IV EURASIP Seminar on the Hardware Design of DSP Systems
Program-to-hardware transformation and its use in heterogenous high-performance DSP

Madhav P. Desai
Department of Electrical Engineering
Indian Institute of Technology – Bombay, Powai, Mumbai – 400076, INDIA

March 6, 2013

Overview

High-performance DSP can take advantage of several platforms:

- Processors such as multi-cores, GPUs: "easy" to use, but power dissipation is an issue.
- Programmable hardware: lots of parallelism in principle, potential power benefits, difficult to map algorithms to platform. Can they beat power-performance characteristics of processors?
- ASICs: lots of parallelism, high efficiency, will surely beat power-performance characteristics of processors. Difficult to design, and expensive to manufacture.

In terms of efficiency (performance/watt):

```
ASIC   > >  Processor, FPGA
Processor < ? >  FPGA
```

Our work

- Make the path to ASIC/FPGA simpler for algorithm developers.
- Develop (and use) a compiler flow for mapping algorithms to hardware.
- Currently, we have a fully functional tool-chain which takes a C program to VHDL. A multi-threaded C program (with statically defined threads) can be easily mapped to a hardware pipeline.
- We are now working on optimizations (will talk more about that later) and **applications**.

How good is the compiler flow?

We have some experimental data (from 2010)[1] :

- comparison in terms of area, delay and **energy**.
- for a set of programs, use standard commercial synthesis tools to obtain an ASIC starting from the RTL generated by the C-to-RTL flow.
- compare with processor implementations of the same set of programs.

[1] S. Sahasrabuddhe, S. Subramanian, K. Ghosh, K. Arya, M.P. Desai, "A C-to-RTL flow as an energy efficient alternative to embedded processors in digital systems", EUROMICRO 2010, Lille France.

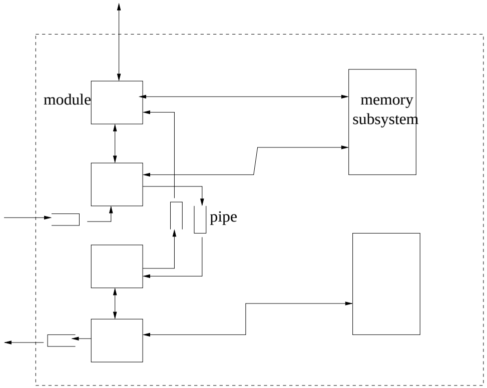
How applicable is it?

We have some data from a practical use-scenario (from 2012)[2]:

- The Click2Netfpga toolchain (joint work with T. Rinta-Aho, M. Karlstedt at Ericsson research) converts a C++ program (generated by the Click modular router) to VHDL.
- Verified on FPGA, performance up to 50% of hand-coded Verilog.

[2] T. Rinta-Aho, M. Karlstedt, M. P. Desai, "The Click2Netfpga toolchain", USENIX ATC 2012, Boston USA.

A system in AHIR



- A system description consists of a collection of modules (a module is similar to a function in a C program). The modules communicate with the environment, with each other (through calls, and through FIFO pipes), and with memory subsystems.

An AHIR module

An AHIR module description is of the form:

$$\mathbf{CP} \times \mathbf{DP} \times \mathbf{S}$$

where **CP** represents control flow, **DP** represents data flow, and **S** represents storage.

The control path **CP**

- The control path is modeled by a petri-net. Each transition is associated with a symbol, which can either be an input symbol or an output symbol.
- When a transition labeled by an input symbol is enabled, it fires when the input symbol is received.
- When a transition labeled by an output symbol is enabled, it fires eventually and emits the output symbol.
- The petri-net has a special structure, which guarantees that **CP** is live and safe.

The data path **DP**

- A directed graph of operators.
- Each operator has a set of request symbols and a set of acknowledge symbols. The operator responds to the arrival of request symbols by eventually generating appropriate acknowledge symbols.
- Operator library: all the standard arithmetic and logical operators, as well as a multiplexor and load/store operators.

The storage subsystem **S**

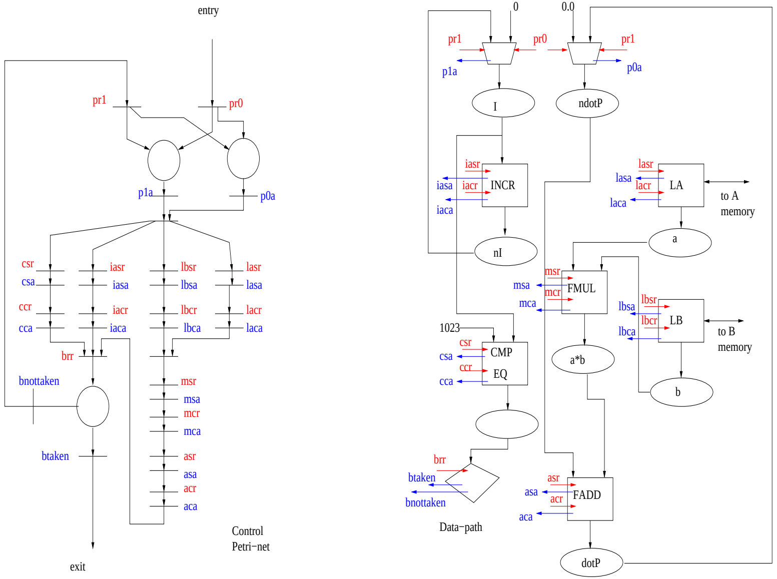
- The storage subsystem connects to the load/store operators in the datapath, and is required to eventually service the load/store requests.
- Declared variables in the source program are grouped into storage spaces.
- The storage spaces are determined by static reference analysis (the system can have several distinct memory subsystems).
- The memory subsystems have a relaxed consistency model: all accesses are time-stamped, and accesses to the same memory location are finished in first-come-first-served order.

An example: the dot-product

Consider the program fragment:

```
float A[1024], B[1024];
float dotP = 0.0;
for(I=0; I < 1024; I++) {
   dotP += A[I]*B[I];
}
```

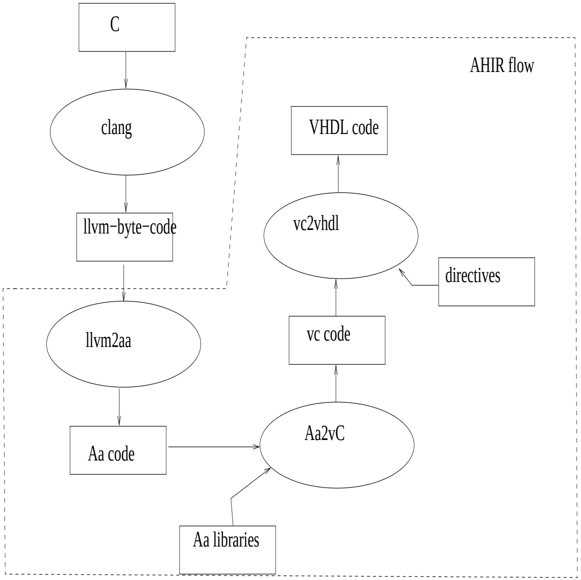
Dot-product circuit



Our C-to-VHDL flow

- The starting point is a program written in C
  - restrictions: no cycles in call graph, and no **function** pointers.
- Use the CLANG front end to convert C program into LLVM byte code (CDFG).
- Map the CDFG to an Ahir assembly program (using the **Aa** language, in which parallelism can be expressed in a native way).
- Map the **Aa** description to a virtual circuit consisting of interacting AHIR modules.
  - identify disjoint memory spaces.
  - dependency analysis to maximize parallelism in straight-line code.
- Map the virtual circuit to VHDL.
  - identify arbiter-less resource sharing opportunities to reduce hardware cost.
  - instantiate the system: modules with their control and data-paths, the inter-module link layer to handle calls, the memory subsystem.

Our C-to-VHDL flow



Mapping to VHDL

- A synchronous, single clock, positive edge-triggered paradigm is used.
- Transitions are coded as pulses which are sampled high by one clock edge.
- Only join transitions need flip-flops for their implementation. Other elements are purely combinational in nature, and in many cases, can be optimized away.
- The datapath is mapped to an equivalent VHDL netlist constructed using a library of operators. When a set of operations shares a single operator, input multiplexors and output demultiplexors are introduced.
- The memory subsystem is implemented using multiple banks, and can offer multiple ports (performance-cost tradeoff).

Experimental Evaluation 1

- Select a range of programs
  - A5 (stream cipher), AES encryption, Red-black Trees (data-structure), Linpack (LU factorization), Fast-Fourier Transform (FFT).
- Using the C-to-VHDL flow, map each program to RTL. The run-time for mapping is neglible in all cases (less than a minute).
- Use standard synthesis tools (Synopsys Design Compiler, Cadence SOC encounter) to implement ASIC from RTL (we use the 180nm TSMC CMOS process, with OSU standard cell libraries). Extract area, delay and energy numbers for the implemented ASIC.
  - Operators are not pipelined or optimized in any fashion.
- Run each program on a processor based platform (we use the Intel Atom N270 as a reference). Extract area, delay, energy numbers using the processor data sheet.
- Compare the ASIC numbers with the processor numbers.

Comparison of normalized (to 45nm) C-to-VHDL circuits with 45nm processor

Table: Area/Delay/Power/Energy RATIOS (processor values relative to the scaled C-to-RTL circuit values)

	Area	Freq	Delay	Power	Energy
A5/1	275.8	5.6	1.7	68.5	116.6
AES	61.5	5.6	0.34	14.8	4.9
FFT	78.4	9.6	0.57	52.6	30.3
LPK	14.8	9.6	0.84	18.3	15.3
RBT	22.2	9.6	0.15	32.7	4.7

Analysis

- The C-to-RTL circuits are between 4.7X and 116X more energy efficient than the processor.
- The processor delays are lower than those of the C-to-RTL circuit in most cases (except for A5) but the difference is less than one order of magnitude in all cases.
  - pipelining of operators and retiming of the C-to-RTL circuits should reduce this gap.

For single-threaded applications, the C-to-RTL circuits have much better energy efficiency than the processor, but have lower performance than the processor.

Update: Loop optimizations seem to be very useful for improving single thread performance (we will talk about this later).

Experimental Evaluation 2: Click2Netfpga

This work is a joint effort with Teemu Rinta Aho and Mika Karlstedt at Ericsson Research, and will be presented at the USENIX ATC 2012 (Boston).

- Click is a modular framework for describing networking devices (e.g. routers, classifiers etc.). A Click description is converted to C++ code, which can then be compiled and executed.
- The Click2Netfpga tool-chain takes Click-generated C++ code and produces VHDL which can be mapped to the Stanford NetFPGA card and validated in real-time.
  - The Click2llvm front-end generates LLVM modules (which form a pipeline) from the C++ code.
  - The AHIR tool flow maps the LLVM modules to a hardware pipeline.
- A router generated in this manner gives up to 50% of the performance of a reference (hand-coded Verilog) design.

Update: Loop optimizations should reduce the gap between hand-designed and compiler generated hardware (will talk about this later).

Trends

- On purely sequential code, the hardware that is generated consumes substantially less energy than a processor running the same code. Performance of single-threaded hardware needs to improve.
- For processing pipelines inferred from application specific C++ code, the performance is upto 50% of hand-coded Verilog. The performance gap needs attention, but 50% of hand-coded is “not bad”.

Offers a viable option to the system designer (in addition to the use of embedded processors and custom hardware).

Optimizations: loop-pipelining

Clearly, there is scope for improvement. The most critical problem in improving single-thread performance is the inner loop.

```
float A[1024], B[1024];
float dotP = 0.0;
for(I=0; I < 1024; I++) {
   dotP += A[I]*B[I];
}
```

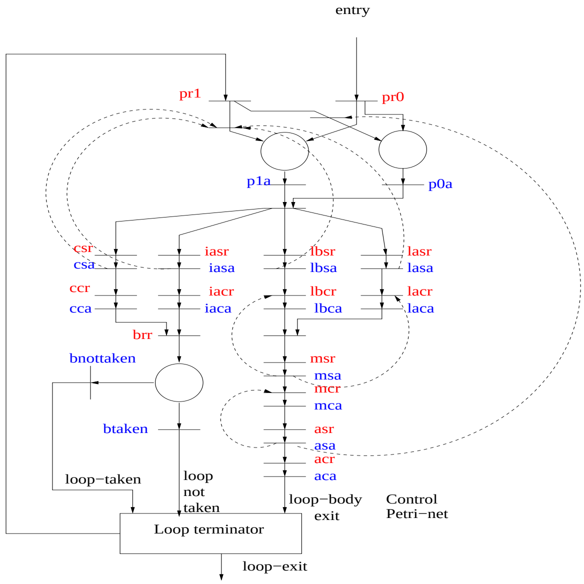
Execute multiple-iterations of a loop whenever possible.

Dynamic loop-pipelining

In an AHIR system, loop-pipelining can be managed if the control-path can maintain dependencies across loops:

- $F_k \rightarrow F_{k+1}$ for all operations F, for all k.
- If G depends on F, then $F_k \rightarrow G_k \rightarrow F_{k+1}$.
- If P, Q are memory operations with an explicit dependency (WAR, RAW, WAW), then $P_k \rightarrow Q_k \rightarrow P_{k+1} \rightarrow Q_{k+1}$.

Modified control-path for dynamic loop-pipelining



Impact of dynamic loop-pipelining on single-threads

We looked at some simple, but important examples:

- Dot-product.
- FFT.
- Matrix-multiply.

In each case, single-threaded performance improvements were measured. Number of FP adders and multipliers used $= 1$ of each.

Impact of loop-optimizations on the 64-point dot-product

For the 64-point dot-product (time/area numbers taken from FPGA logic synthesis targeting a Xilinx Virtex-6):

	plain	pipelined	unrolled	unrolled and pipelined
Cycles	4071	1874	1894	582
LUTs	4288	5195	4809	7344
FFs	3799	4345	4378	5911
Freq.(MHz)	199.7	199.7	199.7	199.7

Best case FLOPS/cycle $= 127/582 = 0.22$.

Impact of loop-optimizations on the 64-point FFT

For a single stage of the 64-point FFT (32 butterflies, 300 FP ops).

	plain	pipelined	unrolled	unrolled and pipelined
Cycles	4151	2885	3110	1064
LUTs	12155	23139	16032	37831
FFs	11955	18632	15299	28698
Freq. (MHz)	186.9	164.1	186.9	164.1

Best case FP-ops/cycle $= 300/1064 = 0.3$.

Impact of loop-optimizations on the 16x16 matrix-multiply

Here, some aggressive loop-unrolling was used.

	plain	pipelined	unrolled	unrolled and pipelined
Cycles	161K	77K	13K	7810
LUTs	6323	9408	16032	31744
FFs	6974	8818	15299	21060
Freq.(MHz)	199.7	199.7	186	164.1

Best case FLOPS/cycle = 7936/7810 =1.01.

Observations

- Excellent performance improvement with loop-pipelining and unrolling.
- Single-threaded hardware performance is pretty OK.
  - BUT: resource utilization levels need to be improved (anything above 50% is reasonable).
  - Main cause is latency in non-parallel code (Amdahl's law). This effect can be reduced by reducing latency in the logic (in progress).
  - As the problem size is increased, utilization levels tend to increase.
- In practice, we will need to use multiple threads to exploit advantages of hardware relative to the processor.

Future directions for our research

- Applications: signal processing (image/multi-media), cryptology, CFD, Databases etc.
- More optimizations: latency reduction in non-parallel paths.
- Algorithm-to-ASIC.
- OpenMP, OpenCL support.

Conclusion

The AHIR flow is open-source. If you wish to try it out, please get in touch with me: madhav@ee.iitb.ac.in
Thank you!