1,d2_PE1,proc.PE1_out());
  out=mux_update(out,d1_PE2,d2_PE2,proc.PE2_out());
  ...
Attempt #3: Unroll dynamic indexing into series of muxes.
rule getResult(proc.output_ready() && i < i_max);
  case (i)
  0: out[0][1]=proc.PE1_out(); out[1][0]=proc.PE2_out();
  1: out[1][0]=proc.PE1_out(); out[0][1]=proc.PE2_out();
  ...
```

Algorithm 1: Code Case Study: Dynamically indexed array.

Alg. 1 shows an example of updating a 2D array for each index `i` representing a step in a programmed schedule. For small schedule lengths and numbers of processing elements (PEs), the *Original Code* compiled and simulated successfully. However, it did not scale for large `i_max` and `num_PEs`, and in fact never finished compiling. We then explicitly declared a function, `mux_update` (Alg. 1, *Attempt #2*), to help Bluespec infer a linear chain of muxes and not inline them; this did compile and simulate successfully. However when synthesizing, we found a combinatorial explosion in the number of LUTs used by these muxes, suggesting that the Bluespec compiler was trying to exhaustively generate muxes for every possible value of `d1` and `d2`. We ended up fully unrolling the matrix updates manually in case statements (Alg. 1, *Attempt #3*). A compiler should be able to infer that `d1_PE1`... `d2_PE2` can be statically determined by the index `i`, and automatically generate the case statements in *Attempt #3*. Unfortunately, we had to implement unrolling manually, adding to code complexity and complicating the debugging process. Challenges raised by this example follow:

Resource utilization/compilation blowing up after scaling. When code-generating HDL using higher-level tools like Bluespec, it is not always obvious which lines of high-level code will cause resource utilization to skyrocket as parameters vary. We discovered scalability issues in our Bluespec template only *after* experimentally stress-testing our template using larger indices and more PEs.

Iterating between hardware template and instantiation. As the Alg. 1 example demonstrates, our workflow was a cycle of changing the hardware template, seeing how it scales, and iterating (Fig. 2). Our workflow had hardware generation in-the-loop, so the tools guided design decisions. In debugging, we often directly edited the codegen-ed output HDL files. Once the design worked in simulation, we back-propagated changes to our generation scripts. This process could be tedious and error-prone for big design changes.

Interpretability of input code to output code generated. When working with multiple layers of code generation tools, understanding which input snippet of code produced what output is essential for tracking down errors and figuring out which codegen tool is responsible. We have manually inspected and tested codegen-ed output, and as the Alg. 1 example shows, it is not always easy to anticipate what higher-level code will cause an explosion in generated HDL. Worse, addressing these issues can lead to solutions that add to code complexity, further complicating interpretability.

4 CONCLUSION AND FUTURE DIRECTIONS

We contribute this case study to the conversation in the accelerator tools design community. Hardware generation tools have come a long way, but a full rewrite of an already-existing codebase in order to use them is not always feasible. As potential users, we would like to see more examples of tools interoperating with legacy HDL modules, including generating code for *other* code generation tools already in a pre-existing system stack (e.g., Connectal, Bluespec). Additionally, making hardware compilation interpretable is essential for increasing adoption among hardware designers. The successful growth of domain-specific computing is inextricably tied to the development of elegant and user-friendly hardware design solutions. We look forward to hardware generation tools enabling future agile domain-specific architecture development.

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 *DAC Design automation conference 2012*. IEEE, 1212–1221.

[2] 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)*.

[3] Myron King, Jamey Hicks, and John Ankcorn. 2015. Software-driven hardware development. In *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*. 13–22.

[4] Sabrina M Neuman, Brian Plancher, Thomas Bourgeat, Thierry Tambe, Srinivas Devadas, and Vijay Janapa Reddi. 2021. Robomorphic computing: a design methodology for domain-specific accelerators parameterized by robot morphology. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 674–686.

[5] 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.

[6] 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.

[7] David I Rich. 2003. The evolution of SystemVerilog. *IEEE Design & Test of Computers* 20, 04 (2003), 82–84.

[8] Felix Winterstein, Samuel Bayliss, and George A Constantinides. 2013. High-level synthesis of dynamic data structures: A case study using Vivado HLS. In *2013 International conference on field-programmable technology (FPT)*. IEEE, 362–365.