The Heterogeneous Block Architecture

A Flexible Substrate for Building Energy-Efficient High-Performance Cores

Chris Fallin$^1$ Chris Wilkerson$^2$ Onur Mutlu$^1$

$^1$ Carnegie Mellon University
$^2$ Intel Corporation
Executive Summary

- **Problem:** General purpose core design is a compromise between performance and energy-efficiency
- **Our Goal:** Design a core that achieves high performance and high energy efficiency at the same time
- **Two New Observations:** 1) Applications exhibit fine-grained heterogeneity in regions of code, 2) A core can exploit this heterogeneity by grouping tens of instructions into atomic blocks and executing each block in the best-fit backend
- **Heterogeneous Block Architecture (HBA):** 1) Forms atomic blocks of code, 2) Dynamically determines the best-fit (most efficient) backend for each block, 3) Specializes the block for that backend
- **Initial HBA Implementation:** Chooses from Out-of-order, VLIW or In-order backends, with simple schedule stability and stall heuristics
- **Results:** HBA provides higher efficiency than four previous designs at negligible performance loss; HBA enables new tradeoff points
Picture of HBA

Shared Frontend

“Specialization”

Out-of-order

Out-of-order

VLIW/IO

VLIW/IO
Talk Agenda

- Background and Motivation
- Two New Observations
- The Heterogeneous Block Architecture (HBA)
- An Initial HBA Design
- Experimental Evaluation
- Conclusions
High performance and high energy-efficiency

- Difficult to achieve both at the same time

High performance: Sophisticated features to extract it

- Out-of-order execution (OoO), complex branch prediction, wide instruction issue, ...

High energy efficiency: Use of only features that provide the highest efficiency for each workload

- Adaptation of resources to different workload requirements

Today's high-performance designs: Features may not yield high performance, but every piece of code pays for their energy penalty
Principle: Exploiting Heterogeneity

- Past observation 1: Workloads have different characteristics at coarse granularity (thousands to millions of instructions)
  - Each workload or phase may require different features

- Can We Design a More Energy-Efficient Core?

- Past coarse-grained heterogeneous designs
  - provide higher energy efficiency
  - at slightly lower performance vs. a homogeneous OoO core
Talk Agenda

- Background and Motivation
- **Two New Observations**
- The Heterogeneous Block Architecture (HBA)
- An Initial HBA Design
- Experimental Evaluation
- Conclusions
Two Key Observations

- **Fine-Grained Heterogeneity of Application Behavior**
  - Small chunks (10s or 100s of instructions) of code have different execution characteristics
    - Some instruction chunks always have the same instruction schedule in an out-of-order processor
    - Nearby chunks can have different schedules

- **Atomic Block Execution with Multiple Execution Backends**
  - A core can exploit fine-grained heterogeneity with
    - Having multiple execution backends
    - Dividing the program into atomic blocks
    - Executing each block in the best-fit backend
Fine-Grained Heterogeneity

- Small chunks of code have different execution characteristics

Coarse-Grained Heterogeneity

\[(1K-1M \text{ insns})\]

<table>
<thead>
<tr>
<th>Phase</th>
<th>Time (instructions)</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>high ILP</td>
<td>regular floating-point</td>
</tr>
<tr>
<td>2</td>
<td>low ILP</td>
<td>pointer-chasing</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>stall</td>
</tr>
</tbody>
</table>

Good fit for Wide VLIW
Good fit for Wide OoO
Good fit for Narrow In-order
Instruction Schedule Stability Varies at Fine Grain

- Observation 1: Some regions of code are scheduled in the same instruction scheduling order in an OoO engine across different dynamic instances

- Observation 2: Stability of instruction schedules of (nearby) code regions can be very different

- An experiment:
  - **Same-schedule chunk**: A chunk of code that has the same schedule it had in its previous dynamic instance
  - Examined all chunks of <=16 instructions in 248 workloads
  - Computed the fraction of “same schedule chunks” in each workload
Instruction Schedule Stability Varies at Fine Grain

1. Many chunks have the same schedule in their previous instances
   - Can reuse the previous instruction scheduling order the next time

2. Stability of instruction schedule of chunks can be very different
   - Need a core that can dynamically schedule insts. in some chunks

3. Temporally-adjacent chunks can have different schedule stability
   - Need a core that can switch quickly between multiple schedulers
Two Key Observations

- **Fine-Grained Heterogeneity of Application Behavior**
  - Small chunks (10s or 100s of instructions) of code have different execution characteristics
    - Some instruction chunks always have the same schedule
    - Nearby chunks can have different schedules

- **Atomic Block Execution with Multiple Execution Backends**
  - A core can exploit fine-grained heterogeneity with
    - Having multiple execution backends
    - Dividing the program into atomic blocks
    - Executing each block in the best-fit backend
Atomic Block Execution w/ Multiple Backends

- Fine grained heterogeneity can be exploited by a core that:
  - Has **multiple (specialized) execution backends**
  - Divides the program into **atomic (all-or-none) blocks**
  - Executes each atomic block in the **best-fit backend**

---

Atomicity enables specialization of a block for a backend (can freely reorder/rewrite/eliminate instructions in an atomic block)
Talk Agenda

- Background and Motivation
- Two New Observations
- The Heterogeneous Block Architecture (HBA)
- An Initial HBA Design
- Experimental Evaluation
- Conclusions
HBA: Principles

- **Multiple different execution backends**
  - Customized for different block execution requirements: OoO, VLIW, in-order, SIMD, ...
  - Can be simultaneously active, executing different blocks
  - Enables efficient exploitation of fine-grained heterogeneity

- **Block atomicity**
  - Either the entire block executes or none of it
  - Enables specialization (reordering and rewriting) of instructions freely within the block to fit a particular backend

- **Exploit stable block characteristics (instruction schedules)**
  - E.g., reuse schedule learned by the OoO backend in VLIW/IO
  - Enables high efficiency by adapting to stable behavior
HBA Operation at High Level

Application → Shared Frontend → "Specialization" → Backend 1 → Backend 2 → Backend 3

Instruction sequence:
- Block 1
- Block 2
- Block 3
HBA Design: Three Components (I)

- Block formation and fetch
- Block sequencing and value communication
- Block execution
3. Heterogeneous block execution units execute different types of blocks in a specialized way and communicate via global registers.
Talk Agenda

- Background and Motivation
- Two New Observations
- The Heterogeneous Block Architecture (HBA)
  - An Initial HBA Design
- Experimental Evaluation
- Conclusions
An Example HBA Design: OoO/VLIW

- Block formation and fetch
- Block sequencing and value communication
- Block execution

- Two types of backends
  - OoO scheduling
  - VLIW/in-order scheduling
Block Formation and Fetch

- Atomic blocks are microarchitectural and dynamically formed
- Block info cache (BIC) stores metadata about a formed block
  - Indexed with PC and branch path of a block (ala Trace Caches)
  - Metadata info: backend type, instruction schedule, ...
- Frontend fetches instructions/metadata from I-cache/BIC
- If miss in BIC, instructions are sent to OoO backend
- Otherwise, buffer instructions until the entire block is fetched

- The first time a block is executed on the OoO backend, its’ OoO version is formed (& OoO schedule/names stored in BIC)
  - Max 16 instructions, ends in a hard-to-predict or indirect branch
An Example HBA Design: OoO/VLIW

- Block formation and fetch
- Block sequencing and value communication
- Block execution

Two types of backends
- OoO scheduling
- VLIW/in-order scheduling
Block Sequencing and Communication

- **Block Sequencing**
  - Block based reorder buffer for in-order block sequencing
  - All subsequent blocks squashed on a branch misprediction
  - Current block squashed on an exception or intra-block misprediction
  - Single-instruction sequencing from non-optimized code upon an exception in a block to reach the exception point

- **Value Communication**
  - Across blocks: via the Global Register File
  - Within block: via the Local Register File
  - Only liveins and liveouts to a block need to be renamed and allocated global registers [Sprangle & Patt, MICRO 1994]
    - Reduces energy consumption of register file and bypass units
An Example HBA Design: OoO/VLIW

- Block formation and fetch
- Block sequencing and value communication
- **Block execution**

- Two types of backends
  - OoO scheduling
  - VLIW/in-order scheduling
Block Execution

- Each backend contains
  - Execution units, scheduler, local register file

- Each backend receives
  - A block specialized for it (at block dispatch time)
  - Liveins for the executing block (as they become available)

- A backend executes the block
  - As specified by its logic and any information from the BIC

- Each backend produces
  - Liveouts to be written to the Global RegFile (as available)
  - Branch misprediction, exception results, block completion signal to be sent to the Block Sequencer

- Memory operations are handled via a traditional LSQ
An Example HBA Design: OoO/VLIW

- Block formation and fetch
- Block sequencing and value communication
- Block execution

- Two types of backends
  - OoO scheduling
  - VLIW/in-order scheduling
OoO and VLIW Backends

- Two customized backend types (physically, they are unified)

**Out-of-order**
- Dynamic scheduling with matrix scheduler
- Renaming performed before the backend (block specialization logic)
- No renaming or matrix computation logic (done for previous instance of block)
- No need to maintain per-instruction order

**VLIW/In-order**
- Static scheduling
- Stall-on-use policy
- VLIW blocks specialized by OoO backends
- No per-instruction order
OoO Backend

Block Dispatch
Liveins (reg, val)

Resolved Branches
Block Completion

Liveouts (reg, val)

Scheduler (no CAM)

Writeback bus

To shared EUs

Generic Interface

Dynamically-scheduled implementation

SAFARI
VLIW/In-Order Backend

Block Dispatch
Liveins (reg, val)

Dispatch Queue
Livein bus

Resolved Branches
Block Completion

Liveouts (reg, val)

Liveout?
Branch?
Counter

Ready-bit Scoreboard

Local RF

Execution Units

Writeback bus

Issue Logic

To shared EUs

Generic Interface

Statically-scheduled implementation
Backend Switching and Specialization

- **OoO to VLIW:** If dynamic schedule is the same for N previous executions of the block on the OoO backend:
  - Record the schedule in the Block Info Cache
  - Mark the block to be executed on the VLIW backend

- **VLIW to OoO:** If there are too many *false stalls* in the block on the VLIW backend
  - Mark the block to be executed on the OoO backend
Talk Agenda

- Background and Motivation
- Two New Observations
- The Heterogeneous Block Architecture (HBA)
- An Initial HBA Design
- Experimental Evaluation
- Conclusions
Experimental Methodology

- **Models**
  - Cycle-accurate *execution-driven x86-64 simulator*
    - ISA remains unchanged
  - Energy model with modified *McPAT [Li+ MICRO’09]*
    - various leakage parameters investigated

- **Workloads**
  - 184 different *representative execution checkpoints*
  - SPEC CPU2006, Olden, MediaBench, Firefox, Ffmpeg, Adobe Flash player, MySQL, lighttpd web server, LaTeX, Octave, an x86 simulator
### Simulation Setup

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Baseline Core:</strong></td>
<td></td>
</tr>
<tr>
<td>Fetch Unit</td>
<td>ISL-TAGE [55]; 64-entry return stack; 64K-entry BTB; 8-cycle restart latency; <strong>4-wide fetch</strong></td>
</tr>
<tr>
<td>Instruction Cache</td>
<td>32KB, 4-way, 64-byte blocks</td>
</tr>
<tr>
<td>Window Size</td>
<td><strong>256-μop ROB</strong> 320-entry physical register file (PRF), 96-entry matrix scheduler</td>
</tr>
<tr>
<td>Execution Units</td>
<td>4-wide; 4 ALUs, 1 MUL, 3 FPUs, 2 branch units, 2 load pipes, 1 store address and data pipe each</td>
</tr>
<tr>
<td>Memory Unit</td>
<td>96-entry load queue (LQ), 48-entry store queue (SQ)</td>
</tr>
<tr>
<td>L1/L2 Caches</td>
<td>64KB, 4-way, 5-cycle L1; 1MB, 16-way, 15-cycle L2; 64-byte blocks</td>
</tr>
<tr>
<td>DRAM</td>
<td>200-cycle latency; stream prefetcher, 16 streams</td>
</tr>
<tr>
<td><strong>Heterogeneous Block Architecture (HBA):</strong></td>
<td></td>
</tr>
<tr>
<td>Block Size</td>
<td><strong>16 μops, 16 liveins, 16 liveouts max</strong></td>
</tr>
<tr>
<td>Fetch Unit</td>
<td>Baseline + 256-block into cache, 64 bytes/block</td>
</tr>
<tr>
<td>Global RF</td>
<td>256 entry; 16 rd/8 wr ports; 2-cyc. inter-backend comm.</td>
</tr>
<tr>
<td>Instruction Window</td>
<td>16-entry Block ROB</td>
</tr>
<tr>
<td>Backends</td>
<td><strong>16 unified backends (OoO- or VLIW-mode)</strong></td>
</tr>
<tr>
<td>OoO backend</td>
<td>4-wide, 4 ALUs, 16-entry local RF, 16-entry scheduler</td>
</tr>
<tr>
<td>VLIW backend</td>
<td>4-wide, 4 ALUs, 16-entry local RF, scoreboard sched.</td>
</tr>
<tr>
<td>Shared Execution Units</td>
<td>3 FPUs, 1 MUL, 2 load, 1 store address/1 store data; 2-cycle roundtrip penalty for use</td>
</tr>
<tr>
<td>LQ/SQ/L1D/DRAM</td>
<td>Same as baseline</td>
</tr>
</tbody>
</table>
Four Core Comparison Points

- **Out-of-order** core with large, monolithic backend

- **Clustered** out-of-order core with split backend [Farkas+ MICRO’97]
  - Clusters of <scheduler, register file, execution units>

- **Coarse-grained heterogeneous design** [Lukefahr+ MICRO’12]
  - Combines out-of-order and in-order cores
  - Switches mode for coarse-grained intervals based on a performance model (ideal in our results)
  - Only one mode can be active: in-order or out-of-order

- **Coarse-grained heterogeneous design**, with clustered OoO core
Key Results: Power, EPI, Performance

Averaged across 184 workloads

<table>
<thead>
<tr>
<th>Row</th>
<th>Configuration</th>
<th>△ Power</th>
<th>△ EPI</th>
<th>△ IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4-wide OoO (Baseline)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>4-wide Clustered OoO [18]</td>
<td>-11.5%</td>
<td>-8.3%</td>
<td>-1.4%</td>
</tr>
<tr>
<td>3</td>
<td>Coarse-grained [37]</td>
<td>-5.4%</td>
<td>-8.9%</td>
<td>-1.2%</td>
</tr>
<tr>
<td>4</td>
<td>Coarse-grained, Clustered</td>
<td>-16.9%</td>
<td>-17.3%</td>
<td>-2.8%</td>
</tr>
<tr>
<td>5</td>
<td>HBA, OoO Backends Only</td>
<td>-28.7%</td>
<td>-25.5%</td>
<td>+0.4%</td>
</tr>
<tr>
<td>6</td>
<td>HBA, OoO/VLIW</td>
<td>-36.4%</td>
<td>-31.9%</td>
<td>-1.0%</td>
</tr>
</tbody>
</table>

- HBA greatly reduces power and energy-per-instruction (>30%)
- HBA performance is within 1% of monolithic out-of-order core
- HBA’s performance higher than coarse-grained hetero. core
- HBA is the most power- and energy-efficient design among the five different core configurations
Energy Analysis (I)

- HBA reduces energy/power by concurrently
  - Decoupling Execution Backends to Simplify the Core (Clustering)
  - Exploiting Block Atomicity to Simplify the Core (Atomicity)
  - Exploiting Heterogeneous Backends to Specialize (Heterogeneity)

- What is the relative energy benefit of each when applied successively?
Energy Analysis (II)

HBA effectively takes advantage of clustering, atomic blocks, and heterogeneous backends to reduce energy.
Comparison to Best Previous Design

- Coarse-grained heterogeneity + clustering the OoO core

Averaged across 184 workloads

Energy/Instruction (nJ)

HBA provides higher efficiency (+15%) at higher performance (+2%) than coarse-grained heterogeneous core with clustering

HBA: 15% lower EPI

+Coarse-grained hetero: 9%

+Clustering: 9%
Per-Workload Results

- HBA reduces energy on almost all workloads
- HBA can reduce or improve performance (analysis in paper)
- HBA enables new design points in the tradeoff space
Cost of Heterogeneity

- Higher core area
  - Due to more backends
  - Can be optimized with good design (unified backends)
  - Core area cost increasingly small in an SoC
    - only 17% in Apple’s A7
  - Analyzed in the paper but our design is not optimized for area

- Increased leakage power
  - HBA benefits reduce with higher transistor leakage
  - Analyzed in the paper
More Results and Analyses in the Paper

- Performance bottlenecks
- Fraction of OoO vs. VLIW blocks
- Energy optimizations in VLIW mode
- Energy optimizations in general
- Area and leakage

- Other and detailed results available in our technical report

Talk Agenda

- Background and Motivation
- Two New Observations
- The Heterogeneous Block Architecture (HBA)
- An Initial HBA Design
- Experimental Evaluation
- Conclusions
Conclusions

- HBA is the first heterogeneous core substrate that enables
  - concurrent execution of fine-grained code blocks
  - on the most-efficient backend for each block

- Three characteristics of HBA
  - Atomic block execution to exploit fine-grained heterogeneity
  - Exploits stable instruction schedules to adapt code to backends
  - Uses clustering, atomicity and heterogeneity to reduce energy

- HBA greatly improves energy efficiency over four core designs

- A flexible execution substrate for exploiting fine-grained heterogeneity in core design
Building on This Work …

- HBA is a substrate for efficient core design
- One can design different cores using this substrate:
  - More backends of different types (CPU, SIMD, reconfigurable logic, ...)
  - Better switching heuristics between backends
  - More optimization of each backend
  - Dynamic code optimizations specific to each backend
  - ...

SAFARI
The Heterogeneous Block Architecture

A Flexible Substrate for Building Energy-Efficient High-Performance Cores

Chris Fallin\textsuperscript{1}  Chris Wilkerson\textsuperscript{2}  Onur Mutlu\textsuperscript{1}

\textsuperscript{1} Carnegie Mellon University  \textsuperscript{2} Intel Corporation
Why Atomic Blocks?

- Each block can be pre-specialized to its backend (pre-rename for OoO, pre-schedule VLIW in hardware, reorder instructions, eliminate instructions)

- Each backend can operate independently

- User-visible ISA can remain unchanged despite multiple execution backends
Nearby Chunks Have Different Behavior

Chunks Retired Before Chunk-Type Switch
In-Order vs. Superscalar vs. OoO

**Average Performance (IPC)**

- In-Order
- Superscalar
- Out-of-Order

**Average Core Power (W)**

- In-Order
- Superscalar
- Out-of-Order
Baseline EPI Breakdown

Energy/Instruction (nJ)

- L2
- L1
- LSQ
- Bypass Buses
- Exec (ALUs)
- RF
- RS (Scheduler)
- ROB
- RAT
- Frontend

1-wide In-order
4-wide In-order
4-wide OoO
Retired Block Types (I)

Fraction of Retired Blocks

Benchmark (Sorted by OoO Fraction)

0
0.2
0.4
0.6
0.8
1

VLIW blocks (wide or narrow)

OoO blocks
Retired Block Types (II)

Frequency Spectrum of Retire-Order Block Types

Spectrum

Frequency ( / 64 block retires)
Retired Block Types (III)

Average Block Type: Histogram per Block

Frequency

0
0.1
0.2
0.3
0.4
0.5

OoO
50% OoO/VLIW
VLIW

Average Block Type
BIC Size Sensitivity

Relative IPC

Block Info Cache Size (blocks of 16 uops)

default
Livein-Liveout Statistics

![Liveins per Block](chart)

![Liveouts per Block](chart)
Auxiliary Data: Benefits of HBA

The diagram illustrates the normalized accesses for different components, comparing Baseline and HBA conditions.

- **RAT**
  - Baseline: Approximately 0.5
  - HBA: Slightly less than 0.5

- **ROB**
  - Baseline: Approximately 0.4
  - HBA: Lower than Baseline

- **Global RF**
  - Baseline: Approximately 0.6
  - HBA: Lower than Baseline

The graph shows that the HBA condition generally results in lower normalized accesses compared to the Baseline condition.
Auxiliary Data: Benefits of HBA

The charts above illustrate the normalized access rates for RAT, ROB, and RF for Baseline and HBA under different conditions.

- **Normalized RAT Accesses**
  - Baseline
  - HBA

- **Normalized ROB Accesses**
  - Baseline
  - HBA

- **Normalized RF Accesses**
  - Baseline
  - HBA

The charts show a comparison of reads and writes for each category, indicating the benefits of HBA in reducing access times in these systems.
Some Specific Workloads

![Graph showing relative performance vs. baseline for various workloads.

- **Workloads**: ffmpeg, 473.astar, 429.mcf, latex, 401.bzip2, graphchi, 403.gcc, 437.leslie3d, octave, 456.hmmer, 450.soplex, 433.milc

- **Axes**:
  - Y-axis: Rel. vs. Baseline (0.5, 1)
  - X-axis: Workloads

- **Lines**:
  - IPC
  - EPI

- **Legend**:
  - IPC
  - EPI

SAFARI
Related Works

- Forming and reusing instruction schedules [DIF, ISCA’97] [Transmeta’00][Banerjia, IEEE TOC’98][Palomar, ICCD’09]

- Coarse-grained heterogeneous cores [Lukefahr, MICRO’12]

- Atomic blocks and block-structured ISA [Melvin&Patt, MICRO’88, IJPP’95]

- Instruction prescheduling [Michaud&Seznec, HPCA’01]

- Yoga [Villavieja+, HPS Tech Report’14]