Kiwi Scientific Acceleration at Large

Incremental Compilation and Multi-FPGA HLS Demo.

David J. Greaves
University of Cambridge Computer Laboratory

***Abstract*—The Kiwi project revolves around a compiler that converts C# .NET bytecode into Verilog RTL and/or SystemC. An alpha version of the Kiwi toolchain is now open source and a user community is growing.**

We will demonstrate an incremental approach to large system assembly of HLS and blackbox components, based on an extended IP-XACT intermediate representation. We show how to address multi-FPGA designs with object passing between components, automatic configuration of shared memory maps and automatic assembly of debugging infrastructure.

We will also demonstrate the use of the unsafe subset of the C# language for type casting between byte arrays and structures which is a common coding style for network protocol implementations. Unsafe programming is also needed for user-coded memory managers that need to essentially perform address arithmetic, but such procedures can commonly defeat the memory pool disambiguation algorithms in static analysis.

I. Introduction

Kiwi is a compiler and library and infrastructure for hardware accelerator synthesis and general support for high-performance scientific computing. The output is intended for execution on FPGA or possibly in custom silicon ASIC.

We aim to compile a fairly broad subset of the **concurrent** C# language subject to some restrictions:

In Kiwi 1, the prior version, we provided:

- Program can freely instantiate classes but not at run time - a fixed number of instantiation operations must be detectable at compile time.
- Array and heap structure sizes must all be statically determinable (i.e. at compile time).
- Program can use recursion but the maximum calling depth must be statically determined.
- Stack and heap must have same shape at each run-time iteration of non-unwound loops. In other words, every allocation made in the outer loop of your algorithm must be matched with an equivalent, manifestly-implicit garbage generation event or explicit `obj.Dispose()` or `Kiwi.Dispose(Object obj)` in that outer loop.
- Program can freely create new threads but creation operations statically determined too.
- Manual assembly of blocks from separate compilations is needed by editing wrapper RTL.

In Kiwi 2, (available 3Q2017), we relax the static restrictions and allow the size of data structures in DRAM to be determined at runtime. Kiwi 2, supports three major compilation modes. These can be mixed in a single design, at a subsystem granularity, and combined with the new incremental compilation support based on IP-XACT.

1) The Sequencer major mode is 'classical HLS'. For each thread it makes a custom datapath made up of RAMs, ALUs and external DRAM connections and folds the program onto this structure using some small number of clock cycles for each iteration of the inner loops.
2) The Fully-Pipelined Accelerator major mode runs the whole subsystem every clock tick, accepting new data every clock cycle, albeit with some number of clock cycles latency between a particular input appearing at the output.
3) The SoC Render major mode provides C# access to an IP-XACT-driven wiring generator with support for automatic glue logic insertion. This can target multi-FPGA designs and provides a clean mechanism to wrap up third-party IP blocks, such as CAMs.

Specific demonstrables:

1) Incremental compilation with each block described in extended IP-XACT.
2) Wrapping third-party IP blocks so they may be invoked as part of the HLS schedule.
3) Smart FSM decomposition - i.e. subroutine call and return without the state explosion that arises from total inlining.
4) Automatic partition and sharing of main DRAM banks between separate pre-compiled subsystems.
5) Automatic wiring of data paths for method call, control, abend, debug and memory access.
6) Mixing of hard/soft HLS coding style where precise control over packing of operations to clock cycles is sometimes required by the programmer to implement net-level protocols or certain hard-real time constraints.
7) Automated load balancing over instantiated components and automated selection of the number of mirrorable IP blocks to instantiate.
8) Automated partition between FPGAs with marshalling and instantiation of SERDES or other hardware bridges.
9) Automatic connection to AXI-4 components.
10) Using unsafe C# for custom memory managers and network protocol parsing/emitting.