High-level Compilation for Reconfigurable Architectures

Jonathan Babb (jbabb@mit.edu)
MIT Laboratory for Computer Science

What if instead of designing general-purpose hardware, computer architects wrote compilers that generated the specific hardware an application required? Previous reconfigurable computing research has shown that we *can* build application-specific architectures out of a collection of simple, programmable building blocks. However, compiling even modest high-level programs down to such hardware is still an open research question. This abstract presents an initial architecture compiler in a framework that can transform a sequential program into reconfigurable hardware.

Compilation Framework

A successful architecture compilation system must incorporate both the latest code optimization and parallelization techniques as well as state-of-the-art hardware synthesis technology. Our new framework (Figure 1) thus leverages both the Raw Compiler frontend [2] and the Virtual Wires Compiler [1] backend.

The classic optimization/parallelization phase is performed by the frontend of the Raw Compiler [2]. This compiler, built on the SUIF compiler infrastructure, takes as input a sequential program and produces a set of communicating low-level threads in a generic intermediate instruction format. Important optimizations include locality sensitive partitioning and static memory disambiguation.

For our current reconfigurable target, a multi-FPGA logic emulator, we employ a commercial RTL synthesizer coupled with the Virtual Wires Compiler [1] as the backend technology mapper. The input is an RTL-level verilog netlist describing one or more communicating finite state machines (FSMs). This netlist describes the cycle-by-cycle operation of the architecture. This input is then transformed into a physical description of the target hardware technology – a reconfigurable hardware bitstream.

Architecture Compilation

The unbounded nature of generating a new architecture for each program requires the adoption of rigid compilation disciplines to create a working system. We describe the architecture compiler in the context of three key disciplines: spatial partitioning, deep structuring, and free-form scheduling.

Spatial partitioning is required if a compiled architecture is larger than a small number of gates to avoid impacting clock cycle time with long wires and to keep the hardware synthesis problem tractable. The architecture compiler remains faithful to the partitioning created by the Raw frontend, mapping the parallel threads into communicating finite state machines connected by specialized hardware datapaths.

Deep structuring takes advantage of program information not obvious in the surface structure of the program to create specialized hardware structures. Information from pointer analysis is used to determine which variables and array elements can be staticly referenced and grouped into specialized memories. Information from bitwidth analysis is used to specialize the datapaths, function units, and memories to the exact number of bits required.

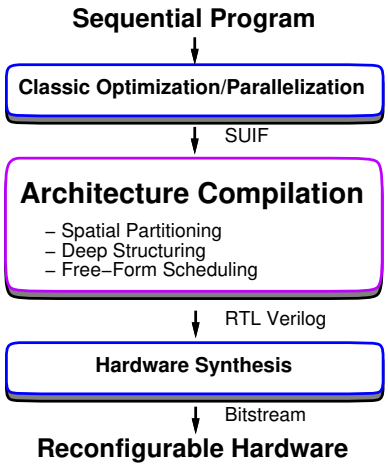

Figure 1: Compiler Flow

In *free-form scheduling*, operations within a basic block are scheduled to clock cycles, assuming unlimited non-memory resources. Resource allocation is thus a secondary constraint to be optimized during later hardware synthesis. In addition, the current approach extends the space-time scheduling found in the Raw Compiler to support instruction parallelism within a thread, chaining of dependent operations in a single cycle, multiple memories, simple loop pipelining, and instruction predication.

The computation model resulting from these disciplines consists of communicating finite state machines with datapaths and multiple memories. Intermediate program variables are mapped to individual hardware registers and distinguishable arrays are mapped to distinctive hardware memories. Specialized datapaths and multiplexers route communication between specialized function units and memory elements as required by the FSM schedule, which hardwires the program control flow. Finally, a synthesized communication network supports communication between FSMs and external I/O.

We are just starting to compile the integer benchmarks of the Raw Benchmark Suite using this new compiler and will be reporting the results in future publications.

References

[1] J. Babb et al. Logic emulation with virtual wires. *IEEE Transactions on Computer Aided Design*, 16(6):609–626, June 1997.

[2] W. Lee et al. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *ASPLOS98*, San Jose, California, October 1998.