Exposing ILP in Custom Hardware with a Dataflow Compiler IR

Ali Mustafa Zaidi

Supervisor: Dr. David Greaves

University of Cambridge
Computer Laboratory
The Dark Silicon Problem

Amdahl's Law

\[ Perf = \frac{1}{(1 - f)} \frac{f}{s_{seq}} n \]

Utilization Wall

\[ \frac{Perf(s_{seq} = 3)}{Perf(s_{seq} = 1)} \]

\[ \begin{array}{c}
\text{2.1GHz @ 90nm (80W) } \\
18\% \\
\text{5.2GHz @ 45nm (80W) } \\
7\% \\
\text{7.3GHz @ 32nm (80W) } \\
3\%
\end{array} \]

45nm → 8nm (32x resources)

- CPU: 3.5x, GPU 2.4x (Cnsrv.)
- CPU: 7.9x, GPU 2.7x (ITRS)

The Dark Silicon Problem

Amdahl's Law

Utilization Wall

Dark Silicon

Can we achieve Superscalar Performance, w/o Superscalar Overheads?

Solution: Spatial Architectures?

- Custom Hardware, FPGAs, CGRAs, MPPAs, etc.

**Advantages**
- Scalable, Decentralized architectures, with short, p2p wiring.
- High Computational Density
- 10-1000x Energy and Performance efficiency.

**Issues**
- Poor Programmability: often requiring low-level hardware knowledge
- Limited Amenability: poor performance on sequential, irregular, or complex control-flow code.

**Examples**
- Conservation Cores: Performance ≈ in-order MIPS24KE core
- Phoenix CASH Hardware: Performance 30% less than 4-way OOO Core.
Solution: Spatial Architectures?

Key Reasons for High Performance of Complex, OOO Superscalars:

- Aggressive Control-flow Speculation
- Dynamic, out-of-order execution scheduling

Custom hardware has very limited speculation
- Single flow of control
- If-conversion & hyperblock formation for forward branches.
- **No acceleration of backwards branches!**

Control-Data Flow Graph

```
for (i = 0; i < 100; i++)
    if (A[i] > 0) foo();
    bar();
```

McFarlin et al., “Discerning the dominant out-of-order performance advantage: is it speculation or dynamism?”, ASPLOS ’13
Solution: Spatial Architectures!

Control-Data Flow Graph

for (i = 0; i < 100; i++)
  if (A[i] > 0) foo();
  bar();

Our Solution

Instead of

CDFG IR + Compile-time Execution Scheduling

We Employ

VSFG IR + Dataflow Execution Model
Overcoming Control-Flow with the VSFG

- **Hierarchical Dataflow Graph**
  - Instead of [Basic Blocks + Control Flow], we have [Nested Subgraphs + Dataflow]
  - Functions → nested subgraphs
  - Loops → tail-recursive functions.

- **Dataflow execution of operations**
  - Multiple Subgraphs may execute concurrently in Dataflow order (unlike basic blocks).
  - Exposes **Multiple Flows of Control**!
Overcoming Control-Flow with the VSFG

for (i = 0; i < 100; i++)
    if (A[i] > 0) foo();
    bar();

• Infinite DAG
  – Loops represented as Tail Recursion
  – Branches represented via if-conversion
  – Enables **Aggressive Speculation**!

• No single 'Flow of Control'
  – Instead, control implemented via 'Boolean Predicate Expressions'.
  – Logic minimization can simplify expressions, facilitating **Control Dependence Analysis**!
Overcoming Control-Flow with the VSFG

Hierarchical Dataflow Graph
- Subgraphs may be 'predicated', or executed speculatively (via 'if-conversion').
- 'Flattening' loop tail-call subgraphs → loop unrolling/pipelining.
- Multiple loops in a loop-nest may be unrolled independently to expose ILP
Overcoming Control-Flow with the VSFG
Overcoming Control-Flow with the VSFG
High Level Synthesis Case Study

Any High Level Language → LLVM → VSFG → Low-Level IR → Bluespec SystemVerilog → ASIC / FPGA

%1 = mul i32 %x, %y;
%2 = srem i32 %1, %z;
%3 = icmp slt i32 %2, %1;

FIFOF(int) x ← mkFIFO1;
FIFOF(int) y ← mkFIFO1;
FIFOF(int) z ← mkFIFO1;
FIFOF(int) srem_1 ← mkFIFO1;
FIFOF(int) icmp_1 ← mkFIFO1;
FIFOF(int) icmp_2 ← mkFIFO1;
FIFOF(int) out_3 ← mkFIFO1;

rule mul_inst;
let val1 = x.first; x.deq;
let val2 = y.first; y.deq;
let rslt = val1 * val2;
srem_1.enq (rslt);
icmp_1.enq (rslt);
endrule

rule srem_inst;
let val1 = srem_1.first; srem_1.deq;
let val2 = z.first; z.deq;
let rslt = val1 % val2;
icmp_2.enq (rslt);
endrule
Hardware Oriented Dataflow IR

• Performance and Energy Evaluation by comparing with
  - LegUp HLS Tool, & Altera Nios IIf Processor, implemented on Altera Stratix IV GX FPGA.
  - Nehalem Core i7 (Sniper interval simulator from Intel).
  - In all cases, memory access latency assumed == 1 Cycle.

• LegUp
  - LLVM 2.9
  - O2
  - No LTO, no LTI
  - No Op Chaining
  - Statically Scheduled CFG

• Our Toolchain
  - LLVM 2.6
  - O2
  - No LTO, no LTI
  - No Op Chaining
  - Dynamically Scheduled VSFG
Performance (Cycle Counts)

Matrix Transpose (x1k cycles)

adpcm (x1k cycles)

dfs in (x1k cycles)

Neural Net Simulator (x1M cycles)

Compared to Nios II/f & Intel Nehalem Core i7 (SniperSim)
Frequency & Delay

Frequency (Higher is Better)

Normalized Delay (Lower is Better)

Nios II f @ 250MHz
Power and Speculation Overheads

Power estimation assuming 250MHz operating frequency
Power and Speculation Overheads

Power estimation assuming 250MHz operating frequency
Normalized Energy

![Normalized Energy Chart]

Legend:
- **LegUp**
- **VSFG_0**
- **VSFG_1**
- **VSFG_3**
- **Nios**
Sources of Energy Inefficiency

- Energy Cost Comparison:
  - vs Nios II/f: **0.25 x** (GEOMEAN)
  - vs LegUp: **3-4 x** (GEOMEAN)

- Overheads of Speculation
  - Balance between speculation & predication must be found for efficiency & performance

- Part of power dissipation proportional to Area
  - Clock Gating for predicated regions to reduce dynamic power
    - (consider asynchronous Ckts)
  - Power gating for predicated regions to reduce static power?
  - Selective loop unrolling.
Limitations on Performance

• 35% better performance than statically scheduled CFG, without any optimizations:
  – Improvements due to dynamic scheduling, MFC & CDA
  – Unrolling helps, but speed-up saturates quickly.

• Further Improvements possible:
  – Balance between **predication** & **speculation**, to improve speed-up without unrolling (thus reducing area and energy costs)
  – State-edge is on critical path – limits both unrolling & MFC.
    • Last remnant of 'sequential' nature of program.

• Frequency Scaling limited by Memory Interconnect
  – Partition memory & pipeline memory access tree
Thank You
Implicit Parallelism & State-edge Partitioning

Assertion: Implicit (deterministic) parallel programming models are essentially means of partitioning the state-edge.
Overcoming Control-Flow with the VSFG

**Control-Data Flow Graph**

```
for (i = 0; i < 100; i++)
  if (A[i] > 0) foo();
  bar();
```

**Value State Flow Graph**

```
Start

i = 0

A[i] = A[i]

> 0

i++

< 100

T

F

bar()

End

Next iteration of 'for' loop

STATE_IN

STATE_OUT

bar()
Performance (Cycle Counts)

- Cycle counts normalized to LegUp results
- VSFG implemented with all loops unrolled 0, 1, and 3 times
- Full Speculation: all subgraphs (except loops) triggered without predicates
Performance (Cycle Counts)

Cycle Counts with Full Speculation

Predication: only one block will execute

Speculation: both blocks execute, but only one result is chosen
Performance (Cycle Counts)

Cycle Counts with Full Speculation

Predicates

Split

T

F

64

2

Merge

InGate

64

OutGate

2

Mux

Predicate

T

F

64

2

Mux

Predicate

T

F
Performance (Cycle Counts)

Cycle Counts with Full Speculation

Cycle Counts with Predicated Subgraphs
Performance (Cycle Counts)

- **dfadd**
- **dfdiv**
- **dfmul**
- **mips**

Bar charts comparing the performance of different processors and software tools for arithmetic operations.
Understanding OOO Performance

Control-Data Flow Graph

- Control flow is the primary constraint on ILP
  - Wall (1991): Conventional processors limited to ILP of 4-8!
    - Single Flow of control
    - Branch prediction (+95% accuracy)
  - Lam & Wilson (1993), Mak & Mycroft (2009): 10x ILP possible, with:
    - Control Dependence Analysis (CDA)
    - Multiple Flows of Control (MFC)

- Custom hardware has very limited speculation
  - Single flow of control
  - If-conversion & hyperblock formation for forward branches.
  - No acceleration of backwards branches!

```plaintext
for (i = 0; i < 100; i++)
  if (A[i] > 0) foo();
  bar();
```
Formalizing & Evaluating the VSFG

- Plotkin-style operational semantics developed for VSFG
  - Assuming Static Dataflow execution model

- Low-Level IR developed to facilitate conversion to Bluespec
  - Based on Hierarchical Coloured Petri-nets

- High-Level Synthesis Toolchain implemented
Hardware Oriented Dataflow IR

%1 = mul i32 %x, %y      ; <i32>
%2 = srem i32 %1, %z      ; <i32>
%3 = icmp slt i32 %2, %1  ; <i1>

Value-State Flow Graph

Petri Net based Low Level Dataflow IR

LLVM IR

→ Registers
→ Instructions
→ Petri Net Places
→ Petri Net Transitions
Hardware Oriented Dataflow IR

%1 = mul i32 %x, %y ; <i32>
%2 = srem i32 %1, %z ; <i32>
%3 = icmp slt i32 %2, %1 ; <i1>

Petri Net Places
→
Petri Net Transitions

Petri Net based
Low Level
Dataflow IR

LLVM IR

FIFOF(int) x ← mkFIFOF1;
FIFOF(int) y ← mkFIFOF1;
FIFOF(int) z ← mkFIFOF1;
FIFOF(int) srem_1 ← mkFIFOF1;
FIFOF(int) icmp_1 ← mkFIFOF1;
FIFOF(int) icmp_2 ← mkFIFOF1;
FIFOF(int) out_3 ← mkFIFOF1;

rule mul_inst;
let val1 = x.first; x.deq;
let val2 = y.first; y.deq;
let rslt = val1 * val2;
srem_1.enq (rslt);
icmp_1.enq (rslt);
endrule

rule srem_inst;
let val1 = srem_1.first; srem_1.deq;
let val2 = z.first; z.deq;
let rslt = val1 % val2;
icmp_2.enq (rslt);
endrule

Equivalent Bluespec Code