A New Dataflow Compiler IR for Accelerating Control-Intensive Code in Spatial Hardware

Ali Mustafa Zaidi, David Greaves
{ali-mustafa.zaidi, david.greaves}@cl.cam.ac.uk

University of Cambridge
Computer Laboratory
The Dark Silicon Problem

Amdahl's Law

Utilization Wall

= Dark Silicon

\[ \text{Perf} = \frac{1}{\left(1 - f\right)} \cdot \frac{s_{seq}}{f} \]

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

Need for both high 'Sequential' Performance, AND Very High Energy Efficiency

45nm → 8nm (32x resources)
- CPU: 3.5x, GPU 2.4x (Cnsrv.)
- CPU: 7.9x, GPU 2.7x (ITRS)

Amdahl's Law

\[ \text{Utilization Wall} \]

\[ \text{Dark Silicon} \]

Superscalar Processors
- Only Option for Seq. Performance!
- Power scales exponentially with Performance

Custom Hardware:
- 10 – 1000 x Efficiency!
- Not for Sequential!

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 efficiency & Performance.

**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.
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!**
Our Solution

Instead of
CDFG IR + Compile-time Execution Scheduling

We Employ
VSFG IR + Dataflow Execution Model
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!
VSFG: Value-State Flow Graph

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**!
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.

```
for (i = 0; i < 100; i++)
    if (A[i] > 0) foo();
bar();
```
VSFG: Value-State Flow Graph
High Level Synthesis Case Study

Any High Level Language $\rightarrow$ LLVM $\rightarrow$ VSFG $\rightarrow$ Low-Level IR $\rightarrow$ Bluespec SystemVerilog $\rightarrow$ ASIC / FPGA

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

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
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 based
Low Level
Dataflow IR

Value-State
Flow Graph

Petri Net based
Low Level
Dataflow IR

LLVM IR

→ Registers
→ Instructions
→ Petri Net Places
→ Petri Net Transitions
%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

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
High Level Synthesis Case Study

- 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)

Normalised to LegUp

Matrix Transpose (x1k cycles)
- epic*: 0.99
- adpcm: 0.81
- dfadd: 0.68
- dfdiv: 0.87
- dfmul: 0.97
- dfsin: 1.07

Adpcm (x1k cycles)
- mips: 0.80
- bimpa: 0.68
- GEOMEAN: 0.88

Dfsin (x1k cycles)
- LegUp (CDFG): 0.81
- VSFG_0: 0.72
- VSFG_1: 0.66
- VSFG_3: 0.66

Neural Net Simulator (x1M cycles)
- LegUp (CDFG): 1.07
- VSFG_0: 0.97
- VSFG_1: 0.97
- VSFG_3: 0.97

Compared to Nios II/f & Intel Nehalem Core i7 (SniperSim)
Power & Speculation Overheads

Power estimation assuming 250MHz operating frequency
Power estimation assuming 250MHz operating frequency
Normalized Energy

![Bar chart showing normalized energy for various benchmarks and architectures.](image-url)
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.
Current Performance Limitations

- 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
Implicit Parallelism & State-edge Partitioning

**Assertion**: Implicit (deterministic) parallel programming models are essentially means of partitioning the state-edge.

---

**Increasing Programmer / Compiler Effort**

- OpenMP
- OpenCL
- Sieve C++

**Alias Analysis**

- Specul. Loads

**Increasing Runtime Effort**

- Dynamic OOO LSQ
- SpMT / TLS
Thank you for listening!

Questions &/or Comments?
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

i = 0

A

i

= A[i]

> 0

T

foo()

i++

< 100

T

F

bar()

End

STATE_IN

STATE_OUT

inPred

i = 0

A

= A[i]

> 0

foo()

i++

< 100

Next iteration of 'for' loop

STATE_OUT
- 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

- epic
- adpcm
- dfadd
- dfdiv
- dfmul
- dfsin
- mips**
- small_bimpa

Legend:
- LegUp (CFG)
- VSFG_0
- VSFG_1
- VSFG_3

Predicate

Split

Predicate

InGate

OutGate

Mux
Performance (Cycle Counts)

Cycle Counts with Full Speculation

Cycle Counts with Predicated Subgraphs
Performance (Cycle Counts)

- Core i7
- Nios 2f
- LegUp
- VSFG_0
- VSFG_1
- VSFG_3

**Graphs:**

1. **epic**
   - Core i7: 39664956
   - Nios 2f: 373347552
   - LegUp: 142386696
   - VSFG_0: 114361494
   - VSFG_1: 98179648
   - VSFG_3: 97430648

2. **adpcm**
   - Core i7: 42662
   - Nios 2f: 119794
   - LegUp: 71349
   - VSFG_0: 57860
   - VSFG_1: 51580
   - VSFG_3: 51186

3. **dfsln**
   - Core i7: 104953
   - Nios 2f: 1420558
   - LegUp: 105773
   - VSFG_0: 72007
   - VSFG_1: 71896
   - VSFG_3: 71896

4. **small_bimpa**
   - Core i7: 39664956
   - Nios 2f: 373347552
   - LegUp: 114361494
   - VSFG_0: 98179648
   - VSFG_1: 97430648
   - VSFG_3: 97430648
Understanding OOO Performance

Control-Data Flow Graph

$\text{for } (i = 0; i < 100; i++)$
\[
\text{if } (A[i] > 0) \text{ foo()};
\]
\[
\text{bar()};
\]

- 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!