An hardware inspired model for parallel programming

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

What we said in the first lecture

This subject is about

- The foundations of functional languages:
  - the $\lambda$-calculus, types, monads, confluence, operational semantics, TRS...
- General purpose implicit parallel programming in Haskell & pH
- Parallel programming based on atomic actions or transactions in Bluespec
- Dataflow model of computation
  and understanding connections ...

Bluespec and pH borrow heavily from functional languages but their execution models differ completely from each other
pH: Implicit Parallel Programming

pH: parallel Haskell
(Types, Higher-order functions, I-structures, M-structures)

Multithreaded Intermediate Language

Dataflow and multithreaded compilation model

R.S. Nikhil, Arvind &
many brilliant students
@ MIT mid 80's to 90's

Multithreaded C
SMP's
Clusters

We didn’t discuss compilation much!

Fully Parallel, Multithreaded Model

Tree of Activation Frames

Global Heap of Shared Objects

Synchronization?

Efficient mappings on architectures has proved difficult
Instead of focusing on compilation, we will study

- A hardware inspired methodology for “synthesizing” parallel programs
  - Rule-based specification of behavior (Guarded Atomic Actions)
    - Lets you think one rule at a time
  - Composition of modules with guarded interfaces

Bluespec

Example: 802.11a transmitter

Unity – late 80s
Chandy & Misra

Warning: The ideas are untested in the software domain; you are the trailblazers.

Bluespec: State and Rules organized into modules

All state (e.g., Registers, FIFOs, RAMs, ...) is explicit.
Behavior is expressed in terms of atomic actions on the state:

Rule: condition ➔ action

Rules can manipulate state in other modules only via their interfaces.
Programming with rules:
Example Euclid’s GCD

Terms
GCD(x, y), integers

Rewrite rules
\[ \text{GCD}(x, y) \Rightarrow \text{GCD}(y, x) \quad \text{if} \quad x > y, \ y \neq 0 \quad (R_1) \]
\[ \text{GCD}(x, y) \Rightarrow \text{GCD}(x, y-x) \quad \text{if} \quad x \leq y, \ y \neq 0 \quad (R_2) \]

Initial term
GCD(initX, initY)

Execution
\[
\begin{align*}
\text{GCD}(6, 15) & \Rightarrow \text{GCD}(6, 9) \\
& \Rightarrow \text{GCD}(6, 3) \\
& \Rightarrow \text{GCD}(3, 6) \\
& \Rightarrow \text{GCD}(3, 3) \\
& \Rightarrow \text{GCD}(3, 0)
\end{align*}
\]

R2 R2 R1 R2 R2

GCD in Bluespec

```plaintext
class mkGCD (I GCD);

Reg#(int) x <- mkRegU;
Reg#(int) y <- mkReg(0);

rule swap when ((x>y) && (y!=0)) =>
  x <= y; y <= x;
endrule

rule subtract when ((x<=y) && (y!=0)) =>
  y <= y - x;
endrule

method Action start(int a, int b) when (y==0) =>
  x <= a; y <= b;
endmethod

method int result() when (y==0);
  return x;
endmethod
endmodule
```

Assumes x /= 0 and y /= 0
GCD Hardware Module

In a GCD call `t` could be `Int#(32)`, `UInt#(16)`, `Int#(13)`, ...

The module can easily be made polymorphic

Many different implementations can provide the same interface:

```markdown
module mkGCD (I_GCD)
endmodule
```

Bluespec: Two-Level Compilation

- **Bluespec** (Objects, Types, Higher-order functions)
- Lennart Augustsson @Sandburst 2000-2002
  - Type checking
  - Massive partial evaluation and static elaboration

- Level 1 compilation
  - Rules and Actions (Term Rewriting System)
  - James Hoe & Arvind @MIT 1997-2000
  - Rule conflict analysis
  - Rule scheduling

- Object code (Verilog/C)

```

November 2, 2006
http://csg.csail.mit.edu/6.827/"
Static Elaboration

- Inline function calls and datatypes
- Instantiate modules with specific parameters
- Resolve polymorphism/overloading

Software Toolflow:

1. Compile
2. Run w/ params

Hardware Toolflow:

1. Source
2. Elaborate w/ params
3. Design
4. Run w/ params

Expressing designs for 802.11a transmitter in Bluespec (BSV)
802.11a Transmitter Overview

Controller

Scrambler

Encoder

Interleaver

Mapper

IFFT

Cyclic Extend

Must produce one OFDM symbol every 4 μsec

Depending upon the transmission rate, consumes 1, 2 or 4 tokens to produce one OFDM symbol

IFFT Transforms 64 (frequency domain) complex numbers into 64 (time domain) complex numbers

One OFDM symbol (64 Complex Numbers) accounts for > 95% area

Uncoded bits

Preliminary results

<table>
<thead>
<tr>
<th>Design Block</th>
<th>Lines of Code (BSV)</th>
<th>Relative Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controller</td>
<td>49</td>
<td>0%</td>
</tr>
<tr>
<td>Scrambler</td>
<td>40</td>
<td>0%</td>
</tr>
<tr>
<td>Conv. Encoder</td>
<td>113</td>
<td>0%</td>
</tr>
<tr>
<td>Interleaver</td>
<td>76</td>
<td>1%</td>
</tr>
<tr>
<td>Mapper</td>
<td>112</td>
<td>11%</td>
</tr>
<tr>
<td>IFFT</td>
<td>95</td>
<td>85%</td>
</tr>
<tr>
<td>Cyc. Extender</td>
<td>23</td>
<td>3%</td>
</tr>
</tbody>
</table>

Complex arithmetic libraries constitute another 200 lines of code
Combinational IFFT

All numbers are complex and represented as two sixteen bit quantities. Fixed-point arithmetic is used to reduce area, power, ...

Design Alternative

Reuse a block over multiple cycles

we expect:

Throughput to reduce – less parallelism
Energy/unit work to increase - due to extra HW
Area to decrease – reusing a block
Combinational IFFT

Opportunity for reuse

Reuse the same circuit three times

Circular pipeline: Reusing the Pipeline Stage

16 Radix 4s can be shared but not the three permutations. Hence the need for muxes
Superfolded circular pipeline: Just one Radix-4 node!

Which design consumes the least energy to transmit a symbol?

- Can we quickly code up all the alternatives?
  - single source with parameters?

Not practical in traditional hardware description languages like Verilog/VHDL
Bluespec code: Radix-4 Node

```verilog
def function Vector#(4,Complex) m = newVector(),
    Vector#(4,Complex) t, Vector#(4,Complex) k);

Vector#(4,Complex) m = newVector(),
y = newVector(),
z = newVector();
m[0] = k[0] * t[0]; m[1] = k[1] * t[1];
y[0] = m[0] + m[2]; y[1] = m[0] – m[2];
z[0] = y[0] + y[2]; z[1] = y[1] + y[3];
return(z);
endfunction
```

Polymorphic code: works on any type of numbers for which *, + and - have been defined

Combinational IFFT
Can be used as a reference

```
stage_f function
repeat it three times
```

Bluespec Code for Combinational IFFT

```bluespec
function SVector#(64, Complex) ifft (SVector#(64, Complex) in_data);

//Declare vectors
SVector#(4,SVector#(64, Complex)) stage_data = replicate(newSVector);
stage_data[0] = in_data;
for (Integer stage = 0; stage < 3; stage = stage + 1)
    stage_data[i+1] = stage_f(stage, stage_data[i]);
return(stage_data[3]);
```

The code is unfolded to generate a combinational circuit

Bluespec Code for stage_f

```bluespec
function SVector#(64, Complex) stage_f (Bit#(2) stage, SVector#(64, Complex) stage_in);
begin
    for (Integer i = 0; i < 16; i = i + 1)
        begin
            Integer idx = i * 4;
            let twid = getTwiddle(stage, fromInteger(i));
            let y = radix4(twid, stage_in[idx:idx+3]);
            stage_temp[idx] = y[0]; stage_temp[idx+1] = y[1];
            stage_temp[idx+2] = y[2]; stage_temp[idx+3] = y[3];
        end
    //Permutation
    for (Integer i = 0; i < 64; i = i + 1)
        stage_out[i] = stage_temp[permute[i]];
end
return(stage_out);
```

Stage function
Synchronous pipeline

```
rule sync-pipeline (True);
inQ.deq();
sReg1 := f1(inQ.first());
sReg2 := f2(sReg1);
outQ.enq(f3(sReg2));
endrule
```

This is real IFFT code; just replace f1, f2 and f3 with stage_f code

What about pipeline bubbles?

```
typedef union tagged {
  void Invalid;
  data_T Valid;
} Maybe#(type data_T);

rule sync-pipeline (True);
  Maybe#(data_T) sx, ox;
  for (Integer i = 1; i < n; i = i + 1)
    begin
      // Get stage input
      if (i == 0)
        begin
          sx = inQ.first(); inQ.deq();
        end
      else
        sx = sRegs[i-1];
      case (sx)
        tagged Valid .x: ox = f(fromInteger(i), x);
        tagged Invalid: ox = Invalid;
      endcase
    if (i == n-1) outQ.enq(ox); // Write Outputs
    else sRegs[i] <= ox;
  end
endrule
```
Folded pipeline

```
rule folded-pipeline (True);
    if (stage==1)
        begin
            inQ.deq();
            sxIn = inQ.first();
        end
    else
        sxIn = sReg;
    sxOut = f(stage,sxIn);
    if (stage==3) outQ.enq(sxOut);
    else sReg <= sxOut;
    stage <= (stage==3)? 1 : stage+1;
endrule
```

```
function f (stage,sx);
    case (stage)
    1: return f1(sx);
    2: return f2(sx);
    3: return f3(sx);
    endcase
endfunction
```

This is real IFFT code too ...

Expressing these designs in Bluespec is easy

- All these designs were done in less than one day!
- Area and power estimates?

<table>
<thead>
<tr>
<th>Design Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combinational</td>
</tr>
<tr>
<td>Pipelined</td>
</tr>
<tr>
<td>Folded (16 Radices)</td>
</tr>
<tr>
<td>Super-Folded (8 Radices)</td>
</tr>
<tr>
<td>Super-Folded (4 Radices)</td>
</tr>
<tr>
<td>Super-Folded (2 Radices)</td>
</tr>
<tr>
<td>Super-Folded (1 Radix)</td>
</tr>
</tbody>
</table>
802.11a Transmitter Synthesis results

<table>
<thead>
<tr>
<th>IFFT Design</th>
<th>Area (mm²)</th>
<th>Symbol Latency (CLKs)</th>
<th>Throughput Latency (CLks/sym)</th>
<th>Min. Freq Required</th>
<th>Average Power (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipelined</td>
<td>5.25</td>
<td>12</td>
<td>04</td>
<td>1.0 MHz</td>
<td>4.92</td>
</tr>
<tr>
<td>Combinational</td>
<td><strong>4.91</strong></td>
<td>10</td>
<td>04</td>
<td>1.0 MHz</td>
<td><strong>3.99</strong></td>
</tr>
<tr>
<td>Folded (16 Radices)</td>
<td><strong>3.97</strong></td>
<td>12</td>
<td>04</td>
<td>1.0 MHz</td>
<td>7.27</td>
</tr>
<tr>
<td>Super-Folded (8 Radices)</td>
<td>3.69</td>
<td>15</td>
<td>06</td>
<td>1.5 MHz</td>
<td>10.9</td>
</tr>
<tr>
<td>SF(4 Radices)</td>
<td>2.45</td>
<td>21</td>
<td>12</td>
<td>3.0 MHz</td>
<td>14.4</td>
</tr>
<tr>
<td>SF(2 Radices)</td>
<td>1.84</td>
<td>33</td>
<td>24</td>
<td>6.0 MHz</td>
<td>21.1</td>
</tr>
<tr>
<td>SF (1 Radix)</td>
<td>1.52</td>
<td>57</td>
<td>48</td>
<td>12 MHz</td>
<td>34.6</td>
</tr>
</tbody>
</table>

Why are the areas so similar

- Folding should have given a 3x improvement in IFFT area
- BUT a constant twiddle allows low-level optimization on a radix4 block
  - a 2.5x area reduction!
802.11a Observation

- Dataflow network
  - aka Kahn networks
- How should this level of concurrency be expressed in a reference code (say in C or systemC)?
- Can we write Specs which work for both hardware and software

Bluespec Tool flow

Bluespec System Verilog source

Bluespec Compiler

C

Verilog 95 RTL

Verilog sim

RTL synthesis

VCD output

Gates

Debussy Visualization

Power estimation tool

Place & Route

Tapeout

FPGA