The Heterogeneous Block Architecture

[§†]Chris Fallin cfallin@c1f.net
[†]Chris Wilkerson chris.wilkerson@intel.com
[§]Onur Mutlu onur@cmu.edu

[§]Carnegie Mellon University
[†]Intel Corporation

Abstract

This paper makes two new observations that lead to a new heterogeneous core design. First, we observe that most serial code exhibits fine-grained heterogeneity*: at the scale of tens or hundreds of instructions, regions of code fit different microarchitectures better (at the same point or at different points in time). Second, we observe that by grouping contiguous regions of instructions into* blocks *that are executed atomically, a core can exploit this heterogeneity: atomicity allows each block to be executed independently on its own execution backend that fits its characteristics best.*

Based on these observations, we propose a fine-grained heterogeneous design *that combines heterogeneous execution backends into one core. Our core design, the* heterogeneous block architecture (HBA)*, breaks the program into blocks of code, determines the best backend for each block, and* specializes *the block for that backend. As an initial, concrete design, we combine out-of-order, VLIW, and in-order backends, using simple heuristics to choose backends. We compare HBA to multiple baseline core designs (including monolithic out-of-order, clustered out-of-order, in-order and a state-of-the-art heterogeneous core design) and show that HBA can provide significantly better energy efficiency than all designs at similar performance. Averaged across 184 traces from a wide variety of workloads, HBA reduces core power by 36.4% and energy per instruction by 31.9% compared to a 4-wide out-of-order core. We conclude that HBA provides a flexible substrate for exploiting fine-grained heterogeneity, enabling new energy-performance tradeoff points in core design.*

1. Introduction

Core designs today face two competing goals. They must continue to improve *single-thread (serial) performance*, because such performance is important for many algorithms and for any application with serial bottlenecks [3, 53, 26]. Cores must also improve *energy efficiency* because it has become a primary limiter: datacenters are largely power-limited, so that improved energy efficiency allows more servers and higher throughput, while many consumer devices are battery-powered, so that improved energy efficiency translates to longer battery life.

A primary difficulty in achieving both high performance and high energy efficiency is that no single core microarchitecture is the best design for both metrics for all programs. Any particular design spends energy on some set of features (e.g., out-of-order instruction scheduling, sophisticated branch prediction, or wide pipeline width), but these features do not always yield improved performance. As a result, a general-purpose core today is usually a *compromise*: it is designed to meet some performance objectives while remaining within a power envelope, but for any given program, it is frequently not the most efficient design.

The difficulty in designing a good general-purpose core arises because code is *heterogeneous*, both between and within programs. In other words, each program has different characteristics, and a single program may have different characteristics in different regions of its code. To exploit this diversity, past works proposed *core-level heterogeneity*. These heterogeneous designs either combine multiple separate cores (e.g., [29, 22, 3, 20, 53, 7, 12, 26, 4, 56]), or else combine an in-order pipeline and out-of-order pipeline with a shared frontend in a single core [35]. Past works demonstrate energy-efficiency improvements with usually small impact to performance.

This paper makes two key observations that motivate a new way of building a heterogeneous core. Our **first observation** is that *applications have fine-grained heterogeneity.* Prior work has observed heterogeneity at a coarse granularity: for example, programs have *memory* and *compute* phases, and such phases can be exploited by migrating a thread to "big" cores for compute-intensive phases and "little" cores for memory-intensive phases [56, 35]. However, at a finer granularity (of tens to hundreds of instructions), adjacent blocks of code often have different properties as well (as we will show). For example, one block of code might have a consistent instruction schedule across its dynamic execution instances in an OoO machine, whereas neighboring blocks schedule each execution differently depending on cache miss behavior. Such behavior suggests the use of both dynamic and static schedulers. Migration of execution between separate cores or pipelines cannot easily exploit this fine-grained property.

Our **second observation** is that a core can exploit *fine-grained heterogeneity* if it splits code into *atomic blocks* and executes each block on a separate *execution backend*, which includes functional units, local storage, and some form of instruction scheduling. In order to exploit fine-grained heterogeneity, a core will need execution backends of multiple types, and the core will need to *specialize* pieces of code for each backend. By enforcing *atomicity*, or the property that a region (block) of code either executes successfully or not at all, the core can freely analyze and *morph* this block of code to fit a particular backend. For example, atomicity allows the core to reorder instructions freely within the block. Atomic block-based design allows execution backends to operate independently except for a well-defined interface (liveins/liveouts) between blocks.

Based on these two key observations, we propose a *fine-grained heterogeneous core* that dynamically forms user code

into blocks, *specializes* those blocks to execute on the most appropriate type of execution backend, and executes blocks on these various backends. This core design serves as a general substrate for fine-grained heterogeneity which can combine many different types of execution backends. As an initial heterogeneous design, this paper evaluates a core which includes *out-of-order*, *VLIW*, and *in-order* execution backends, and logic to assign each block to a backend. Our design first executes each block on the out-of-order execution backend, but monitors schedule stability of the block over time. When a block of code has an unchanging instruction schedule, indicating that instruction latencies are likely not variable, it is converted to a VLIW or in-order block (depending on ILP), using the instruction schedule recorded during out-of-order execution. When the block again requires dynamic scheduling (determined based on a simple stall-cycle statistic), it is converted to an out-of-order block. At any given time, multiple backend types can be active for different blocks.

Our major contributions are:

1. We design a *fine-grained heterogeneous core* which forms atomic blocks of code and executes these blocks on out-of-order, VLIW, and in-order backends, depending on the observed instruction schedule stability and ILP of each block.

2. We propose mechanisms that enable blocks of code to be executed on VLIW and in-order execution backends without requiring static scheduling by leveraging dynamic information from an out-of-order execution backend to produce a code schedule at runtime.

3. Extensive evaluations of our core design in comparison to an out-of-order core, a clustered-microarchitecture core [17], and a state-of-the-art coarse-grained heterogeneous core [35] demonstrate higher energy efficiency than all previous designs across a variety of workloads. Our design reduces average core power by 36.4% with 1% performance degradation over the out-of-order baseline. In addition, our design provides a flexible substrate for future heterogeneous designs, enabling new power-performance tradeoff points in core design.

2. Motivation: Fine-Grained Heterogeneity

Our first major observation is that **applications have fine-grained heterogeneity**. In this context, "fine-grained" indicates regions of tens or hundreds of instructions. This level of heterogeneity is quite distinct from larger program phases [15, 50] that occur because a program switches between wholly different tasks or modes. Fine-grained heterogeneity occurs when small chunks of code have different characteristics due to the particular instructions or dataflow in the chunks.

Fig. 1 represents this distinction graphically. The left half of the figure depicts an application that has at least two phases: a regular floating-point phase and a memory-bound pointer-chasing phase. These phases occur at a scale of thousands to millions of instructions. If we focus on one small portion of the first phase, however, we see *fine-grained* heterogeneity. The right half of the figure depicts a region tens of instructions long within the floating point phase. In the first group of instructions, three of the four operations are independent and can issue in parallel. All instructions in this group also have constant, statically-known latencies. Hence, the small group of instructions has high ILP and a consistent (unchanging) dynamic instruction schedule. The second instruction group also has high ILP, but has variable scheduling due to intermittent cache misses. Finally, the third instruction group has low ILP due to a dependence chain. Overall, each small code region within this single "regular floating point" phase has different properties, and thus benefits from different core features.

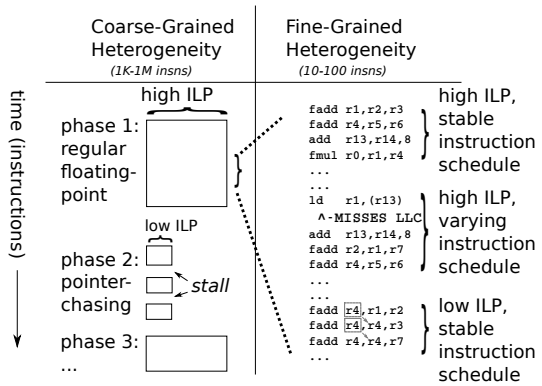

Figure 1: Coarse-grained vs. fine-grained heterogeneity.

To motivate our approach, we demonstrate the existence of a particular code property, *instruction schedule stability*: some regions of code always (or frequently) schedule in the same order in the dynamic out-of-order scheduling logic. We also demonstrate that this property varies greatly between different nearby regions of code (hence, is *fine-grained*). To do this, we analyze the behavior of a large set of benchmarks on a 4-wide superscalar out-of-order core (§4 provides all parameters). We observe the retired instruction stream and group these dynamic instructions into *chunks* of up to 16 μops each. Chunks are broken at certain boundaries according to the heuristics in §3.3.1. Then, for each chunk, we compare the actual dynamic instruction schedule of that chunk to the schedule of the previous instance of the same (static) code. We record whether the schedule was the same as or different from before. These annotations, called "chunk types," indicate the extent to which each chunk has a stable schedule.

Fig. 2 shows the distribution of the "same" and "different" chunk types, per benchmark, as a sorted curve. Many chunks repeat prior instruction schedules. Hence, there is significant opportunity to reuse schedules. Furthermore, there are many applications (in the center of the plot) that exhibit a *mix* of behavior: between 20% and 80% of retired chunks of instructions exhibit consistent schedules. Hence, individual applications often have *heterogeneous* instruction schedule stability across different regions of code. This observation suggests a core design that can reuse instruction schedules for some code and dynamically schedule other code.

Moreover, we observe that this heterogeneity exists between *nearby* chunks in the dynamic instruction stream: in other words, that there is *fine-grained heterogeneity*. We observe the sequence of chunk types in retire order and group chunks into

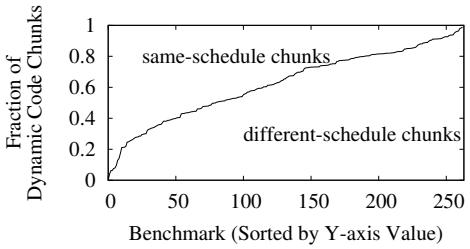

Figure 2: Fraction of chunks in the instruction stream that have a different schedule than their previous instance.

runs. One run is a consecutive series of chunks with the same instruction schedule stability. We then accumulate the length of all runs. The length of these runs indicates whether the heterogeneity is coarse- or fine-grained. We find that almost 60% of runs are of length 1, and the histogram falls off rapidly thereafter. In other words, the schedule stability of chunks of instructions is often different even between temporally-adjacent static regions of code. This behavior indicates *fine-grained heterogeneity*.

This fine-grained heterogeneity exists *within* program phases. It is thus distinct from coarse-grained (inter-program/inter-phase) heterogeneity exploited by, e.g., heterogeneous multicores. Instead, it motivates a new approach: a single core that can execute nearby chunks of code with different mechanisms best suited for each chunk.

3. A Fine-Grained Heterogeneous Core

We introduce our new core design, *HBA* (Heterogeneous Block Architecture), based on our observations so far.

3.1. High-Level Overview

Key Idea #1: Build a core that executes fine-grained blocks of code on heterogeneous backends. As we demonstrated in §2, application code is heterogeneous at not only coarse but also fine granularity. We thus build a core that contains many heterogeneous execution backends *within a single core* to exploit this heterogeneity. The core groups the application's instructions into chunks (which we call *blocks*) and determines the best type of execution backend for each block. Multiple execution backends can be active simultaneously with different chunks of code, and these backends communicate program values directly.

Key Idea #2: Leverage block atomicity to allow block specialization. In order to allow for a block of code to be adapted properly to a particular backend, the block of code must be considered as a *unit*, isolated from the rest of the program. Our second key idea is to impose **atomicity** on each block of code: the core either commits each block's side effects at once, or throws away them all. Atomicity guarantees the core will see an entire block; hence, it allows the use of backends that leverage code properties extracted once over the entire region (e.g., by reordering or rewriting instructions) to adapt the block to a particular backend. Atomicity thus *enables the core to exploit fine-grained heterogeneity*.

Key Idea #3: Combine out-of-order and VLIW/in-order execution backends by using out-of-order execution to form stable VLIW schedules. Our final idea leverages dynamically-scheduled (out-of-order) execution in order to enable statically-scheduled (VLIW/in-order) execution with runtime-informed instruction schedules. The out-of-order backend observes the dynamic schedule and, when it is *stable* (unchanging) over multiple instances of the same code, records the schedule and uses it for VLIW or in-order execution. If the core later determines that this schedule leads to unnecessary stalls, the schedule is thrown away and the block is again executed by the out-of-order backend. Hence, most of the the performance of out-of-order execution is retained while saving a significant amount of energy.

3.2. Atomicity

Here, we briefly describe several concepts that are important to understanding our design. First, **atomicity** of a block means that a block either completes execution and commits its side-effects, or else is squashed and has no side-effects. This is in contrast to a conventional core, in which the atomic unit of execution is a single instruction, and each instruction commits its results separately. Second, **liveins** and **liveouts** are the inputs and outputs, respectively, to and from a block. A livein is any register that an instruction in a block *reads* that is not written (produced) by an earlier instruction in the block. A liveout is any register that an instruction in a block *writes* that is not overwritten by a later instruction in the block.

3.3. Block Core Design

Fig. 3 illustrates the basic HBA design. HBA is a block-oriented microarchitecture. The core consists of three major parts: (i) *block fetch*, (ii) *block sequencing and communication*, and (iii) *block execution*. We discuss each in turn.

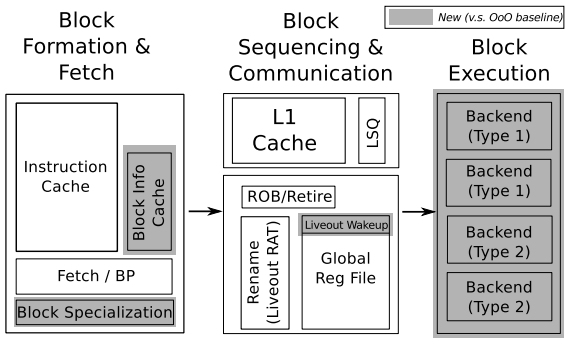

Figure 3: HBA (Heterogeneous Block Architecture) overview.

3.3.1. Block Fetch The HBA core executes user code that conforms to a conventional ISA. Hence, the code must be formed into blocks *dynamically*. These blocks are *microarchitectural*: the block-level interface is not software-visible. In order to avoid storing every block in full, the HBA core uses an instruction cache (as in the baseline), and stores only *block metadata* in a *block info cache*. This cache is indexed with the start PC of a block and its branch path, just as in a conventional trace cache [47, 42]. The block info cache stores information that the core has discovered about the block. This information depends

on the block type: for example, for a block that executes on a VLIW backend, the info includes the instruction schedule.

At fetch, the block frontend fetches instructions from the I-cache, using a conventional branch predictor. In parallel, it looks up information in the block info cache. As instructions are fetched, they are not sent to the backend right away, but are kept in a *block buffer*. These instructions become a block and are sent to the backend either when the block name (PC and branch path) *hits* in the block info cache, and no longer matches exist, or else when the PC and branch path *miss* in the block info cache. If there is a miss, the block takes on default characteristics: it executes on an OoO backend, which requires no pre-computed information about the block. In this case, the block is terminated whenever any *block termination condition* holds: when (i) it reaches a maximum length, (ii) it ends at an indirect branch, or (iii) it ends at a difficult-to-predict conditional branch, as determined by a small (1K-entry) table of 2-bit saturating counters incremented whenever a branch is mispredicted.

The use of a block *info* cache in parallel with a conventional instruction cache, rather than a full trace cache [47, 42], allows the HBA core to achieve the *best of both worlds*: it achieves the space efficiency of the instruction cache while retaining the learned information about code blocks (enabling the core to leverage fine-grained heterogeneity between blocks).

3.3.2. Block Sequencing and Communication The central portion of the core depicted in Fig. 3 handles *sequencing and communication*: that is, managing block program order and repairing it upon branch mispredicts, sending blocks to the appropriate execution units, and communicating program values between those execution units.

Block Dispatch and Sequencing: Once a block is fetched, the *block dispatch* logic must send it to an appropriate execution backend. Each execution backend executes one block at a time until all operations within that block are complete. The block dispatch logic maintains one free-list of block execution backends per type, and allocates the appropriate type for each block. The backend is returned to the free list soon as it completes the block's execution.

The *block sequencing* logic maintains program order among blocks in flight and handles branch misprediction recovery. The logic contains a block-level ROB (reorder buffer), analogous to the ROB in a conventional out-of-order design. Two types of branch mispredicts can occur: *intra-block* mispredicts, which involve a conditional branch in the middle of a block, and *inter-block* mispredicts, which involve a branch (conditional or indirect) that is the last instruction in a block. Inter-block misprediction recoveries squash the blocks that follow the mispredicted branch in program order, roll back state using the ROB, and restart the frontend on the proper path. Intra-block mispredicts must additionally squash the block that contains the mispredicted branch, because of block-level atomicity (the core cannot keep half of the block and squash the other half). The frontend is then directed to re-fetch a block with a *different branch-path* at the same fetch PC.

Global Registers and Liveout-Livein Communication: Blocks executing on block execution backends communicate via *global registers* that receive liveouts from producer blocks as they execute and provide liveins to consumer blocks. The global register file is centrally located between the block execution backends. In addition to data values, this logic contains *subscription bits* and a *livein wakeup unit*, described in more detail below.

When a block is dispatched, its liveins are renamed by looking up global register pointers in a *liveout register alias table (RAT)*, which contains an entry for each architectural register. Its liveouts are then allocated global registers and the *liveout RAT* is updated. Liveout-to-livein communication between blocks occurs as soon as a given liveout is produced within a backend. The liveout is first written to the global register file. The livein wakeup unit then sends the value to any blocks that consume it as a livein. In other words, blocks do not read liveins all at once, then execute, then write all liveouts. Rather, values are communicated from producers to consumers as soon as the values become available. (Without this design choice, the core achieves only 31% of its performance due to the false dependences.)

In order to support this wakeup, each global register has a corresponding set of *subscription bits* indicating waiting backends. When a block is dispatched, it subscribes to its livein registers. At each global writeback, the subscription bits allow wakeups to be sent efficiently to each consuming backend. This structure is similar to a bit-matrix scheduler [21].

3.3.3. Block Execution Finally, when a block is sent to a block execution backend, the backend executes the block in an implementation-specific way. Each backend receives (i) a block specialized for that backend, and (ii) a stream of liveins for that block, as they become available. The backend performs the specified computation and produces (i) a stream of liveouts for its block, as they are computed, (ii) any branch misprediction results, and (iii) a completion signal. When a block completes execution, the backend clears out the block and becomes ready to receive another block.

In this initial work, we implement two types of block execution backends: an out-of-order backend and a VLIW/in-order backend. Both types of backends share a common datapath design, and differ only in the instruction issue logic and pipeline width. Note that these backends represent only two points in a wide design spectrum; more specialized backends are possible and are left for future work.

Local execution cluster: Both the out-of-order and VLIW/in-order execution backends in our design are built around a *local execution cluster* that contains simple ALUs, a local register file, and a bypass/writeback bus connecting them. When a block is formed, each instruction in the block is allocated a destination register in the local register file. An instruction that produces block live-outs additionally sends its result to the global register file.

Shared execution units: In addition to the simple ALUs in each execution backend, the HBA core shares its more expensive execution units (such as floating-point units and load/store pipelines). Execution backends arbitrate for access to these units when instructions are issued (and arbitration conflicts are handled oldest-block-first, with conflicting instructions wait-

ing in skid buffers to retry). Sharing these units between all execution backends amortizes these units' cost [30].

Memory operations: The HBA core shares the L1 cache, the load/store queue, and the load/store pipelines between all execution backends. Note that the use of blocks is orthogonal to both the correctness and performance aspects of memory disambiguation. Because our core design achieves similar performance to the baseline core, as we show later, the same memory pipeline throughput is sufficient. Because the backends generate the same single thread's memory requests (by executing the same architectural program), and each block can specify a load/store order within the block, memory ordering remains correct. Memory disambiguation does not (and need not) interact with block atomicity, except that all stores in a block logically commit at once.

Load/store queue (LSQ) entries are allocated in program order when blocks are dispatched, and the LSQ ensures that loads receive correct values and that stores commit in-order when their associated blocks commit. Any load replay due to incorrect memory speculation can be handled by sending the load back to the affected block's execution backend (just as a conventional out-of-order machine's LSQ sends misspeculated loads back for re-execution). For simplicity, we use conservative disambiguation in our evaluations, but speculation is no more difficult in HBA than in an out-of-order design.

Out-of-order execution backend (Fig. 4a): This backend implements dataflow-order instruction scheduling within a block. The instruction scheduler is bit matrix-based [21, 48]. When a livein is received at the backend, it wakes up dependents as any other value writeback would. Note that because the block execution backend does not need to maintain program order within the block (because blocks are atomic), the backend has no equivalent to a ROB. Rather, it has a simple counter that counts completed instructions and signals block completion when all instructions have executed.

In order to *specialize* a block for the out-of-order backend, the block specialization logic (i) *pre-renames* all instructions in the block, and (ii) *pre-computes the scheduling matrix*. This information is stored in the block info cache and provided with the block if present. Because the out-of-order backend also executes *new* blocks which have no information in the block info cache, this logic also performs the renaming and computes this information for the first dynamic instance of each new block. Because of this block specialization, the out-of-order backend does not need any renaming logic and does not need any dynamic matrix allocation/setup logic (e.g., the producer tables in Goshima et al. [21]). These simplifications save energy relative to a baseline out-of-order core.

VLIW execution backend (Fig. 4b): Unlike the out-of-order backend, the VLIW backend has no out-of-order instruction scheduler. Instead, it contains an issue queue populated with pre-formed instruction bundles, and a scoreboard stage that stalls the head of the issue queue until the sources for all of its constituent instructions are ready. The scoreboard implements a stall-on-use policy for long-latency operations.

Specialization of blocks for the VLIW backend is more involved than for the out-of-order backend because VLIW execution requires pre-formed *bundles* of instructions. Rather than require the compiler to form these bundles (which requires a new ISA), the HBA core leverages the existing instruction-scheduling logic in the core to form bundles dynamically at runtime. We now describe this specialization and its interaction with the out-of-order execution backends.

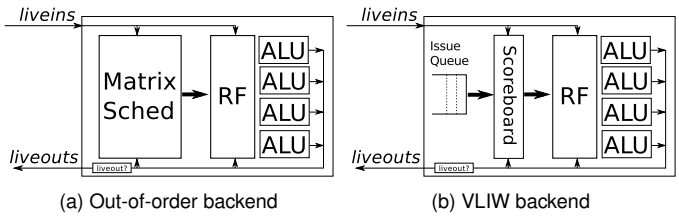

Figure 4: Block execution backend designs.

3.4. Combining Out-of-Order and VLIW Execution

The key to exploiting heterogeneity successfully is to use the most appropriate execution backend for each block of code. In order to combine out-of-order and VLIW block execution backends, our initial HBA design introduces a set of simple mechanisms which we call *memoized scheduling*.

Memoized scheduling makes use of the observation (seen in §2) that small blocks of code often exhibit *schedule stability*. The **key idea** of memoized scheduling is to make use of an out-of-order execution backend to initially execute a block of code, but also observe its schedule stability. Each time the block executes on an out-of-order backend, its instruction schedule (as executed) is recorded. If the instruction schedule of the block remains stable (i.e., changes very little or not at all) over multiple executions of that block, then the block of code is converted to use a VLIW backend. (Our evaluations use a threshold of four consecutive executions that use exactly the same schedule.) The recorded instruction schedule is taken as a set of instruction bundles for a VLIW backend. The VLIW backends are designed to have the same issue width and functional units as the out-of-order backends so that the recorded schedule can be used verbatim. Thus, the schedule is *recorded and replayed*, or *memoized*.

In the base case, if the schedule stability is long-term, then subsequent executions of the block on the VLIW backend will have the same performance as if the out-of-order backend were used, and the execution will use less energy because instruction scheduling is no longer performed dynamically. However, subsequent executions may experience *false stalls*, or cycles in which a VLIW instruction bundle stalls because some but not all of its bundled instructions are not ready to execute. False stalls occur because the dynamic timing (e.g., livein availability or a long-latency cache miss) differs from when the schedule was recorded; the out-of-order engine could have used a different schedule to execute the instructions that are ready. These false stalls thus result in performance loss with respect to execution on an out-of-order engine. In order to ensure timely re-scheduling, the VLIW backend monitors such false stall cycles. If the number of false stall cycles for each block (as a ratio of all execution cycles for that given block) exceeds a

threshold of 5%, the schedule is discarded and the block then executes on an out-of-order backend next time it is dispatched.

3.4.1. Universal Block Backends: Combining OoO/VLIW Hardware So far, we have exploited instruction-scheduling heterogeneity by using two separate types of execution backends: one (out-of-order) that performs dynamic scheduling and one (VLIW) that uses a static schedule. We observe, however, that a VLIW backend's hardware is almost a subset of the out-of-order backend's hardware. The pipeline configurations are identical and only the scheduler is different. Thus, we combine the two into one *universal block backend* that simply turns off its out-of-order scheduling logic (bit-matrix) when it is executing in VLIW mode. (One other recent work, MorphCore [27], also observed that an in-order scheduler can be made by disabling parts of an out-of-order scheduler.)

3.5. Reducing Execute Power

So far, we have exploited instruction-scheduling heterogeneity. We now describe two ways of exploiting further degrees of heterogeneity within blocks executing on VLIW backends: (i) *dynamic pipeline narrowing* and (ii) *dead write elision*.

Dynamic Pipeline Narrowing: Many blocks cannot utilize the full superscalar width of a VLIW backend. To exploit this, the VLIW block formation process records the maximum bundle width across all VLIW bundles that are formed from the recorded instruction schedule of the block. When a VLIW backend executes a block, it can dynamically narrow its issue width if the block will not use some of the issue slots, thus saving static and dynamic energy. (Note that an "in-order" backend is simply a VLIW backend that has reduced its width to one way.) These savings occur via clock and power gating to unused execution units and superscalar issue logic.

Dead Write Elision: In a block executing on a VLIW backend, any values that are used only while on the bypass network need not be written to their local destination register. Conversely, any values that are never used while on the bypass network need not be written to the bypass network. (Values that are never used locally at all need not be written to either.) The VLIW block formation process detects values with such unnecessary writes and marks them to be elided by VLIW backends, thus saving dynamic energy.

4. Methodology

System Configuration: We evaluate HBA against several baseline core designs in a single-core system. We model a simple memory hierarchy, including an L2 cache, DRAM, and a prefetcher. Table 1 gives major system parameters.

Simulator: To model the behavior of the HBA core, we employ an in-house cycle-accurate simulator. Our simulator is execution-driven. All major structures and algorithms within the HBA core are modeled. The model executes user-mode x86-64 code by cracking instructions into μops (using a modified PTLsim [58] decoder). To collect each checkpoint/trace, we use a custom Pin-tool [34] and use PinPoints [44] to find the representative portions of benchmarks.

Power Model: To model core power/energy, we use a modified

Parameter	Setting
	Baseline Core
Fetch Unit	ISL-TAGE [49]; 64-entry return stack; 64K-entry BTB; 8-cycle restart latency; 4-wide fetch
Instruction Cache	32KB, 4-way, 64-byte blocks
Window Size	256-μop ROB, 320-entry physical register file (PRF), 96-entry matrix scheduler
Execution Units	4-wide issue; 4 ALUs, 1 MUL, 3 FPUs, 2 branch units, 2 load pipes, 1 store address/1 store data pipe.
Memory Unit	96-entry load queue (LQ), 48-entry store queue (SQ), conservative disambiguation
L1/L2 Caches	64KB, 4-way, 5-cycle L1; 1MB, 16-way, 15-cycle L2; 64-byte blocks
DRAM	Uniform 200-cycle latency; stream prefetcher, 16 streams
	Heterogeneous Block Architecture (HBA):
Block Size	16 μops, 16 liveins, 16 liveouts max
Fetch Unit	Baseline Fetch Unit; 256-block info cache, 64 bytes/block
Global RF	256 entries; 16 read ports, 8 write ports; 2 cycles inter-backend latency
Instruction Window	16-entry Block ROB
Backends	16 "universal" backends (OoO- or VLIW-mode)
OoO backend	4-wide, 4 ALUs, 16-entry local RF, 16-entry scheduler
VLIW backend	4-wide, 4 ALUs, 16-entry local RF, scoreboard scheduler
Shared Eexecution Units	3 FPUs, 1 MUL, 2 load, 1 store address/1 store data; 2-cycle roundtrip penalty for use
LQ/SQ/L1D/DRAM	Same as baseline

Table 1: Evaluation system parameters.

version of McPAT [33]. To model HBA energy consumption, we use McPAT's component models to construct a faithful model. We replaced McPAT's ALU model with a custom, more accurate, Verilog model we developed and synthesized for a commercial process with Synopsys tools. Energy numbers and formulas used in our model are provided in [1].

One parameter of our model is the sensitivity of design to static power. The parameters in our model are based on a 28nm FDSOI (fully depleted silicon on insulator) process technology as described in [19]. Depending on operating conditions and the choice of low V_t (fast, leaky) devices or regular V_t (slow, less leaky) devices, the relative contribution of static and dynamic power may vary. For example, leakage will be 15% of total power for a processor implemented with fast, low V_t devices operating at nominal voltage (0.9V) [19]. The use of regular leakage devices will reduce leakage power by about an order of magnitude but will reduce performance by about 10–15%. Results will change depending on the characteristics of the underlying process technology and choice of operating voltage. In this work, we focus on two evaluation points: worst-case leakage (all fast low-V_t devices at 0.9V), resulting in 15% of total power, and more realistic leakage with a 50%/50% mix of low-V_t and high-V_t devices, resulting in 10% of total power. A real design [14] might use both types by optimizing critical path logic with fast transistors while reducing power in non-critical logic with less leaky transistors. Our main analysis assumes 10% leakage but we also summarize key results for 15% leakage in §5.1.

Finally, our model assumes power gating in three mecha-

nisms: we gate 1) scheduling logic in BEUs when they are in VLIW mode, 2) superscalar ways when BEUs execute narrow VLIW blocks, and 3) shared execution units (FPUs and the multiplier) in both HBA and in the baseline. We assumed perfect gating as an upper bound; however, we believe that more realistic power or clock gating in the HBA mechanisms could achieve most of the same benefits due to two reasons. First, when most of the power is dynamic, simply gating the scheduling logic's clocks and avoiding its use would remove most of the scheduler power. Second, gating superscalar ways can be accomplished with two techniques: segmenting the bypass/writeback network so that only buses to active ways are driven, and (assuming that the register file achieves its high port count by using multiple replicas) writing results only to register file replicas for active execution lanes/ways.

Workloads: We evaluate HBA with a set of 184 distinct checkpoint/traces, collected from the SPEC CPU2006 [52], Olden [45], and MediaBench [32] suites, and an array of other software: Firefox, FFmpeg, the Adobe Flash player, the V8 Javascript engine, the GraphChi graph-analysis framework [31], MySQL, the lighttpd web server, LaTeX, Octave (a MATLAB replacement), and a checkpoint/trace of the simulator itself. Many of these benchmarks had multiple checkpoint/traces collected at multiple representative regions as indicated by PinPoints. All checkpoint/traces are listed in [1], along with their individual performance and energy consumption on each of the core models evaluated in this paper.

Baseline Core Designs: We compare HBA to four core designs. First, we compare to two variants of a high-performance out-of-order core: one with a monolithic backend (instruction scheduler, register file, and execution units), and one with a *clustered microarchitecture* that splits these structures into separate clusters and copies values between them as necessary [17]. The clusters have equivalent scheduler size and issue width as the block execution backends in the HBA core. We also compare to a coarse-grained heterogeneous design that combines an out-of-order and an in-order core [35]. To be fair in our comparisons, we idealized the controller we model for this coarse-grained design, which provides an upper bound on efficiency and performance relative to the real controller-based mechanism of [35]. To do so, we first run each benchmark on 4-wide OoO and 2-wide in-order cores and record statistics separately for each 1K-instruction epoch, then examine each epoch and combine the simulation results post-mortem. The control algorithm is a PI controller similar to Lukefahr et al. [35] that targets < 5% performance degradation.

5. Evaluation

In this section, we evaluate the performance of HBA in detail in comparison to a comprehensive set of baseline core designs. We will show that three main conclusions hold: (i) HBA has nearly the same performance as a baseline 4-wide out-of-order core, with only 1% performance degradation on average; (ii) HBA saves 36% of average core power relative to this baseline; (iii) HBA is the most energy-efficient design among a large set of evaluated core designs, including monolithic and clustered out-of-order cores, in-order cores, and a state-of-the-art heterogeneous core design (§5.3 summarizes this result by evaluating HBA against a variety of core designs that fall into different power-performance tradeoff points).

5.1. Power

The primary benefit of the HBA design is that it saves significant core energy (and hence, average core power). Table 2 shows average core power and Energy Per Instruction (EPI) (in addition to performance, discussed below) for six core designs: baseline out-of-order, clustered out-of-order [17], coarse-grained heterogeneous [35], coarse-grained heterogeneous combined with clustered out-of-order, HBA with only out-of-order backends, and HBA with heterogeneous backends. Overall, HBA (row 6) reduces average core power by 36.4% and EPI by 31.9% over a baseline out-of-order core (row 1). HBA is also the most energy-efficient core design evaluated in terms of both average core power and EPI.

Row	Configuration	Δ IPC	Δ Power	Δ EPI
1	4-wide OoO (Baseline)	—	—	—
2	4-wide Clustered OoO [17]	-1.4%	-11.5%	-8.3%
3	Coarse-grained [35]	-1.2%	-5.4%	-8.9%
4	Coarse-grained, Clustered	-2.8%	-16.9%	-17.3%
5	HBA, OoO Backends Only	+0.4%	-28.7%	-25.5%
6	HBA, OoO/VLIW	-1.0%	-36.4%	-31.9%

Table 2: Summary of performance, power, and EPI of different core designs compared to the baseline out-of-order core.

To provide more insight into these numbers, Fig. 5 shows a breakdown of the EPI. We make several major observations:
1. Energy reductions in HBA occur for three major reasons: (i) *decoupled execution backends*, (ii) *block atomicity* and (iii) *heterogeneity*. The clustered out-of-order core, which has execution clusters configured equivalently to HBA but which sends instructions to clusters dynamically (item *(i)*), saves 8.3% energy per instruction relative to the baseline monolithic core (first to second bar). Then, leveraging block atomicity (item *(ii)*), the HBA design that uses only out-of-order execution backends reduces energy by a further 17.2% (second to fourth bar). Finally, making use of all heterogeneous execution backends (item *(iii)*) reduces energy by 6.4% (fourth to fifth bar). We now examine each of these trends in more detail.
2. *Decoupled execution backends:* the clustered core saves energy in the instruction scheduling because each cluster has its own dynamic scheduling logic that operates independently of the other clusters. As a result, the RS (scheduler) power reduces by 71% moving from the first to second bar in Fig. 5.
3. *Block atomicity:* HBA, even without heterogeneous backends, saves energy in renaming (RAT), program-order sequencing/retire (ROB), and program value storage (global register file) because it tracks blocks rather than instructions. Fig. 6 shows the root cause of these savings: (i) the block core renames only liveouts, rather than all written values, so RAT accesses reduce by 62%; (ii) the block core dispatches and retires whole blocks at a time and stores only information about liveouts in the ROB, thus reducing ROB accesses by 74%; and

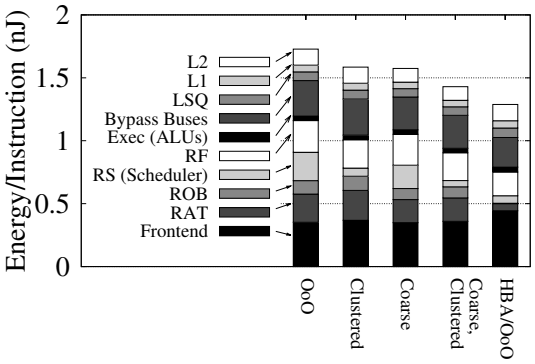

Figure 5: Energy-per-Instruction (EPI) breakdowns.

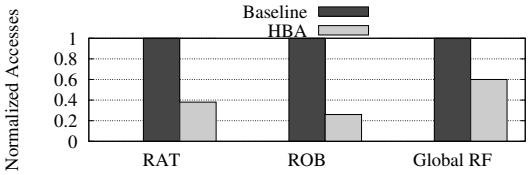

Figure 6: Reductions in basic core events in HBA due to use of atomic blocks.

(iii) only 60% of register file accesses go to a large central register file.

4. *Heterogeneity*: the HBA design with all mechanisms enabled saves energy in (i) instruction scheduling, due to the use of VLIW backends for 61% of blocks, and (ii) the register file and the bypass network: dynamic pipeline narrowing for narrow blocks causes 21% of all μops to execute on a narrow pipe, and dead write elision eliminates 44% of local RF writes and 19% of local bypass network writes.

5. Examining the coarse-grained heterogeneous core (which is the state-of-the-art in heterogeneous core design [35]), we see that it can save energy in both the out-of-order logic (RAT, ROB, and RS) as well as the execution resources (bypass buses and register file) because it is able to use the narrower in-order backend a portion of the time. However, when it is using the out-of-order backend, it cannot improve the energy-efficiency of that backend. Compared to this baseline heterogeneous core, HBA saves additional energy because it can exploit finer-grained heterogeneity.

6. Using a clustered out-of-order backend in the coarse-grained heterogeneous core (as shown in row 4 of Table 2 and as *Coarse, Clustered* in Fig. 5) enables more EPI (and power) reduction than provided by either the clustered out-of-order core or the coarse-grained heterogeneous core [35] alone, mainly due to the reduction of the scheduler power with clustering, as observed above. However, this comes with higher performance degradation than any of the designs (2.8% as seen in row 4 of Table 2). HBA outperforms this coarse-grained, clustered design in IPC, power and EPI, as seen in Table 2 and Fig. 5.

Overall, HBA reduces core energy significantly compared to all baseline designs, including the non-clustered and clustered versions of a state-of-the-art heterogeneous core design [35], by leveraging block atomicity and heterogeneity to execute application code more efficiently.

Finally, we note that these savings are relatively robust to power-modeling assumptions. While we assumed in the above evaluation that leakage comprises 10% of total power (§4), corresponding to a realistic mix of fast, leaky transistors and slower, less leaky transistors, we also report power results for the worst case in our process, 15% leakage. Under this assumption, HBA still reduces average core power by 21.9% and EPI by 21.1% relative to the baseline out-of-order core.

5.2. Performance

5.2.1. Performance of HBA vs. Baselines In addition to energy/power, Table 2 shows normalized average (geometric mean) performance for our baseline and HBA evaluation points. Several major conclusions are in order:

1. The HBA design (row 6) experiences 1.0% performance degradation, on average, compared to the baseline out-of-order core (row 1). The similar performance is a result of equivalent instruction window size and a balance of two major factors: performance gain due to higher available issue width and performance loss due to inter-block communication latency, as we explain in §5.2.2.

2. When all blocks are executed on out-of-order backends only (row 5), the HBA design experiences a 0.4% performance *gain* relative to baseline (row 1), and 1.4% relative to nominal HBA (row 6). We conclude that instruction memoization has some performance penalty (because it sometimes sends blocks to VLIW backends even though dynamic scheduling would be better for the blocks), but that this penalty is modest (1.0%) for the energy reduction it obtains.

3. In comparison to the coarse-grained heterogeneous core design (row 3), HBA experiences approximately the same performance, but much lower average core power. Although both schemes are able to exploit heterogeneity, the HBA design can save more energy because it uses its VLIW (lower-cost) backends only for blocks of code that achieve good performance on these backends. It can thus save energy while maintaining good performance. In contrast, the coarse-grained heterogeneous design must execute the whole program on only one core at a time, and it loses more performance when using the in-order core. It thus uses the in-order core less frequently than HBA uses its VLIW backends.

4. Using a clustered out-of-order backend in the coarse-grained heterogeneous core (row 4) degrades performance over either the clustered or coarse-grained core alone (rows 2,3) due to the additive overheads of clustering [17] and coarse grained heterogeneity [35]. HBA (rows 5,6) has higher performance as well as better power/energy efficiency than this design.

5.2.2. Limit Studies in Performance In order to better understand the factors in HBA's performance and its potential relative to other designs, we show several control and limit studies in Table 3. This table shows a baseline out-of-order design as (i) its issue width is widened and (ii) its fetch and retire width bottlenecks are removed. It also shows HBA (without heterogeneous backends) as its inter-block communication latency is removed and as its fetch width bottleneck is alleviated. We make several major conclusions:

1. Inter-block communication latency penalizes performance. To demonstrate this impact, we evaluate HBA with "instant

inter-block communication" (row 5). In this design, the ideal HBA design (row 5) achieves 6.2% higher performance than the baseline out-of-order (row 1).

2. Against this performance degradation, however, *higher aggregate issue rate* increases HBA's performance. This higher issue rate arises because each block execution backend has its own scheduler and simple ALUs that work independently. In some cases, these independent clusters can extract higher ILP than in the baseline core. This effect is especially visible when inter-block latency is removed, and is responsible for HBA's 6.2% IPC increase above baseline.

3. Higher available issue rate alone cannot account for all of the idealized HBA design's performance: other bottlenecks are also present. To see this, we evaluate a 64-wide out-of-order core with a monolithic scheduler (row 2). Such a design performs only 2.1% better than baseline (row 2), in contrast to 6.2% for idealized HBA (row 5). Because the wide out-of-order core does not provide an upper bound over HBA, we conclude that other effects are present.

4. The remaining advantage of HBA is due to *block-wide dispatch and retire* in HBA: because it tracks precise state only at block boundaries, it is able to achieve high instruction retire throughput when the backend executes a region with high ILP. Allowing for block-wide (or 16 μop-wide) fetch/dispatch/retire in *both* the out-of-order and HBA designs, we observe 23.6% (OoO, row 6) and 23.1% (HBA, row 3) performance above baseline, respectively. Hence, HBA is capable of harnessing nearly all available ILP discovered by an ideal out-of-order design, subject only to inter-block communication latencies and fetch/retire bottlenecks.

Row	Configuration	IPC Δ
	Baselines:	
1	4-wide OoO (Baseline)	—
2	64-wide OoO	+2.1%
3	64-wide OoO, Wide Fetch/Retire	+23.6%
	HBA Variants:	
4	HBA, OoO Only	+0.4%
5	HBA, OoO Only, Instant Inter-block Communication	+6.2%
6	HBA, OoO Only, Instant Inter-block Comm., Wide Fetch	+23.1%

Table 3: Limit studies and control experiments.

5.2.3. Per-Application Performance and Energy Fig. 7 plots the IPC and EPI of HBA across the set of all benchmarks, normalized against the out-of-order baseline core's IPC and EPI. (Fig. 8 shows selected benchmarks in more detail; the full set of data is available in an anonymized report [1]). The left three-quarters of the benchmarks experience some performance degradation relative to the out-of-order core mainly for the reasons described in §5.2.2. The highest performance degradation is 42% for one benchmark that has a high block squash rate.[1] However, all but four benchmarks achieve at least 80% of the baseline performance. Significantly, the rightmost quarter of benchmarks achieve higher performance on HBA than on the out-of-order core. In a few cases, the performance gain is very large (1.79x maximum), due to additional ILP exploited by independent HBA backends. The benchmarks that perform best with HBA are largely those with regular code (e.g., simple, highly-unrolled loops) that can execute independent chunks in each backend. In all except for a few workloads HBA reduces EPI over the baseline.

[1] We found that two types of code performs poorly on HBA: code with hard-to-predict branches, leading to block squashes, and code with long-dependence chains, leading to high inter-block communication.

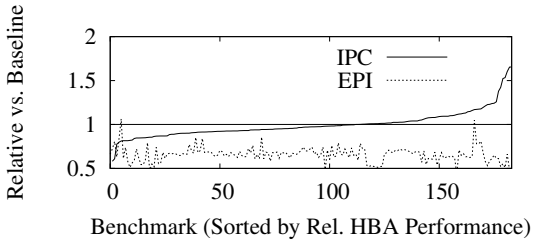

Figure 7: HBA performance and EPI relative to baseline OoO.

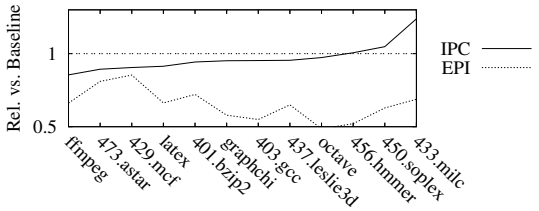

Figure 8: HBA performance and EPI on selected benchmarks.

5.3. Power-Performance Tradeoff Space

Fig. 9 shows multiple HBA and baseline core configurations plotted within the 2D power-performance tradeoff space. Variants of HBA are labeled as "HBA(number of block backends, other options)," with options including *OoO* (out-of-order backends only) and *2-wide* (all block backends are 2-wide). The plot compares HBA configurations against several out-of-order baselines with various window sizes, 1-, 2-, and 4-wide in-order baselines, and several coarse-grained heterogeneous cores. We conclude that (i) HBA's power-performance tradeoff is widely configurable, (ii) HBA is the most energy-efficient design (closest to the bottom-right corner), (iii) HBA enables new points in the power-performance tradeoff space.

5.4. Fine-Grained Heterogeneity

In our initial motivation, we analyzed the behavior of the baseline out-of-order core to demonstrate fine-grained heterogeneity with respect to instruction scheduling. We now verify that this fine-grained heterogeneous behavior is exploited by the block-core design. In order to do so, we observe the stream of retired blocks produced by HBA and form the time-series composed of each block's type (out-of-order or VLIW), and then we examine the *frequency spectrum* of this time-series, i.e., the rate at which the block-type changes when observing execution in program order. Fig. 10 shows the average spectrum obtained by Fourier transform of this time-series over all benchmarks.

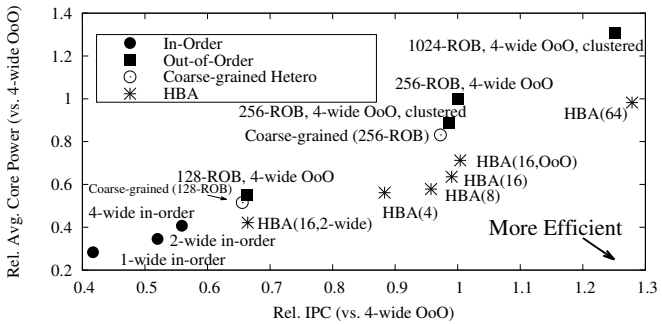

Figure 9: Power-performance tradeoff space of core designs.

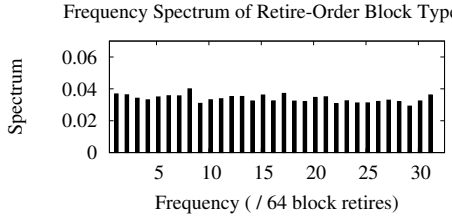

Figure 10: Frequency spectrum of the block type of each retired block, in program order.

As the figure shows, the spectrum is largely "noise," with significant high-frequency energy. Hence, the block type switches frequently when observed in program order. (For example, the rightmost bar indicates occurrences in which each retired block is the opposite type of the last, i.e., OoO and VLIW blocks are interleaved tightly.) This confirms that *fine-grained* intermixing of OoO and VLIW execution occurs in HBA, validating the decision to exploit fine-grained heterogeneity.

Note that unlike the initial motivational result in §2, this analysis records *actual* block types, rather than the earlier *oracular* results that compared every block's instruction schedule to that of its most recent previously-executed instance. The actual mechanism for transitioning between out-of-order and VLIW modes has some hysteresis, such that block types tend to be more stable than if *every* opportunity to re-use a schedule were exploited. This leads to the presence of more low-frequency energy in the spectrum in Fig. 10.

5.5. Symbiotic Out-of-Order and VLIW Execution

Fig. 11 presents the *average block type*. For a given block (out of all blocks executed in all benchmarks), we record its *average type* over all retired instances of that block (so that, e.g., a block that retires 25% of the time after executing on an out-of-order backend has an average type of 25% OoO). We then accumulate a histogram of the average type of all blocks, where each block's contribution is weighted by the number of times it is retired.

As the figure shows, the distribution is very bimodal: most blocks are either *almost always* executed out-of-order, or *almost always* executed VLIW. The mode of a particular block largely depends on whether (i) it contains events that exhibit dynamic (changing) latencies, such as cache misses that occur only sometimes, and/or (ii) its execution timing depends on the preceding blocks and their timing, which may be different for each executed instance of this block. Hence, the execution

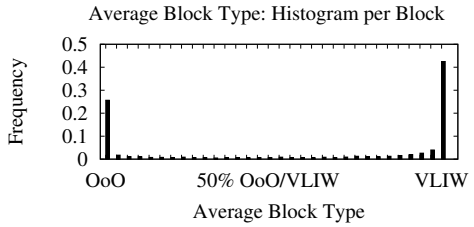

Figure 11: Histogram of average type for each block.

schedule may change, but *whether* it is susceptible to change is itself a consistent property.

Next, we perform a *frequency spectrum* analysis on the block type that a given block takes on each time it is executed, and then average the resulting frequency spectrum over all blocks in all benchmarks. Fig. 12 shows the resulting spectrum. This frequency spectrum differs from the *all-block* spectrum presented in Fig. 10 because we observe each block's behavior relative to *its own* past instances, not relative to the other blocks nearby in program order. As this per-block spectrum shows, most block behavior is low-frequency: in other words, blocks change type relatively infrequently. From this, we conclude that fine-grained heterogeneity occurs because *different blocks* intermix in the program's instruction stream, not because any particular block changes behavior rapidly.

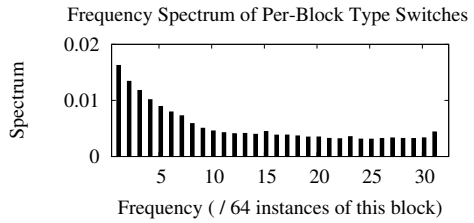

Figure 12: Frequency spectrum of the type of each instance of a given block, averaged over all blocks.

We show a sorted curve of per-benchmark out-of-order vs. VLIW block distribution in Fig. 13. We observe that most benchmarks have a heterogeneous split of block type, with relatively few having either all out-of-order blocks or all VLIW blocks. Interestingly, for a few benchmarks on the left, this is not the case: almost all blocks (> 90%) execute as VLIW. For those benchmarks, that learning one instruction schedule per block is sufficient to capture most of the benefit of out-of-order execution. Finally, a more detailed breakdown of block type by both OoO/VLIW and VLIW width is shown in Table 4. This data shows that dynamic pipeline narrowing captures significant opportunity at least down to a 2-wide configuration (half of the nominal block backend pipeline width).

Type	Out-of-Order	VLIW (4 wide)	VLIW (2 wide)	VLIW (1-wide)
Retired Blocks	38.6%	39.3%	20.1%	2.0%

Table 4: Distribution of block type (detailed).

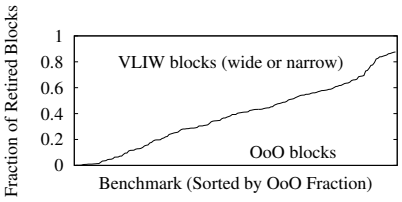

Figure 13: Distribution of block type for all retired blocks.

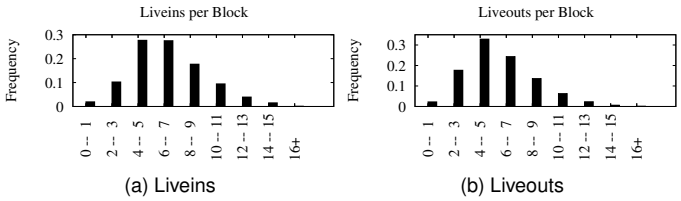

Figure 14: Distribution of liveins and liveouts per block.

5.6. Block Characteristics

Over all benchmarks, the average block has 5.00 liveins and 4.32 liveouts. In contrast to compiler-based approaches to forming atomic blocks [38, 24, 37, 9], HBA does not explicitly attempt to find "good" cuts between blocks that minimize inter-block communication. In addition, the core cannot know whether a value is "dead" past the end of the block. Rather, it must treat every modified architectural register as a live-out. Nevertheless, the number of liveins remains smaller than the total number of sources for all instructions (up to 32 per block) because many values that are consumed are produced locally, within the block. Likewise, the number of liveouts remains smaller than the total number of register destinations (up to 16 per block) because (i) some architectural registers are overwritten within the block and (ii) many μop sequences for macro-instructions write to temporary registers (part of the μISA) which do not need to be communicated between blocks because blocks are cut at macro-instruction boundaries.

5.7. Design Parameter Evaluations

5.7.1. Fine-grained Inter-block Communication Recall that HBA uses a fine-grained livein wakeup mechanism: liveouts are sent out of a block execution backend as soon as they are produced and can wake up dependents in other blocks right away. Although this design choice is more expensive than a simpler alternative design in which liveouts are written all at once when a block completes, and later blocks read liveins all at once, the design is important to performance because it prevents false dataflow dependences when unrelated dependence chains share one block. To demonstrate the performance impact, we implement two alternate design options, *all-at-once liveins* and *all-at-once liveouts*, that enforce that a block receives all liveins before beginning execution, and/or sends all liveouts only after completing execution, respectively. We then evaluate HBA with all four combinations of options. We find that relative to HBA without either option, enforcing only all-at-once liveins results in 64% performance loss. Enforcing only all-at-once liveouts results in 55% performance loss. Combining both restrictions results in 69% performance loss. Hence, we conclude that the fine-grained inter-block communication in HBA is necessary to maintain good performance.

5.7.2. Ideal Backend Choice When HBA sends a block to a VLIW backend, it is *predicting* that the recorded schedule for that block will be accurate. We perform an ideal study to understand the accuracy of this prediction. To perform this study, we execute all blocks on out-of-order backends, but as soon as a block completes, we compare its schedule to that of the block's previous instance. If the block's schedule is the same, we assume that a schedule memoization mechanism with "perfect" prediction could have sent the block to a VLIW backend. We then alter the power model event counts post-mortem as if the block had executed on a VLIW backend instead. We also observe the maximum issue width and we count how many bypass broadcasts and local register writes were actually necessary, thus accounting for narrow-pipe and dynamic event elision savings. Hence, performance is identical to the out-of-order-only design, but power is reduced as much as possible given the identical performance.

With this study, we find that power is reduced by 31.5% relative to the baseline (recall that OoO-only HBA reduces power by 28.7% and HBA with heterogeneity reduces power by 36.4%). These "ideal" savings are *less* than HBA with real VLIW backends because no performance is lost. In contrast, the real HBA schedule memoization mechanism sometimes uses a VLIW backend when an out-of-order backend was actually necessary, reducing performance but saving additional power.

5.7.3. Inter-Backend Communication Latency Our primary evaluations assume a 2-cycle additional latency for all values communicated between backends. We find that performance is highly sensitive to this communication latency: when increased to 3 cycles, performance degrades by 4.0%. Conversely, when latency is decreased to 0 cycles over a monolithic instruction window ("instant" communication), performance increases by 6.2%. Hence, design of the global register file and liveout wakeup logic is critical for good performance.

5.7.4. Block Info Cache Fig. 15 shows performance sensitivity to the block info cache size for HBA. The arrow indicates the design point (256 blocks) at which our main evaluations are performed. Although performance degrades with smaller cache sizes, the cache could be reduced to only 64 blocks while retaining 91% of performance at 256 blocks.

Recall that the block info cache is an auxiliary store *aside* the main instruction cache that contains only metadata (such as VLIW schedules), not the full instruction sequences for each block. Hence, a small block info cache does not impact performance as much as a small instruction cache would: in case of a miss, the block front-end simply fetches instructions and forms a new block, which executes on an out-of-order backend by default (because no VLIW schedule or other properties are known). This behavior actually causes a slight *dip* in performance between 512 and 1024-block cache sizes: a larger block info cache retains more information and hence sends more blocks to VLIW backends, which saves power but loses some performance.

5.7.5. Full Block Cache Recall that HBA uses a *block info cache* that stores only *metadata* for each code block, without

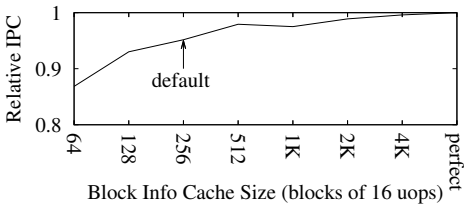

Figure 15: Sensitivity to block info cache size.

storing the block's instructions. This design saves space (a cache of 1024 blocks requires 256 KB of storage), but cannot achieve the higher fetch rates that are possible if pre-formed blocks are fetched [24, 47].

As an alternative design choice, full blocks can be stored in the lower levels of the cache hierarchy (L2 or LLC). A small L1 block cache (which stores complete blocks, not only metadata) can be added so that the L2 need not sustain the full fetch bandwidth. We perform a study that stores blocks in the L2 cache by hashing block identifiers to L2 sets. Each block of 16 μops, with 256 bits per μop, takes 256 bytes (4 cache blocks) of storage. Inserting a code block evicts the corresponding data blocks and vice versa. We add a small L1 block cache of 64 blocks and assume a 20 cycle L1 block cache miss latency.

For benchmarks that require high fetch IPC and have lower data cache footprint, such a design might be an appropriate tradeoff. However, we find that this alternative design with only out-of-order backends suffers a 21.3% performance degradation (at 36.5% average core power reduction). Although increased fetch rate allows higher utilization of the execution resources, the high latency penalty of block cache misses to L2 as well as the data-cache capacity impact both reduce performance significantly. Additional L2 accesses also penalize energy savings relative to the nominal HBA design. Hence, we conclude that our block metadata-based approach is the most efficient design choice for our parameters. If full blocks must be stored for other reasons – for example, more radical code transformations for specialized backends – then a block cache could be used for these blocks.

5.7.6. Window Scalability We show that HBA can enable more efficient scaling to a large instruction window than a baseline core design. Table 5 shows normalized performance (IPC) and average core power for two design points: a moderately-sized instruction window of 256 μops, which we use in our main evaluations, and a large instruction window of 1024 μops. Such a large instruction window allows for higher memory latency tolerance, which improves performance. At each design point we show an out-of-order core and HBA.

Two conclusions are in order. First, for the large instruction window baseline, we show a monolithic large-window design as well as a clustered design. The design with a monolithic instruction scheduler and register file has approximately 3.7x the core power of the baseline (moderate-window) design, and likely would not meet timing at the same clock frequency because of its enlarged critical structures. The clustered design requires only a (relatively) reasonable increase of 31% average core power over the moderate-window baseline. This comparison indicates that segmenting the window via clustered scheduling and register files is necessary to achieve reasonable scalability.

Second, HBA achieves the same or better performance scalability as the large-window design. At the large window size, most of the additional performance comes from the ability to tolerate very long latencies, not necessarily from fine-grained communication between any two μops far apart in the window.

Design	Rel. IPC	Rel. Power
Moderate (256-μop window) designs:		
Baseline, 256-ROB	1.00	1.00
Baseline Clustered, 256-ROB	0.99	0.88
HBA, 16 blocks, 16μops/block	0.99	0.64
Large (1024-entry window) designs:		
Baseline, 1024-ROB, 256-RS (monolithic)	1.23	3.70
Baseline, 1024-ROB, 256-RS (clustered)	1.25	1.31
HBA, 64 blocks, 16μops/block	1.28	0.98

Table 5: Window-size scaling in baseline and HBA cores.

5.7.7. Block Size We evaluate a variant of HBA that forms and executes blocks of up to 64 μops (four times the nominal 16 μops/block). When configured with a window size of 16 blocks (hence 1024 μops total), this core achieves 20.2% performance above the baseline out-of-order core, with a relative core power of 0.94 (6.1% reduction). Note that the HBA core with an equivalent window size but 64 blocks of 16 μops/block achieves a 27.9% performance gain above baseline and has a 1.8% power reduction. Larger blocks, on average, are less efficient for three reasons: (i) they are cut early during formation more frequently, leading to reduced window utilization; (ii) they are more likely to experience intra-block branch mispredictions, leading to more wasted speculative work; and (iii) the window structures that scale superlinearly, such as instruction schedulers in out-of-order block backends, have lower efficiency when sized for larger blocks.

5.7.8. Core Area Like previous approaches that propose heterogeneous designs (e.g., [57, 23, 29]), HBA optimizes for a heavily energy/power-constrained future, targeting designs in which energy and power are performance limiters [16]. Also like other heterogeneous designs that attempt to reduce power/energy through increased specialization, the power benefits of HBA come at the cost of additional chip area. To evaluate the area overhead of HBA, we use McPAT, which is able to estimate area by estimating growth in key structures in the core. Using this approach, we estimate that the core area increases by 62.5% over the baseline out-of-order core. For comparison, our "coarse-grained heterogeneous" baseline was estimated to have 20% area overhead [35] (but provides less energy-efficiency than HBA).

Although HBA comes with significant area overhead, it is worth noting that the cores occupy a relatively small area in a typical mobile SoC (e.g., 17% in Apple's A7 [13]). Hence, the investment of additional area to improve energy efficiency could be reasonable. By trading off die area for a microarchitecture that can specialize, HBA can: 1) achieve significant energy savings that were not easily attainable in a non-heterogeneous core, 2) enable new power-performance tradeoff points in core

design vs. many different designs (§5.3).

6. Related Work

To our knowledge, HBA is the first design that leverages block-level atomicity to exploit heterogeneity at a *fine granularity*. Other works have combined multiple cores or backends and migrated execution between them (*coarse-grained heterogeneity*) [35, 56, 4, 20, 7, 12], or proposed ways to scale a single microarchitecture up and down dynamically by power-gating components or combining cores together [25, 28, 54, 27]. Past works have also used atomicity in other ways to (i) simplify the core microarchitecture, (ii) enable better scalability and/or performance, or (iii) enable optimizations or other code transformations.

Heterogeneous cores: Several works have examined the use of either multiple heterogeneous cores [4, 56, 22, 20, 7, 12], one core with multiple heterogeneous backends [35], or a core with variable parameters [5, 22] in order to adapt to the running application at a coarse granularity for better energy efficiency. We quantitatively compared to a coarse-grained heterogeneous approach [35] in §5 and showed that although coarse-grained designs can achieve good energy-efficiency, HBA does better by exploiting much finer-grained heterogeneity. However, combining these two levels of heterogeneous core design might lead to further gains and we leave this to future work.

Several prior works combine a "cold pipeline" and "hot pipeline" that execute infrequent (cold) and frequent (hot) code traces respectively [46, 8, 23]. PARROT [46] captures hot code traces and performs optimizations on them to produce code for an optimized VLIW engine. Gupta et al. propose BERET [23], which adds an auxiliary co-processor that executes data-flow subgraphs for hot code when those subgraphs fit in the co-processor. These designs are superficially similar to the use of VLIW backends in HBA. However, HBA differs in that its VLIW backends are general-purpose and its mechanism to prepare code for execution on a VLIW backend is very simple, requiring no compiler-based or runtime analysis or optimization. As a result, HBA can execute a large fraction of blocks (two-thirds on average) on its VLIW backends.

Turboscalar [8] uses two pipelines, one which is "short and wide" and the other which is "deep and narrow," that together share a common pool of functional units. The design captures hot code traces and stores them in predecoded form for the short and wide pipeline to execute. This heterogeneous design memoizes information as HBA does, but for a different goal: Turboscalar targets performance through a faster frontend while HBA targets energy through a more efficient backend.

Nair and Hopkins [41] propose DIF (Dynamic Instruction Formation), which caches pre-scheduled groups of instructions to execute on a VLIW-like substrate, reducing complexity and improving efficiency. HBA has two key differences: first, HBA's goal is to provide a *general substrate* for heterogeneity enabled by considering atomic blocks, whereas DIF is a specific proposal to use a VLIW engine in a general-purpose machine without requiring ISA/compiler support. Second, DIF does not use an out-of-order engine but only a simple "primary engine" alongside its VLIW engine, and envisions that most code should execute on the VLIW engine.

Atomic block-based cores: Melvin et al. [39, 38], and later Hao et al. [24] and Sprangle et al. [51], propose a core design in which the compiler provides atomic blocks at the ISA level. These works note multiple advantages of using atomic blocks: the core has a higher instruction fetch rate, and can also use a small local register file to reduce register pressure on a global register file [51]. HBA forms blocks in hardware and uses the notion of atomicity to exploit heterogeneity and reduce energy consumption.

Rotenberg et al. [47] propose the Trace Processor, which forms *traces* (atomic blocks), stores them in a trace cache, and dispatches them to decoupled execution units. HBA has a similar overall design; however, a Trace Processor does not leverage any heterogeneity or specialization.

Patel et al. [43] propose rePLay, which forms *frames* (atomic blocks), stores these in a frame cache, and performs compiler-like optimizations on the frames in order to improve performance. HBA uses a similar insight that atomicity can allow code transformations to exploit heterogeneous backends.

Ceze et al. [11] propose a core that commits instructions in bulk (in atomic blocks) as a way to simplify cache coherence. This use of atomicity is orthogonal to ours, but could be combined to achieve additional benefits.

The TRIPS processor [40, 10] uses atomic blocks consisting of explicit dataflow graphs of instructions statically scheduled onto a grid of execution units. This microarchitecture is very different from HBA, both designs exploit atomicity to enable extensive code transformations while maintaining coarse-grained precise state.

The POWER4 processor [55] performs "instruction group formation," treating several fetched instructions as a group for purposes of tracking throughout the pipeline. Although these groups are superficially similar to atomic blocks, they are not saved, specialized, or reused, as our atomic blocks are.

Instruction scheduling memoization: McFarlin et al. [36] demonstrate that most of out-of-order execution's benefit comes from its ability to use a speculative execution schedule, but not necessarily a *different* schedule for every instance of an instruction sequence. This is similar to our observation that many static regions of code have stable instruction schedules. However, HBA additionally shows that that *heterogeneity* exists in this scheduling behavior and can be exploited.

Banerjia et al. [6] propose MPS (Miss-Path Scheduling), which pre-schedules instructions and places them in a "Schedule Cache." Although this design reuses a schedule many times, as HBA does, its design is very different.

Michaud and Seznec [2] propose pre-scheduling instructions after decode but prior to placing them in the out-of-order scheduling window. This prescheduling makes a smaller scheduling window more effective, but does not save energy by reusing the same schedule multiple times, as HBA does.

VLIW: HBA uses VLIW backends. Traditional VLIW [18] *statically* expresses parallelism in an instruction stream. While a VLIW design can achieve better energy efficiency in principle, because it need not perform any instruction scheduling at

runtime, a conventional VLIW design is hindered by the difficulty of static scheduling. HBA works around this limitation by performing instruction scheduling *pseudo-statically*, based on the runtime behavior of one dynamic instance of a small region of code. That schedule can then be re-used, saving energy as in a conventional VLIW machine.

7. Conclusion

Future core designs must adapt to application code in order to provide good performance and good energy efficiency. In this work, we observe that applications are heterogeneous not only at the coarse grain (i.e., between phases or between separate applications), but also at the fine grain, i.e., between adjacent blocks of code. We present the *Heterogeneous Block Architecture (HBA)*, a new core design that (i) breaks a single application's code into blocks and (ii) executes each block of code on the most appropriate *execution backend* out of a pool of heterogeneous backend designs. HBA uses *block atomicity* in order to allow each block of code to be specialized appropriately to the right backend. In our initial design, we evaluate an HBA core with out-of-order and VLIW (statically scheduled) execution backends, exploiting scheduling heterogeneity.

Our evaluations of the HBA design against a comprehensive set of baseline out-of-order and in-order cores and a state-of-the-art coarse-grained heterogeneous design, over a large set of benchmarks, demonstrate that HBA is the most energy-efficient design. We observe that HBA achieves a 36.4% average core power reduction compared to a conventional 4-wide out-of-order core while degrading performance by only 1%. We conclude that HBA provides energy-efficient and high-performance execution, enables new power-performance tradeoff points in core design, and provides a flexible execution substrate for exploiting fine-grained heterogeneity in future core designs.

References

[1] "Anonymous HBA technical report," https://www.dropbox.com/s/xx2mq890vuwdkem/appendix.pdf.

[2] "Data-flow prescheduling for large instruction windows in out-of-order processors," *HPCA-7*, 2001.

[3] M. Annavaram, E. Grochowski, and J. Shen, "Mitigating Amdahl's law through EPI throttling," *ISCA-32*, 2005.

[4] ARM Ltd., "White paper: Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7," Sept 2011.

[5] R. Bahar and S. Manne, "Power and energy reduction via pipeline balancing," *ISCA-28*, 2001.

[6] S. Banerjia, S. Sathaye, K. Menezes, and T. Conte, "MPS: Miss-path scheduling for multiple-issue processors," *IEEE Trans. Comp.*, vol. 47, pp. pp. 1382 – 1397, Dec. 1998.

[7] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," *JILP*, June 2008.

[8] B. Black and J. Shen, "Turboscalar: A high frequency high ipc microarchitecture," *Workshop on Complexity-Effective Design*, 2000.

[9] S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, "Multiscalar processors," *ISCA*, 1995.

[10] D. Burger, S. Keckler, K. KcKinley, M. Dahlin, L. John, C. Moore, J. Burrill, R. McDonald, W. Yoder, and TRIPS Team, "Scaling to the end of silicon with EDGE architectures," *IEEE Computer*, vol. 37, pp. 44 – 55, Jul. 2004.

[11] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: bulk enforcement of sequential consistency," *ISCA-34*, 2007.

[12] J. Chen and L. John, "Efficient program scheduling for heterogeneous multi-core architectures," *DAC-46*, 2009.

[13] ChipWorks, "Inside the Apple A7 from the iPhone 5s – Updated," http://www.chipworks.com/en/technical-competitive-analysis/resources/blog/inside-the-a7/.

[14] D. Deleganes, J. Douglas, B. Kommandur, and M. Patrya, "Designing a 3 GHz, 130 nm, Intel Pentium 4 processor," in *IEEE Symposium on VLSI Circuits*, 2002.

[15] P. Denning, "Working sets past and present," *IEEE Trans. Soft. Eng.*, vol. SE-6, pp. 64 – 84, Jan. 1980.

[16] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *ISCA-38*, 2011.

[17] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic, "The multicluster architecture: Reducing cycle time through partitioning," *MICRO-30*, 1997.

[18] J. Fisher, "Very long instruction word architectures and the ELI-512," *ISCA-10*, 1983.

[19] P. Flatresse, G. Cesana, and X. Cauchy, "Planar fully depleted silicon technology to design competitive SOC at 28nm and beyond," STMicroelectronics White Paper, Feb. 2012.

[20] S. Ghiasi, T. Keller, and F. Rawson, "Scheduling for heterogeneous processors in server systems," *Conf. Computing Frontiers*, 2005.

[21] M. Goshima, K. Nishino, Y. Nakashima, S. Mori, T. Kitamura, and S. Tomita, "A high-speed dynamic instruction scheduling scheme for superscalar processors," *MICRO-34*, 2001.

[22] E. Grochowski, R. Ronen, J. Shen, and H. Wang, "Best of both latency and throughput," *ICCD*, 2004.

[23] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," *MICRO-44*, 2011.

[24] E. Hao, P.-Y. Chang, M. Evers, and Y. N. Patt, "Increasing the instruction fetch rate via block-structured instruction set architectures," *IJPP*, vol. 26, pp. 449–478, August 1998.

[25] E. İpek, M. Kirman, N. Kirman, and J. Martinez, "Core fusion: Accommodating software diversity in chip multiprocessors," *ISCA-34*, 2007.

[26] J. Joao, M. Suleman, O. Mutlu, and Y. Patt, "Bottleneck identification and scheduling in multithreaded applications," *ASPLOS-XVII*, 2012.

[27] Khubaib, M. Suleman, M. Hashemi, C. Wilkerson, and Y. Patt, "MorphCore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP," *MICRO-45*, 2012.

[28] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, "Composable lightweight processors," *MICRO-40*, 2007.

[29] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen, "Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction," *MICRO-36*, 2003.

[30] R. Kumar, N. P. Jouppi, and D. M. Tullsen, "Conjoined-core chip multiprocessing," *MICRO-37*, 2004.

[31] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," *OSDI-10*, 2012.

[32] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," *MICRO-30*, 1997.

[33] S. Li, J. Ahn, J. Brockman, and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," *MICRO-42*, 2009.

[34] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.

[35] A. Lukefahr, S. Padmanabha, R. Das, F. Sleiman, R. Dreslinski, T. Wenisch, and S. Mahlke, "Composite cores: Pushing heterogeneity into a core," *MICRO-45*, 2012.

[36] D. McFarlin, C. Tucker, and C. Zilles, "Discerning the dominant out-of-order performance advantage: is it dynamism or speculation?" *ASPLOS-18*, 2013.

[37] W. mei W. Hwu *et al.*, "The superblock: An effective technique for vliw and superscalar compilation," *J. Supercomputing*, vol. 7, pp. 229–248, 1993.

[38] S. Melvin and Y. Patt, "Enhancing instruction scheduling with a block-structured ISA," *Int. J. Parallel Program.*, June 1995.

[39] S. W. Melvin, M. Shebanow, and Y. Patt, "Hardware support for large atomic units in dynamically scheduled machines," in *MICRO*, 1988.

[40] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler, "A design space evaluation of grid processor architectures," *MICRO-34*, 2001.

[41] R. Nair and M. Hopkins, "Exploiting instruction level parallelism in processors by caching scheduled groups," *ISCA-24*, 1997.

[42] S. Patel, "Trace cache design for wide issue superscalar processors," Ph.D. dissertation, 1999.

[43] S. Patel and S. Lumetta, "rePLay: a hardware framework for dynamic optimization," *IEEE TC*, June 2001.

[44] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation," *MICRO-37*, 2004.

[45] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren, "Supporting dynamic data structures on distributed memory machines," *ACM Trans. Prog. Lang. Sys.*, vol. 17, pp. 233 – 263, Mar. 1995.

[46] R. Rosner, Y. Almog, M. Moffie, N. Schwartz, and A. Mendelson, "Power awareness through selective dynamically optimized traces," *ISCA-31*, 2004.

[47] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace processors," *MICRO-30*, 1997.

[48] P. Sassone, J. Rupley, E. Brekelbaum, G. Loh, and B. Black, "Matrix scheduler reloaded," *ISCA-34*, 2007.

[49] A. Seznec, "A 64 Kbytes ISL-TAGE branch predictor," *JWAC-2*, 2011.

[50] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *ASPLOS-10*, 2002.

[51] E. Sprangle and Y. Patt, "Facilitating superscalar processing via a combined static/dynamic register renaming scheme," *MICRO-27*, 1994.

[52] Standard Performance Evaluation Corporation, "SPEC CPU2006," http://www.spec.org/cpu2006.

[53] M. Suleman, O. Mutlu, M. Qureshi, and Y. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," *ASPLOS-XIV*, 2009.

[54] D. Tarjan, M. Boyer, and K. Skadron, "Federation: Repurposing scalar cores for out-of-order instruction issue," *DAC*, 2008.

[55] J. Tendler, J. Dodson, J. Fields, H. Le, and B. Sinharoy, "POWER4 system microarchitecture," *IBM J. R&D*, vol. 46, pp. pp. 5–25, Jan. 2002.

[56] K. van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through Performance Impact Estimation (PIE)," *ISCA-39*, 2012.

[57] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. Taylor, "Conservation cores: reducing the energy of mature computations," *ASPLOS-XV*, 2010.

[58] M. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," *ISPASS*, 2007.