Achieving Superscalar Performance without Superscalar Overheads

Ali Mustafa Zaidi
Supervisor: Dr. David Greaves

Computer Laboratory
University of Cambridge
The Need for Energy Efficiency

Embedded devices: growing performance requirements, & limited power budgets

Datacenters: Significant proportion of costs due to computation and cooling power supply

- 2.1GHz @ 90nm (80W): 18%
- 5.2GHz @ 45nm (80W): 7%
- 7.3GHz @ 32nm (80W): 3%

Dark Silicon: with the end of threshold voltage scaling, only a small fraction of on-chip resources may be active at any time.

Urgent need to make the most efficient use of transistors possible.
The Need for Energy Efficiency

• How do we improve energy efficiency?

• Go Parallel?
  – Use multiple, simpler cores instead of a single large core
  – Can we achieve same or better performance at lower power?

• Increase Specialization?
  – Customize computational resources to reduce energy cost, while maintaining/improving performance
The Need for Sequential Performance

- Go Parallel?
  - Use multiple, simpler cores for performance at lower power

Amdahl's Law

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

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

![Graph showing Amdahl's Law with different values of f]
The Need for Sequential Performance

- Go Parallel?
  - Use multiple, simpler cores for performance at lower power

Consumer Workloads

Exhibit very low threaded parallelism, unless signal processing is involved.

The Need for Sequential Performance

- **Go Parallel?**
  - Use multiple, simpler cores for performance at lower power

**Datacenter Workloads**

- “Brawny cores still beat wimpy cores, most of the time”, Urs Hölzle, SVP of Operations & Google Fellow, Google Inc.
- Even when “f ≈ 1”, it is better to have fewer faster cores than many slower cores. Because of...
  - Cost of threaded programming
  - Increased overheads of inter-thread synchronization
  - Non-core cost scaling not linear (both system performance and $)
The Need for Energy Efficiency

• Increase Specialization?
  - Customize computational resources to reduce energy cost, while maintaining/improving performance
The Need for Sequential Performance

• Increase Specialization?
  – Customize computational resources to reduce energy cost, while maintaining/improving performance

Amenability

• Only suitable for certain applications
  – Good for: data intensive, regular control-flow, streaming applications
  – Bad for: complex control-flow, irregular, sequential, (high 'functional-density' but low 'data-intensity')
The Need for Sequential Performance

• Increase Specialization?
  – Customize computational resources to reduce energy cost, while maintaining/improving performance

Productivity

• Implementing custom or reconfigurable hardware is
  – Expensive &
  – Time consuming
  – Therefore only used for important kernels that see justifiable performance/energy improvements (e.g. multimedia codecs, cryptography)
Goals

• Solve for the “3 Ps”:
  – Performance, Programmability, and Power * Time (Energy)
  – How do we improve sequential performance of custom hardware without compromising either energy or programmability.
Goals

• Solve for the “3 Ps”:
  – Performance, Programmability, and Power * Time (Energy)

• Holistic Approach:
  – Understand the prerequisites for sequential performance,
  – Understand the hardware costs involved.
  – Figure out how to minimize them.

• Case Study: Generate Custom Hardware directly from HLL code, without compromising performance
Superscalar Performance

- BP for control dependences
- RR for false dependences
- Bypass and Forwarding + OOO dispatch for true dependences
- Dynamic execution scheduling also helps with runtime variability (cache misses, data dependent latency ops etc.)

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

Instruction Store

Branch Predict

Rename & Dispatch

PE

CRF 128 Regs

ROB
Superscalar Scalability

\[ \text{Power} = \text{Perf}^a \quad \text{where } a = 1.75 \]

Superscalar Scalability

• Additional issues: larger centralized resources limit frequency.

• Partitioning infrastructure incurs additional complexity

• Run-time partitioning is expensive, so must be exposed to compiler or programmer (e.g. Clustered register files in Alpha 21264).

Observations – Power Issues

• Instruction Stream management consumes majority of power
  – Offload as much effort as possible to compiler.
  – Expose physical registers in ISA: Instead of Register renaming at run time, compiler manages a larger set of registers.
Continuing Work

- Instead of Renaming H/W Compiler manages, a larger pool of registers, by approximating SSA.
- Increased Power due to larger CRF and Instructions
- Decreased Power as no Register Renaming or ROB required.
Observations – Power Issues

• Operand broadcast and forwarding consumes most remaining power, and limits scalability.
  – Decentralize resources:
    • Instead of centralized state emphasize simpler distributed state,
    • Instead of broadcast buses, emphasize point-to-point communication.
  – Static allocation and binding of operations to PEs by compiler can remove the need for OOO Dispatch and broadcast, allowing the compiler to optimise for locality in this P2P network.
Continuing Work

- Instead of Dynamic Instruction placement and operand broadcast, compiler-based static placement of instructions and data
- Decreased Power due to simpler components and P2P communication.
Continuing Work

When Number of instructions == Number of PEs, we approximate Reconfigurable Hardware (CGRA).

- Instead of Dynamic Instruction placement and operand broadcast, compiler-based static placement of instructions and data
- Decreased Power due to simpler components and P2P communication.
Observations – Performance Issues

- Decentralization improves scalability and (possibly) energy efficiency.

- But now, how do we maintain a shared sequential program state across distributed resources?
  - Sequential ordering of side effects (Memory Ops) must be preserved.
  - Single flow of control must be enforced across distributed resources.
Observations – Performance Issues

- Custom Hardware
  - No false dependences.
  - Accelerates true dependences: p2p communication of intermediate values
  - Statically scheduled: FSM orchestrates distributed operation based on compile time information only.

  - similar performance issues as VLIW processors (good on regular code with compile-time predictable behavior)

- DOES NOT overcome control dependences – waits for predicates

Observations – Performance Issues

- Dataflow Custom Hardware (CMU Pegasus Project)
  - Compilation from HLL to Dynamically Scheduled (Dataflow) Hardware
  - Fully decentralized state machine leads to more scalable hardware!
  - Side-effect ordering represented as dataflow.
  - STILL DOES NOT overcome control dependences – mandatory synchronization on control-flow predicates.

Control flow is the primary constraint on ILP

Conventional processors use aggressive branch prediction
- 95+% accuracy
- Single flow of control

Custom hardware has very limited speculation
- No Branch prediction
- Single flow of control

for (i = 0; i < 100; i++)
if (A[i] > 0) foo();
bar();
Overcoming Control Flow with the VSFG

Control-Data Flow Graph

Start

i = 0

A

i

= A[i]

> 0

T

foo()

i++

< 100

F

bar()

End

Value State Flow Graph

i = 0

A

STATE_IN

= A[i]

> 0

foo()

i++

< 100

Next iteration of 'for' loop

STATE_OUT

bar()
Overcoming Control Flow with the VSFG

- Data-dependences are explicit
- Control-flow is implicit
- Speculation explicit in IR
- Multiple Flows of Control Achievable!

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

- LegUp implements a Statically Scheduled CDFG
- Our tool implements a Dynamically Scheduled VSFG
- Our Tool Permits arbitrary loop unrolling.
Dataflow Pipeline Balancing

- Predication vs Speculation

Predication: only one block will execute

Speculation: both blocks execute, but only one result is chosen
Performance (with Predicated Subgraphs)

- After predicking long-latency operations worst case performance improves:
Clock Frequency

- Achievable frequencies similar to LegUp
  - Limited by last centralized resource: memory access controller.

![](frequency_graph.png)

![](delay_graph.png)
Overheads due to Speculation

- The VSFG allows for aggressive speculation, which can waste energy.
- Speculation may be controlled by predication of sub-graphs.
Normalized Power

- Normalized Power dissipation for all benchmarks matches the overheads of speculation.
- Clock distribution is major fraction of Dynamic Power - dissipation increases with degree of unrolling
- Results suggest need for judicious balancing between Speculation and Predication for best Power / Performance results.
Normalized Energy

- Even with worst case speculation & unrolling overhead
  - Similar or lower energy than an optimised simple in order processor,
Normalized Energy

• Even with worst case speculation & unrolling overhead
  – Similar or lower energy than an optimised simple in order processor,
  – At much higher performance!
Performance vs Conventional Processors

**Graph 1:**
- **4-OOO:** 200174
- **InOrder:** 3399634
- **LegUp:** 1078444
- **VSFG_0:** 1062436
- **VSFG_1:** 528218
- **VSFG_3:** 265170

**Legend:**
- **epic**

**Graph 2:**
- **4-OOO:** 42662
- **InOrder:** 119794
- **LegUp:** 71349
- **VSFG_0:** 57860
- **VSFG_1:** 51580
- **VSFG_3:** 51186

**Legend:**
- **adpcm**

**Graph 3:**
- **4-OOO:** 104953
- **InOrder:** 1420558
- **LegUp:** 105773
- **VSFG_0:** 72007
- **VSFG_1:** 71896
- **VSFG_3:** 71896

**Legend:**
- **dfsln**

**Graph 4:**
- **4-OOO:** 39664956
- **InOrder:** 373347552
- **LegUp:** 142386696
- **VSFG_0:** 114361494
- **VSFG_1:** 98179648
- **VSFG_3:** 97430648

**Legend:**
- **small_bimpa**
Performance vs Conventional Processors

- **dfadd**
  - 4-OOO
  - InOrder
  - LegUp
  - VSFG_0
  - VSFG_1
  - VSFG_3

- **dfdiv**
  - 4-OOO
  - InOrder
  - LegUp
  - VSFG_0
  - VSFG_1
  - VSFG_3

- **dfmul**
  - 4-OOO
  - InOrder
  - LegUp
  - VSFG_0
  - VSFG_1
  - VSFG_3

- **mips**
  - 4-OOO
  - InOrder
  - LegUp
  - VSFG_0
  - VSFG_1
  - VSFG_3
Continuing Work

• Handling state-edge partitioning
  – Spectrum of options
  – Holistic approach: consider custom memory-hierarchy design and memory partitioning problem together.

• Analytical Model to evaluate programmable / reconfigurable architecture design space.
  – Guiding principles:
    • How much effort can be offloaded to the compiler without compromising run-time performance
    • What configuration of components is best suited for various application types, (represented using the VSFG)
Future Directions

• Incorporate parallel programming models
  – GCC Vector Extensions, OpenCL, OpenMP
  – Pthreads?, MPI?

• Evaluate as Coarse-grained Dynamic Dataflow
  – Full support for recursion, real function calls, etc.

• Design-space exploration of Programmable / Reconfigurable tiled architectures
  – Can we replace conventional superscalars with an EDGE architecture
End
Overcoming Control Flow with the VSFG

No Unrolling

/*

In-place (integer) matrix transpose algorithm.

internal_int_transpose(mat, rows, cols)
  register int *mat;
  register int cols;
  int rows;
  {
    register int modulus = rows*cols - 1;
    register int swap_pos, current_pos, swap_val;
    
    /* loop, ignoring first and last elements */
    for (current_pos=1; current_pos<modulus; current_pos++)
      {
        /* Compute swap position */
        swap_pos = current_pos;
        do
          {
            swap_pos = (swap_pos * cols) % modulus;
          }
        while (swap_pos < current_pos);
        
        if (current_pos != swap_pos)
          {
            swap_val = mat[swap_pos];
            mat[swap_pos] = mat[current_pos];
            mat[current_pos] = swap_val;
          }
      }
*/
Unrolled Once

/*
   In-place (integer) matrix transpose algorithm.
*/
internal_int_transpose(mat, rows, cols)
  register int *mat;
  register int cols;
  int rows;
  register int modulus = rows*cols - 1;
  register int swap_pos, current_pos, swap_val;

/* loop, ignoring first and last elements */
for (current_pos=1; current_pos<modulus; current_pos++)
{
   /* Compute swap position */
   swap_pos = current_pos;
   do
   {
      swap_pos = (swap_pos * cols) % modulus;
   } while (swap_pos < current_pos);

   if ( current_pos != swap_pos )
   {
      swap_val = mat[swap_pos];
      mat[swap_pos] = mat[current_pos];
      mat[current_pos] = swap_val;
   }
}
Overcoming Control Flow with the VSFG

• Notice multiple copies of inner loop may be active
  – “Multiple Flows of control”
  – CDFG must wait for predicate at each block boundary
• OOO SS approximates this only when precise branch prediction is available
  – But restricted to a single flow of control
• VSFG enables much more effective control-dependence analysis & optimization
• Notice single 'state-edge', sequentializing side-effects
  – Last sign of 'sequential' nature of program,
  – Final performance bottleneck of the Von Neumann model
  – Parallelization can be seen as partitioning operations on the state edge chain: converting the total-order to a partial order.