Single-Source Hardware-Software Codesign

Blaise-Pascal Tine
Georgia Tech
Atlanta, Georgia
blaise.tine@gatech.edu

Sudhakar Yalamanchili
Georgia Tech
Atlanta, Georgia
sudah@gatech.edu

Hyesoon Kim
Georgia Tech
Atlanta, Georgia
hyesoon@cc.gatech.edu

***Abstract*—With Moore's Law coming to an end, hardware specialization and systems on chips are providing new opportunities for continuing performance scaling while reducing the energy cost of computation. However, the current hardware design methodologies require significant engineering efforts and domain expertise, making the design process unscalable. More importantly, hardware specialization demands for a much tighter software and hardware co-design environment to exploit domain-specific optimizations and design efficiency. In this work, we introduce Cash, a single-source hardware-software co-design framework. Cash leverages the unique efficiency and generative attributes of Modern C++ to provide new co-design programming abstractions that enable a single-source unified development environment for both hardware and software.**

I. Introduction

The current era of hardware specialization is calling for better integration between hardware and software development methodologies to exploit domain-specific optimizations and reduce verification cost. Currently, traditional hardware and software development processes are completely decoupled, using different languages, and tools, making the iterative software-hardware optimization process inefficient. Existing solutions for closing the software-hardware gap use High-Level Synthesis (HLS) [4] to generate hardware directly from software. While HLS provides good productivity, the generated hardware is sub-optimal compared to RTL-based implementations. Hardware Construction Languages (HCL) [1] [2] have raised the abstraction of RTL, making the design and modeling more productive, but they offer limited co-design support. SystemC provides a unified development environment for software and hardware, but its RTL abstraction is too verbose and limited, forcing designers to rely on traditional HDLs. In this work, we introduce Cash, a single-source hardware-software co-design framework. Cash implements new programming constructs to enable software and hardware co-design with multiple levels of modeling abstraction. Hardware blocks implemented in Cash can interface directly with application software, including HLS, and Cycle Approximate Simulators (CAS) [3]. Under the hood, the Cash Framework integrates a JIT compiler [5] that produces a high-speed RTL simulator and exports the design to Verilog HDL for synthesis on FPGAs or ASIC.

II. Single-Source Hardware/Software Codesign

A. The Cash Single-Source Design Methodology

Figure 3 illustrates the Cash single-source design methodology starting with the design specifications stage where the DSL allows the sharing of data structures and configuration from the host application. The second stage, functional modeling, enables the description of hardware components in behavioral C++, allowing early evaluation to refine the design specifications. The third stage, cycle-Level modeling, allows the cycle-level description of hardware components with optional integration with CAS for Design-space exploration. The description of RTL models happens in the fourth stage. In the fifth stage, the DSL provides an API extension to integrates RTL models with HLS for fast prototyping on FPGA or used as libraries to optimized HLS.

B. Cross-Domain Type Sharing

One major source of bugs in heterogeneous systems development is the replication of type definitions among the various design spaces. For instance, an application may define the layout of a particular data structure that an accelerator should consume. In a discrete design environment where type sharing is not possible, the same layout is redefined in the hardware design space which causes maintenance issues. Cross-domain type sharing addresses this issue by enabling a single type definition to be shared between the application space and the simulated hardware. The Cash DSL implements two distinct namespaces; the system space, defining the type domain of the host program; the logic space, defining the type domain of the hardware. All the primitive data types in the DSL have two implementations; one that resides in system space, and the other that resides in logic space. The DSL provides

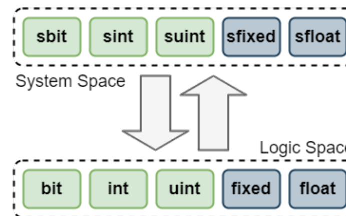

Fig. 1: Type Translation

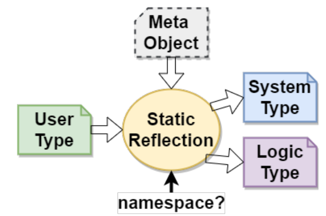

Fig. 2: Meta Reflection

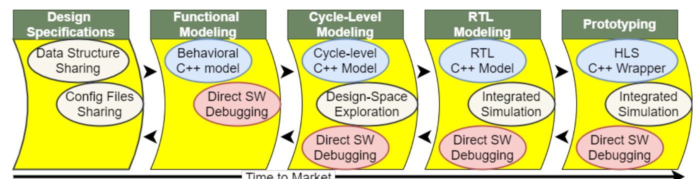

Fig. 3: The Cash Single-Source Design Methodology

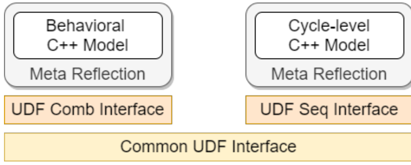

Fig. 4: Cash UDF Interface

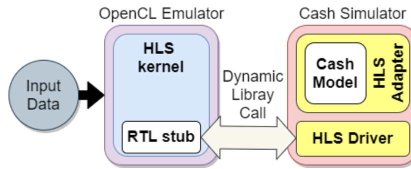

Fig. 5: Cash-HLS Simulation

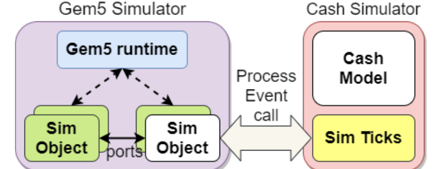

Fig. 6: Cash-CAS Simulator

```
1  #include <cash.h>
2
3  __struct (PerfCounter,
4    (ch_uint16) cycles
5  );
6
7  template <int I, int O>
8  class MAC {
9    __io (
10     __in (ch_int<I>) ia, ib,
11     __out(ch_int<I>) oa, ob,
12     __out(ch_int<O>) out
13   );
14
15   void describe() {
16     io.oa  = ch_delay(io.ia);
17     io.ob  = ch_delay(io.ib);
18     io.out = ch_delay<O>(io.oc + io.ia * io.ib);
19   }
20 };
21
22 template <int I, int O, int N, int P, int M>
23 class MatMul {
24   __io (
25     __in (ch_vec<ch_int<I>,N>) a, b,
26     __out(ch_vec<ch_int<O>,P,N>) out,
27     __out(ch_bool) valid,
28     __out(PerfCounter) perfcntr
29   );
30
31   void describe() {
32     ch_vec<ch_module<MAC<I,O>,P,N> macs;
33     ch_counter<log2ceil(N+P+M)> ctr;
34
35     for (int j = 0; j < N; ++j) {
36       auto a = ch_delay(io.a[j], j);
37       for (int i = 0; i < P; ++i) {
38         auto b = ch_delay(io.b[i], i);
39         macs[j][i].io.ia = i ? macs[j][i-1].io.oa : a;
40         macs[j][i].io.ib = j ? macs[j-1][i].io.ob : b;
41         io.out[j][i] = macs[j][i].io.out;
42       }
43     }
44     io.valid = (ctr == N+P+M);
45     io.perf  = PerfCounter{N+P+M};
46   }
47 };
48
49 void main() {
50   matrix<int,2,3> a{0,1,0,1,7,8};
51   matrix<int,3,4> b{8,7,8,9,5,5,6,4,7,0,0,0};
52
53   ch_device<MatMul<8,24,2,4,3>> matmul;
54   ch_simulator simulator(matmul);
55   simulator.run();
56
57   std::cout << "result=" << matmul.io.out;
58   std::cout << "perf counter=" << matmul.io.perf;
59 }
```

Listing 1: Generic MatMul written in Cash

utility operators for converting data types from one space to the other (see Figure 1). For user-defined types, Cash implements a static reflection engine (SRE) based on meta-objects (see Figure 2). For each user-defined type family (e.g. struct), SRE maintains a meta object that defines the transform templates between logic and system spaces and uses it to automate the translation. For instance, in our MatMul sample in listing 1, we defined a custom type *PerfCounter* for capturing hardware statistics (line 3) that is accessible by both the MatMul accelerator class where it is written (line 45) and the host application code where it is read (see line 58) and any change to its layout is visible to both programs. Cross-Domain Type Sharing powers the Cash's type system, enabling support for combinational and sequential User-Defined functions (UDFs) (Figure 4) used for behavioral and cycle-level modeling, respectively. The feature is also used to implement the Cash HLS simulator (Figure 5), allowing the development and debugging of mixed HLS code with

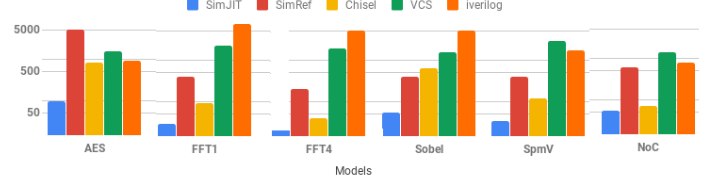

Fig. 7: Simulators Speed comparison (sec)

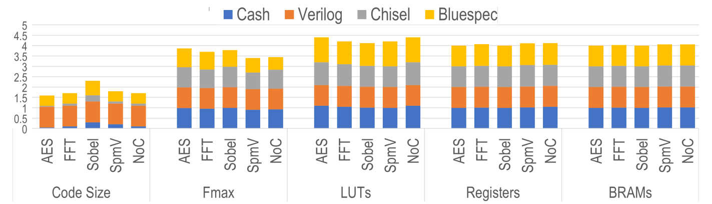

Fig. 8: Normalized Synthesis Results versus Verilog

optimized Cash RTL. Lastly, another application is when using Cash RTL modules in a CAS environment like Gem5 [3] (Figure 6) where SimOjects seamlessly interface with Cash C++ modules, enabling fine-grain design space exploration.

III. Evaluation

For evaluation, we used a micro-benchmark that consists of the following hardware blocks: a 128-bit AES engine (AES), A radix 2^2 FFT, a parallel 4-lanes radix 2^2 FFT, an 8-bit Sobel filter (Sobel), a fixed-point Sparse Matrix Multiplier (SpMV), and a Network-on-Chip router (NoC). The synthesis was performed on Intel Quartus Pro 17 and the simulation on Intel Xeon E5 CPU. Figure 7 compares the Cash simulator (SimJIT) with other RTL simulators including VCS, IVerilog, and Verilator, showing a 3x to 7x speedup across all benchmarks. The simulation speed is particularly important during codesign when interacting with software like CAS or HLS. Figure 8 compares Cash synthesis results with Verilog, Chisel, and Bluespec. Similar to Chisel, the Cash RTL is just as efficient as Verilog. Bluespec synthesis however has a slight degradation in LUT utilization which affects ts overall fmax. The graph also shows Cash's design code size efficiency.

IV. Conclusion

The Cash framework leverages Cross-Domain Type Sharing to provide seamless integration of RTL code with software in the CAS and HLS environments where the same language and development tool can be utilized to maximize productivity. The Cash framework has been made available for public use at https://gtcasl.github.io/cash/.

REFERENCES

[1] Arvind, “Bluespec: A language for hardware design, simulation, synthesis and verification invited talk,” ser. MEMOCODE ’03.
[2] J. Bachrach, H. Vo, B. Richards, and Y. L. et al, “Chisel: Constructing hardware in a scala embedded language,” 2012, pp. 1212–1221.
[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, and J. e. a. Hestness, “The gem5 simulator,” *SIGARCH*, p. 1–7, 2011.
[4] A. Canis, J. Choi, and M. e. a. Aldham, “Legup: High-level synthesis for fpga-based processor/accelerator systems,” in *FPGA*, 2011, p. 33–36.
[5] B.-P. Tine, S. Yalamanchili, and H. Kim, “Tango: An optimizing compiler for just-in-time rtl simulation,” in *DATE*, 2020.