A Wire-Delay Scalable Microprocessor Architecture for High Performance Systems

Stephen W. Keckler¹, Doug Burger¹, Charles R. Moore¹, Ramadass Nagarajan¹, Karthikeyan Sankaralingam¹, Vikas Agarwal², M.S. Hrishikesh², Nitya Ranganathan¹, and Premkishore Shivakumar²

Computer Architecture and Technology Laboratory
¹Department of Computer Sciences
²Department of Electrical and Computer Engineering
The University of Texas at Austin
Outline

• Progress and Limitations of Conventional Superscalar Designs

• Grid Processor Architecture (GPA) Overview
  – Block Compilation
  – Block Execution Flow
  – Results

• Extending the GPA with *Polymorphism*

• Conclusion and Future Work
Superscalar Core – “Spot the ALU”

Only 12% of Non-Cache, Non-TLB Core Area is Execution Units
Looking Back: Conventional Superscalar

• Enormous gains in frequency
  • 1998: 500MHz → 2002: 3000MHz
  • Equal contributions from pipelining and technology

• IPC Basically Unchanged
  • 1998: ~1 IPC → 2002: ~1 IPC
  • uArch innovations just overcome losses due to pipelining
  • Issue width remains at 4 instructions

• Pushing the limits of Complexity Management
  • uArch innovations → Verification is the Gate
  • Hundreds of full custom macros
  • 250-500 person design teams
  • Execution units are a small % of processor area
Faster, Higher IPC Superscalar Processors?

**Faster** → *Deeper Pipelines (8 FO4)*

- Key latencies increase … IPC decreases
  - Pipeline bubbles
  - uArch innovations mitigate losses, but ...
    » Increases complexity and performance anomalies

- After 8 FO4 jump, frequency growth limited to technology only

**Higher IPC** → *Wide Issue (16) and Large Window (512+)*

- Growth is *quadratic* but gain is *logarithmic*
  - Broadcast results to all pending instructions
  - Studies indicate only incremental performance gains

- Wire delay limits size of monolithic structures
  - Large structures must be partitioned to meet cycle time
  - Key latencies increase, reducing IPC gain *(again!)*
  - Additional logical and circuit complexity
Superscalar Cores – *Key Circuit Elements*

<table>
<thead>
<tr>
<th></th>
<th>Conventional 4-Issue</th>
<th>Hypothetical 16-issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution</td>
<td>2 FP, 2 INT, 2 LD/ST</td>
<td>8 FP, 8 INT, 8 LD/ST</td>
</tr>
<tr>
<td>I-Cache</td>
<td>64KB 1 Port, 64B (1 instance)</td>
<td>128KB 2 Ports, 128B (1)</td>
</tr>
<tr>
<td>Mapper</td>
<td>8 port x 72-entry CAM (2)</td>
<td>32 port x 512-entry CAM (2)</td>
</tr>
<tr>
<td>Issue Queue</td>
<td>4P x 20-entry dual CAM (3)</td>
<td>4P x 40-entry dual CAM (12)</td>
</tr>
<tr>
<td>RegFiles</td>
<td>72-entry, 4R, 5W ports (4)</td>
<td>512-entry, 4R, 18W ports (8)</td>
</tr>
<tr>
<td>D-Cache</td>
<td>32KB 2R/1W ports (1)</td>
<td>128KB 8R/4W ports (1)</td>
</tr>
</tbody>
</table>

… and pipeline these to use only 8 FO4 delays / cycle!
What is Going Wrong?

1. **Superscalar MicroArchitecture: Scalability is Limited**
   - Relies on large, centralized structures that want to grow larger
   - Partitioning is a slippery slope: *Complexity, IPC loss…*

2. **Architecture: Conventional Binary Interface is outdated!**
   - Linear sequence of instructions
   - Defined for simple, single-issue machines
   - **Not natural for compiler .....**
     - Internally builds and optimizes 2D Control Flow Graph
     - Forced to map CFG into 1D linear sequence
     - Lots of useful information gets thrown away
   - **Not natural for instruction parallel machines .....**
     - Instruction relationships scattered throughout linear sequence
     - Dynamically re-establish by scanning linear sequence
     - $N^2$ problem $\rightarrow$ large, centralized structures
Grid Processor Overview

- Wire-delay constraints exposed at the *architecture* level
- Renegotiate the Compiler / HW Binary Interface
GPA Execution Model

• Compiler structures program into sequence of *hyperblocks*
  – Atomic unit of fetch / schedule / execute / commit

• Blocks specify *explicit instruction placement* in the GRID
  – Critical path placed to minimize communication delays
  – Less critical instructions placed in remaining positions

• Instructions specify consumers as *explicit targets*
  – CFG cast into instruction encoding \(\rightarrow\) *no HW dependency analysis!*
  – Point-to-point results forwarding \(\rightarrow\) *no associative issue queues!*
  – In-GRID storage expands register space \(\rightarrow\) *no global bypass network!*
  – Only block outputs written back to RF \(\rightarrow\) *Fewer RF ports needed!*

• Dynamic Instruction Issue
  – GRID forms large distributed window with independent issue controls
  – Instructions execute in original *dataflow-order*
Block Compilation

Intermediate Code

i1) add r1, r2, r3
i2) add r7, r2, r1
i3) ld r4, (r1)
i4) add r5, r4, 1
i5) beqz r5, 0xdeac

Data Flow Graph

Inputs (r2, r3)

Temporaries (r1, r4, r5)

Outputs (r7)

Compiler Transforms

move r2, i1, i2
move r3, i1

GPA Code
Block Compilation (cont)

Intermediate Code → Data Flow Graph → Mapping → GPA Code

Data flow graph

move r2, i1, i2
move r3, i1

move r2, i1, i2
move r3, i1

Mapping onto GPA

move r2, (1,3), (2,2)
moved r3, (1,3)

Scheduler
Block Compilation (cont)

Intermediate Code → Data Flow Graph → Mapping → GPA Code

Mapping onto GPA

move r2, (1,3), (2,2)
move r3, (1,3)

GPA code

I1) : (1,3) add (2,2) (2,3)

Code generation

Targets

Opcode

Instruction location
Block Execution

- Instruction distribution
- Input register fetch
- Block execution
- Output register writeback

ICache moves

ICache bank 0

ICache bank 1

ICache bank 2

ICache bank 3

Load store queues

DCache bank 0

DCache bank 1

DCache bank 2

DCache bank 3

Block termination Logic

r2

r3

add

beqz

add

add

Load store queues
Instruction Buffers - *frames*

- Instruction Buffers add *depth* and define *frames*
  - 2D GRID of execution units; 3D scheduling of instructions
  - Allows very large blocks to be mapped onto GRID
  - Result addresses explicitly specified in 3-dimensions (x,y,z)

Instruction Buffers form a logical "z-dimension" in each node

4 logical *frames* each with 16 instruction slots
Using *frames* for Speculation and ILP

Result:
- Enormous effective instruction window for extracting ILP
- Increased utilization of execution units (*accuracy counts!*)
- Latency tolerance for GRID delays and Load instructions
Results – GPA Instructions per Cycle

![Graph showing GPA Instructions per Cycle for various benchmarks.](image-url)
Using *frames* for Thread-Level Parallelism

**Result:**

- Simultaneous Multithreading (SMT) for Grid Processors
- **Polymorphism:** Use same resources in different ways for different workloads (“T-morph”)
Using *frames* for Data-Level Parallelism

Streaming Kernel:
- read input stream element
- process element
- write output stream element

**Result:**
The instruction buffers act as a *distributed I-Cache*
Ability to absorb and process large amounts of streaming data
Another type of *Polymorphism* (“S-morph”)

- Map very large blocks (kernels)
- Fetch once, use many times
- Not shown: *streaming data channels*
Conclusions

• Technology and Architecture Trends:

  Good News: Lots of transistors, faster transistors
  Bad News: Global wire delays are growing
             Pipeline depth near optimal
             Superscalar pushing the limits of complexity

• GPA Represents a Promising Technology Direction

  ▶ Wire delay constraints: MicroArchitecture and Architecture
  ▶ Eliminates difficult centralized structures dominating today’s designs
  ▶ Architectural partitioning encourages regularity and re-use
  ▶ Enhanced information flow between compiler and hardware
  ▶ Polymorphic features: performance on a wide range of workloads
Future Work

• Architectural Refinement
  • Block-oriented predictors
  • Selective re-execution

• Enhance Compilation and Scheduling Tools
  • Hyperblock formation
  • 3D Instruction Scheduling algorithms

• Compatibility bridge to existing architectures

• Hardware Prototype (currently in planning stage)
  • Four 4x4 GPA cores + NUCA L2 cache on chip
  • 0.10um, ~350mm2, 1000+ signal I/O, 300MHz
  • 4Q 2004 tape-out