The Heterogeneous Block Architecture

Chris Fallin* Chris Wilkerson† Onur Mutlu*
cfallin@c1f.net chris.wilkerson@intel.com onur@cmu.edu
*Carnegie Mellon University †Intel Corporation

This paper makes two 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 fine-grained 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 core design*, called the *heterogeneous block architecture (HBA)*, that combines heterogeneous execution backends into one core. HBA breaks the program into blocks of code, determines the best backend for each block, and *specializes* the block for that backend. As an example HBA design, we combine out-of-order, VLIW, and in-order backends, using simple heuristics to choose backends for different dynamic instruction blocks. Our extensive evaluations compare this example HBA design 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 it provides significantly better energy efficiency than all designs at similar performance.

I. Introduction

General-purpose processor core design faces two competing goals. First, a core should provide high *single-thread (serial) performance*. This is important for many algorithms and for any application with serialized code sections [1, 2, 15, 28, 29, 59]. Second, a core should provide high *energy efficiency*. Energy/power consumption is a primary limiter of system performance and scalability, both in large-scale data centers [17] and in consumer devices [22, 41].

Unfortunately, it is difficult to achieve both high performance and high energy efficiency at the same time: no single core microarchitecture is the best design for both metrics for all programs or program phases. 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 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.

Designing a good general-purpose core is difficult because code is *heterogeneous* at multiple levels: each program has different characteristics, and a single program has different characteristics in different regions of its code. To exploit this diversity, past works have proposed *core-level heterogeneity*. These heterogeneous designs either combine multiple separate cores (e.g., [2, 3, 6, 10, 20, 23, 28, 33, 59, 61]), or combine an in-order and an out-of-order pipeline with a shared frontend in a single core [37]. 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 exploited heterogeneity at the coarser granularity of thousands of instructions: e.g., 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 [37, 61]. As we will show, at a much finer granularity (of tens of instructions), adjacent blocks of code often have different properties. For example, one block of code might have a consistent instruction schedule across its dynamic execution instances in an OoO machine, whereas a neighboring block might have different execution schedules at different times depending on cache miss behavior of load instructions in the block. Such behavior suggests the use of both dynamic and static schedulers within a *single* core, perhaps even simultaneously for different instructions in flight, so that each instance of a block is executed using the most efficient instruction scheduling mechanism. Migration of execution between separate cores or pipelines cannot easily exploit this fine-grained heterogeneity in code behavior.

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*, including functional units, local storage, and some form of instruction scheduling. To exploit fine-grained heterogeneity, a core will need to (i) have execution backends of multiple types, and (ii) *specialize* pieces of code for each backend. By enforcing *atomicity*, or the property that a block of code either executes *as a whole or not at all*, the core can freely analyze and *morph* this block of code to fit a particular backend (e.g., atomicity allows the core to reorder instructions freely within the block). Atomic block-based design allows execution backends to operate independently using a well-defined interface (liveins/liveouts) between blocks.

Based on these two observations, we propose a *fine-grained heterogeneous core* that dynamically forms code into blocks, *specializes* those blocks to execute on the most appropriate type of execution backend, and executes blocks on the various backends. This core design serves as a general substrate for fine-grained heterogeneity that can combine many different types of execution backends. As an initial example design, this paper describes and evaluates a core which includes *out-of-order*, *VLIW*, and *in-order* execution backends, and logic to assign each block to a backend. Our concrete design initially 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 instruction-level parallelism, 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 in flight.

This paper makes four major contributions:

1. It introduces the concept and design of the *heterogeneous block architecture (HBA).* HBA exploits the notions of fine-grained heterogeneity, atomic blocks, and block-based instruction scheduling/execution in a synergistic manner to adapt each piece of code to the execution backend it is best fit to execute on. (§II and §III)

2. It introduces the implementation of a new *fine-grained heterogeneous core*, an example HBA design, that 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, with the goal of maximizing energy efficiency while maintaining high performance. (§III-C3 and §III-D)

3. It provides simple mechanisms that enable a block of code to be switched between VLIW/in-order and out-of-order execution backends. These mechanisms do not require any support at compile time; they use dynamic heuristics and instruction schedules, and form blocks dynamically. (§III-D)

4. It extensively evaluates an example HBA design in comparison to four baselines (out-of-order, clustered [18], coarse-grained heterogeneous [37], and clustered coarse-grained), showing higher energy efficiency than all previous designs across a wide variety of workloads

(§V). Our design reduces average core power by 36.4% with 1% performance loss over the baseline. We show that HBA provides a flexible substrate for future heterogeneous designs, enabling new power-performance tradeoff points in core design (§V-C).

II. Motivation: Fine-Grained Heterogeneity

Our first major observation is that **applications have fine-grained heterogeneity**, i.e., heterogeneity across regions of tens or hundreds of instructions. This heterogeneity is distinct from larger program phases [13, 56] 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 particular instructions or dataflow within a single task or operation.

Fig. 1 pictorially represents this distinction. The left half 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, we see *fine-grained* heterogeneity. The right half of Fig. 1 depicts three regions of instructions within the coarse-grained phase. In the first region of instructions, three of the four operations are independent and can issue in parallel, and all instructions have constant, statically-known latencies. Hence, this region has high ILP and a stable (unchanging) dynamic instruction schedule. The second region also has high ILP, but has a variable schedule due to intermittent cache misses. Finally, the third region has low ILP due to a dependence chain. Overall, each small code region within this single "regular floating point" phase has different properties, arising solely from variations in instruction dependences and latencies. Each such region thus benefits from different core features.

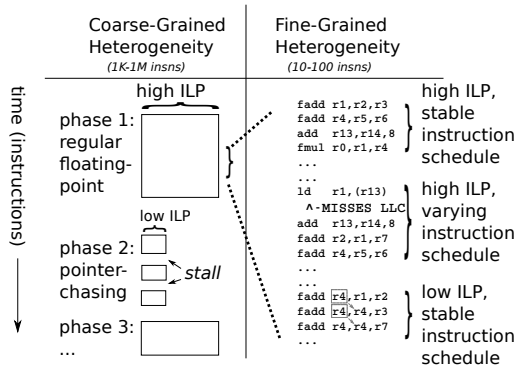


Fig. 1: Coarse-grained vs. fine-grained heterogeneity.

To motivate that adjacent regions of code may have different properties that can be exploited, we show the existence of one such 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 show that this property varies greatly between different nearby regions of code (hence, is *fine-grained*). To do this, we analyze the behavior of 268 workloads on a 4-wide superscalar out-of-order core.[1] We observe the retired instruction stream and group retired instructions into *chunks* of up to 16 μops. Chunks are broken at certain boundaries according to the heuristics in §III-C1. 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 before. These annotations, called "chunk types," indicate the extent to which each chunk has a stable schedule.

Fig. 2 shows the fraction of the "same" and "different" chunk types, per benchmark, as a sorted curve. Two observations are in order. First, many chunks repeat instruction schedules in their previous execution (especially in workloads to the left of the graph). Hence, there is significant opportunity to reuse past instruction scheduling order (as also noted by past work [44]). Second, there are many applications (in the center of the plot) that exhibit a *mix* of behavior: between 20% and 80% of retired chunks exhibit stable schedules. Hence, individual applications often have *heterogeneous* instruction schedule stability across different regions of code. This observation motivates a core design that can reuse instruction schedules for some code and dynamically schedule other code.

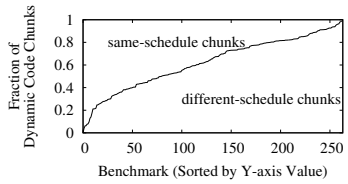


Fig. 2: Fraction of chunks in the instruction stream that have a different schedule than their previous instance.

Moreover, we observe that this heterogeneity exists between *nearby* chunks in the dynamic instruction stream, i.e., there is *fine-grained heterogeneity*. We observe the sequence of chunk types in retire order and group chunks into *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 is often different even between temporally-adjacent static regions of code, indicating *fine-grained heterogeneity*. Fine-grained heterogeneity exists *within* program phases, and is thus distinct from coarse-grained (inter-program/inter-phase) heterogeneity exploited by past works in heterogeneous core/multicore design. Instead, it motivates a new approach: a single core that can execute nearby chunks of instructions with different mechanisms best suited for each dynamic chunk. Our goal in this paper is to provide such a framework for general-purpose core design.

III. HBA: Principles and Example Design

Based on our observations, we introduce our new design, *HBA*.

A. High-Level Overview

Key Idea #1: Build a core that executes fine-grained blocks of code on heterogeneous backends. As shown in §II, application code is heterogeneous at a fine granularity. To exploit this, we build a core that contains multiple different execution backends *within a single core*. The core groups the application's instructions into chunks (called *blocks*) and determines the best type of execution backend for each block. Multiple execution backends (including multiple of the same type) can be active simultaneously, executing different blocks of code, and these backends communicate with each other 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 must be considered as a *unit*, isolated from the rest of the program. Our second key idea is to require **atomicity** of each block: the core commits all results of the block at once or not at all. Atomicity guarantees the core will always handle an entire block at once, allowing the use of backends that leverage code properties extracted once over the entire block (e.g., by reordering or rewriting instructions) to adapt the block to a particular backend. Atomicity thus *enables the core to exploit fine-grained heterogeneous backends*.

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. Most of the performance of out-of-order execution is retained at much lower energy (as shown in §V).

[1] See §IV for our methodology. Later results use a representative subset of 184 of these 268 workloads.

B. Atomicity, Liveins, Liveouts

We briefly describe terms that are important to understanding our design. First, **atomicity** of a block means that a block either completes execution and commits all its results, or none at all. 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.

C. HBA Design

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

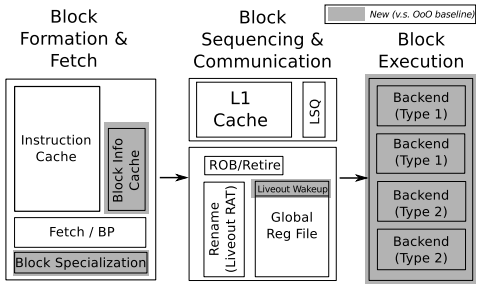


Fig. 3: HBA (Heterogeneous Block Architecture) overview.

1) Block Formation and Fetch: HBA core forms 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 [45, 52]. The block info cache stores information that the core has discovered about the block.[2] This information depends on the block type: for example, for a block that executes on a VLIW backend, the information 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 it (i) reaches a maximum length (16 instructions by default), (ii) ends at an indirect branch, or (iii) 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 [46].

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 on 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 sends it to an appropriate execution backend. Each execution backend executes one block at a time until all operations within the 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. In the concrete design point that we evaluate, there are 16 backends, each of which can execute in OoO or VLIW/in-order mode, so there is only one such free-list.

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. There are two types of branch mispredicts: *intra-block*, due to a conditional branch in the middle of a block, and *inter-block*, due to a branch 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 additionally squash the block that contains the mispredicted branch (due to block-level atomicity) and restart the frontend from the block at the same fetch PC but with a *different internal branch-path*. Finally, the block sequencing logic handles *exceptions* by squashing the excepting block and executing in a special single-instruction block mode to reach the exception point.

Global Registers and Liveout-Livein Communication: Blocks executing on different 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. Thus, values are communicated from producers to consumers as soon as the values become available, and blocks begin executing as soon as any of their instructions has the necessary liveins (avoiding performance loss that would occur if a block were to wait for all liveins first).

To support this fine-grained livein 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 only the consuming backends. This structure is similar to a bit-matrix scheduler [21]. Note that values produced and consumed internally within a block are never communicated nor written to the global register file.

3) Block Execution: When a block is sent to a block execution backend, the backend executes the block in its specialized resources. 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, (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 one.

In our example HBA design, we implement two types of block execution backend: an out-of-order backend and a VLIW/in-order backend. Both types 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. The core will have several such backends (i.e., *not* simply one of each type); in general, an HBA core could contain an arbitrary pool of backends of many different types.

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.

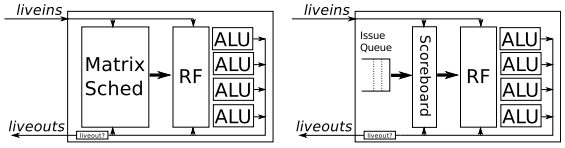
[2]The use of a block *info* cache in parallel with an instruction cache, rather than a full trace cache [45, 52], allows the HBA core to approximate the *best of both worlds*: it achieves the space efficiency of the instruction cache while retaining the learned information about code blocks.

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 waiting in skid buffers to retry). Sharing these units between all execution backends amortizes these units' cost [34].

Memory operations: Execution backends share the L1 cache, the load/store queue (LSQ), and the load/store pipelines. The use of blocks is orthogonal to both the correctness and performance aspects of the memory subsystem: a block preserves memory operation ordering within itself, and allocates loads/stores into the LSQ in original program order. Because our core design achieves similar performance to the baseline core, as we show later, the same memory pipeline throughput as baseline is sufficient. Memory disambiguation does not interact with block atomicity; if a load requires replay, it is sent back to its execution unit as in the baseline.

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, 54]. 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.



(a) Out-of-order backend (b) VLIW backend

Fig. 4: Block execution backend designs.

VLIW execution backend (Fig. 4b): Unlike the out-of-order backend, the VLIW backend has no out-of-order 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 instructions are ready. The scoreboard implements a stall-on-use policy for long-latency operations such as cache-missing loads and operations on the shared FPU.

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 OoO backend to form bundles dynamically at runtime, as we now describe.

D. Combining Out-of-Order and VLIW Execution

To combine out-of-order and VLIW block execution backends, we propose *memoized scheduling*. Memoized scheduling exploits the observation (seen in §II) that blocks often exhibit *schedule stability*. The key idea is to first use an out-of-order execution backend to execute a block and observe its schedule stability. Each time the block executes on this 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, 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 the 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 as-is. Thus, the schedule is *recorded and replayed*, or *memoized*.

If the block's schedule remains stable, subsequent executions of the block on the VLIW backend will obtain the same performance as if the out-of-order backend were used, while saving the energy that instruction scheduling would have consumed. However, if the schedule becomes unstable (e.g., due to changing cache-miss behavior or livein timing), subsequent executions may experience *false stalls*, or cycles in which a VLIW bundle stalls because some of its instructions are not ready to execute, but at least one of the contained instructions is ready and could have executed if it were not bundled. These stalls result in performance loss compared to execution on an out-of-order backend. To minimize this potential loss, the VLIW backend monitors false stall cycles. If the number of such cycles for each block (as a ratio of all execution cycles for that block) exceeds a threshold (5% in our evaluations), the memoized schedule is discarded and the block executes on an out-of-order backend next time it is dispatched.

Unified OoO/VLIW Backend: We observe 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 use a single unified backend that simply turns off its out-of-order scheduling logic (bit-matrix) when it is executing in VLIW mode. (MorphCore [30] exploits a similar observation to combine in-order and out-of-order scheduling.)

E. Reducing Execution Power

We reduce execution power on the VLIW backend in two ways.

Dynamic Pipeline Narrowing: We observe many blocks cannot utilize the full width of a VLIW backend. To exploit this, the VLIW block formation process records the maximum bundle width across all VLIW bundles of a block. When a VLIW backend executes a block, it dynamically narrows its issue width to this maximum, saving static and dynamic energy, similar to [27]. (An "in-order" backend is simply a VLIW backend that has reduced its width to one.) These savings occur via clock and power gating to unused execution units.

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 [49]. Similarly, any values that are never used while on the bypass network need not be written to the bypass network, but only to the register file. The VLIW block formation process detects values with such unnecessary writes and marks them as such, saving energy during execution.

IV. Methodology

System Configuration: We evaluate our example HBA design against several baseline core designs, modeling each core and memory faithfully. Table I shows the main system parameters we model.

Simulator: We employ an in-house cycle-accurate simulator that is execution-driven. We faithfully model all major structures and algorithms within the HBA core and baseline cores, carry values through the model, and check against a functional model to ensure correctness. The model implements the user-mode x86-64 ISA by cracking instructions into μops (using a modified version of the PTLsim [64] decoder).

Power Model: To model core power/energy, we use a modified version of McPAT [36]. To model HBA's energy, we use McPAT's component models to construct a faithful model. We assume 2GHz operation for all core designs. We replaced McPAT's ALU model with a custom, more accurate, Verilog model we developed and synthesized for a commercial process with Synopsys tools. We provide all numbers and formulas of our model in a technical report [16].

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

Parameter	Setting
	Baseline Core:
Fetch Unit	ISL-TAGE [55]; 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; 4 ALUs, 1 MUL, 3 FPUs, 2 branch units, 2 load pipes, 1 store address and data pipe each
Memory Unit	96-entry load queue (LQ), 48-entry store queue (SQ)
L1/L2 Caches	64KB, 4-way, 5-cycle L1; 1MB, 16-way, 15-cycle L2; 64-byte blocks
DRAM	200-cycle latency; stream prefetcher, 16 streams
	Heterogeneous Block Architecture (HBA):
Block Size	16 μops, 16 liveins, 16 liveouts max
Fetch Unit	Baseline + 256-block info cache, 64 bytes/block
Global RF	256 entry; 16 rd/8 wr ports; 2-cyc. inter-backend comm.
Instruction Window	16-entry Block ROB
Backends	16 unified 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 sched.
Shared Execution 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 I: Major system parameters used in evaluation.

V_t (fast, leaky) or regular V_t (slow, less leaky) devices, the relative contribution of static and dynamic power may vary. For example, leakage can 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. 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 [12] 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 summarize key results for 15% leakage in §V-A.

Our power gating mechanism gates (i) scheduling logic in backends when they are in VLIW mode, (ii) superscalar ways when backends execute narrow VLIW blocks, and (iii) shared execution units (FPUs and the multiplier) in both HBA and in the baseline.

Workloads: We evaluate 184 distinct checkpoints, collected from the SPEC CPU2006, Olden [50], MediaBench [35] suites, and many real workloads: Firefox, FFmpeg, Adobe Flash player, V8 Javascript engine, GraphChi graph-analysis framework, MySQL, the lighttpd web server, LaTeX, Octave (a MATLAB replacement), and an x86 simulator. Many of these workloads have multiple checkpoints at multiple representative regions as provided by PinPoints [48]. All checkpoints are listed in [16], along with their individual performance and energy consumption on each of the evaluated core models.

Baselines: We compare HBA to four core designs. First, we compare to two variants of a high-performance out-of-order core: (i) one with a monolithic backend (scheduler, register file, and execution units), (ii) one with a *clustered microarchitecture* that splits these structures into separate clusters and copies values between them as necessary (e.g., [18]). The clusters have equivalent scheduler size and issue width as the block execution backends in the HBA core. Second, we compare to two variants of a coarse-grained heterogeneous design that combines an out-of-order and an in-order core [37] (iii) without clustering and (iv) with clustering. We model an ideal controller for this coarse-grained design, thus providing an upper bound on efficiency and performance relative to the real controller-based mechanism of [37].

V. Evaluation

Summary: 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 (§V-C summarizes this result by evaluating a variety of core designs that fall into different power-performance tradeoff points).

We analyzed HBA and other baselines extensively but can report only some analyses below due to space constraints. Our technical report [16] provides additional results, including sensitivity studies, power model details, individual benchmark results and more analyses.

A. Power

The main benefit of HBA is that it saves significant core energy (i.e., average core power). Table II shows average core power and Energy Per Instruction (EPI) for six core designs: baseline out-of-order, clustered out-of-order [18], coarse-grained heterogeneous [37], coarse-grained heterogeneous combined with clustered out-of-order, HBA with only out-of-order backends, and HBA with heterogeneous backends. HBA (row 6) reduces 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 in both core power and EPI.

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

TABLE II: Power, EPI, and performance vs. baseline out-of-order execution core.

To provide 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 (item *(i)*), saves 8.3% in EPI over the baseline monolithic core (first to second bar). 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 fifth bar). Making use of all heterogeneous execution backends (item *(iii)*) reduces energy by an additional 6.4% (fifth to sixth bar).

2. *Decoupled execution backends:* the clustered core saves instruction scheduling energy because each cluster has its own scheduling logic operating independently of the other clusters. Thus, the RS (scheduler) power reduces by 71% from the first to second bar in Fig. 5.

3. *Block atomicity:* Even without heterogeneous backends, HBA saves energy in renaming (RAT), program-order sequencing/retire (ROB), and global register file as it tracks blocks rather than instructions. Savings are because: (i) the block core renames only liveouts, rather than all written values, so RAT accesses reduce by 62%; (ii) the block core dispatches/retires whole blocks at a time and stores information about only liveouts in the ROB, reducing ROB accesses by 74%; and (iii) only 60% of register file accesses go to the global register file.

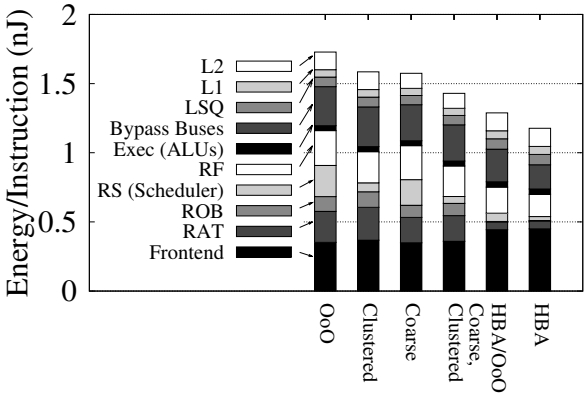


Fig. 5: Energy-per-Instruction (EPI) of six core designs.

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, (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. The state-of-the-art coarse-grained heterogeneous core [37] saves energy in both the out-of-order logic (RAT, ROB, and RS) and execution resources (bypass buses and register file) as it can use the in-order backend a portion of the time. However, when using the out-of-order backend, it cannot improve energy-efficiency. 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 (*Coarse, Clustered*) reduces EPI/power more than either the clustered core or coarse-grained core alone. However, this comes with higher performance degradation than any of the designs (2.8%, row 4 of Table II). HBA outperforms this coarse-grained, clustered design in IPC, power and EPI, as Table II and Fig. 5 show.

Overall, these results show that 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 [37], by leveraging block atomicity and heterogeneity synergistically.

Sensitivity: These savings are robust to power modeling assumptions. The above evaluation assumes leakage comprises 10% of total power (see §IV). If we assume 15% leakage power, worst case in our process, HBA still reduces average core power by 21.9% and EPI by 21.1% over the baseline out-of-order core.

B. Performance

1) Performance of HBA vs. Baselines: Table II 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) reduces average performance by only 1.0% over the out-of-order core. This is a result of (i) equivalent instruction window size, yielding similar MLP for memory-bound programs, and (ii) performance gain due to higher available issue width that balances performance loss due to inter-block communication latency, as explained in §V-B2.

2. When all blocks are executed on out-of-order backends only (row 5), the HBA design *improves* average performance by 0.4% over baseline, and 1.4% over nominal HBA (row 6). Thus, memoized scheduling has some performance penalty (as it sometimes sends a block to VLIW backends although dynamic scheduling would be better), but this penalty is modest (1.0%) for the energy reduction it obtains.

3. HBA provides similar performance to the coarse-grained heterogeneous core (row 3), but has much lower power/EPI. HBA saves more energy as it uses VLIW backends *for fine-grained blocks*, exposing more opportunity, and uses *memoized scheduling* to enable more blocks to use these backends. The coarse-grained design executes *a longer chunk of the program* on *only one core at a time*, and with strict in-order (as opposed to memoized) scheduling, so it must limit its use of the efficient in-order core to maintain performance.

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 [18] and coarse-grained heterogeneity [37]. HBA (rows 5,6) has higher performance and efficiency than this design.

2) Additional Analysis: To understand HBA's performance and potential, we perform several limit and control studies. Table III shows an out-of-order design as (i) its issue width is widened, and (ii) its fetch/retire width bottlenecks are removed. It also shows HBA (without heterogeneous backends) as inter-block communication latency is removed and the fetch width bottleneck is alleviated. We make several conclusions:

1. Inter-block communication latency penalizes performance. The "instant inter-block communication" HBA design (row 5) has 6.2% higher performance than the baseline (row 1).

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 III: Limit studies and control experiments.

2. *Higher aggregate issue rate* increases HBA's performance. This arises because each block execution backend has its own scheduler and ALUs that work independently, extracting higher ILP than in the baseline. 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 issue rate alone cannot account for all of the idealized HBA's performance. 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). Thus, other effects are present that make HBA better.

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

3) Per-Application Energy and Performance: Fig. 6 plots the IPC and EPI of HBA across all 184 benchmarks, normalized to the out-of-order baseline. In almost all workloads, HBA greatly reduces EPI. The left two-thirds of the benchmarks lose some performance mainly for the reasons described in §V-B2. The highest performance degradation is 42% for one benchmark (*ffmpeg*) that has a high block squash rate.[3] All but four benchmarks achieve at least 80% of the baseline performance. The rightmost one-third of benchmarks achieve higher performance on HBA. In some cases (e.g., *milc* from SPEC CPU2006), the performance gain is very large (79%), due to additional ILP exploited by independent HBA backends.[4]

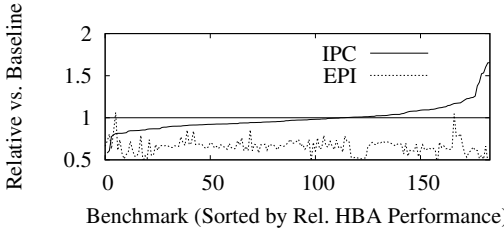


Fig. 6: HBA performance and EPI relative to baseline OoO.

C. Power-Performance Tradeoff Space

Fig. 7 shows multiple HBA and baseline core configurations in the 2D power-performance tradeoff space. HBA variants 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 is the most energy-efficient design (closest to the bottom-right corner), (ii) HBA's power-performance tradeoff is widely configurable, and (iii) HBA enables new points in the power-performance tradeoff space, not achievable by past designs.

[3]Two types of code perform 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.

[4]Workloads that perform best on HBA are largely those with regular code that can execute independent chunks in each backend.

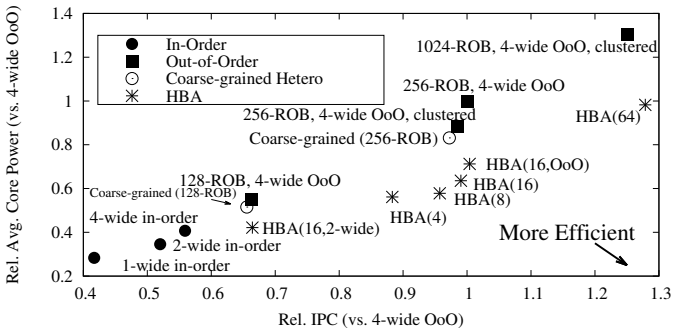


Fig. 7: Power-performance tradeoff space of core designs.

D. Symbiotic Out-of-Order and VLIW Execution

Fig. 8 shows a sorted curve of per-benchmark fraction of out-of-order vs. VLIW blocks. Most benchmarks execute both types of blocks often, with few having either all out-of-order or all VLIW blocks. In a few benchmarks on the left, almost all blocks (greater than 90%) execute as VLIW: for such benchmarks, learning one instruction schedule per block is sufficient to capture most of the benefit of out-of-order execution. Of the VLIW blocks, 32.7% are 2-wide, 3.3% are 1-wide. Thus, dynamic pipeline narrowing yields significant energy savings.

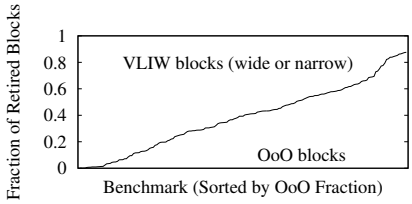


Fig. 8: Per-benchmark block type breakdown.

E. Core Area

Like past heterogeneous designs (e.g., [24, 33, 62]), HBA optimizes for a future in which energy and power (and not core area) are key performance limiters [14]. Similar to such designs, HBA's power reduction comes at the cost of chip area. We use McPAT, which provides a rough area estimate by estimating growth in key structures in the core. McPAT reports that our our initial HBA implementation increases core area by 62.5% over the baseline out-of-order core. For comparison, McPAT estimates 20% area overhead for [37]. Note that we did not aim to optimize for area in our HBA implementation in order to freely explore the power/performance tradeoff space.

Although HBA comes with area overhead, cores actually occupy a relatively small area in a modern mobile SoC (e.g., 17% in Apple's A7 [11]). Hence, the investment of additional core area to improve system energy efficiency can be a good tradeoff. By trading off die area for specialization, HBA: (i) achieves large energy savings that were not attainable in a smaller core (§V-A), and (ii) enables new power-performance tradeoff points not previously achievable (§V-C).

VI. Related Work

HBA is the first heterogeneous core design that enables the concurrent execution of very fine-grained code blocks (tens of instructions long) on the most efficient backend for each block. It combines several concepts, including heterogeneity, atomic block-based execution, and instruction schedule memoization, in a holistic manner to provide a new heterogeneous core design framework for energy-efficient execution. Though these concepts have been applied individually or in some combination (as we discuss briefly below), no past work applied them in the manner HBA does *to dynamically find the best execution backend for each fine-grained code block*. Other novel contributions were discussed at the end of §I.

Forming and Reusing Instruction Schedules: Several works use one execution engine to schedule/format instructions within a code block, cache the scheduled/formatted code block, and reuse that schedule on a simpler execution engine when the same code block is encountered later. An early example is DIF (Dynamic Instruction Formation) [43], which uses a simple in-order engine and a hardware-based instruction scheduler to schedule instructions in a code block the first time it is encountered. Later instances of the same code block are *always* executed on a *primary* VLIW engine. DIF thus uses an in-order engine to format the code to be executed on a VLIW engine. Similarly, Transmeta processors [32] use Code Morphing software to translate code for execution on a VLIW engine. Banerjia et al. [5] propose a similar high-level approach that pre-schedules instructions and places them in a "schedule cache" for later execution. A later example, ReLaSch [44], moves the out-of-order instruction scheduler to the commit stage. The first time a code block is encountered, its schedule is formed by this commit-stage scheduler. Later instances of the same code block are *always* executed in the *primary* in-order scheduler. ReLaSch thus uses an out-of-order instruction scheduler to format the code to be executed on an in-order scheduler. Several other works combine a "cold pipeline" and "hot pipeline" that execute infrequent (cold) and frequent (hot) code traces respectively [7, 24, 51], and use various mechanisms to form the traces to be executed by the "hot pipeline". *None of the above designs dynamically switch the backend the code block executes on*: once a code block is formatted/scheduled, it is *always* executed on the *primary* backend/engine, regardless of whether or not the schedule is effective. In contrast, HBA dynamically determines which backend is likely the best for any given instance of a code block: e.g., a code block may execute on the VLIW backend in one instance and in the OoO backend in the next. In other words, there is no "primary" or "hot" backend in HBA, but rather, the backends are truly equal and the best one is chosen depending on code characteristics.

Yoga [63], developed concurrently with HBA, can switch its backend between out-of-order and VLIW modes. HBA can exploit heterogeneity, and therefore adapt to characteristics of code blocks, at a finer granularity than Yoga as it can *concurrently execute blocks in different (VLIW and OoO) backends* whereas Yoga uses only one type of backend at a time. In addition, HBA's heuristic for switching a block from VLIW to OoO mode takes into account the "goodness" of the VLIW schedule (hence, the potential performance loss of staying in VLIW mode) whereas Yoga switches to OoO mode when an optimized VLIW frame does not exist for the code block.

Coarse-grained Heterogeneous Cores: Several works propose the use of statically heterogeneous cores [2, 3, 6, 10, 20, 23, 28, 33, 58, 59], dynamically heterogeneous cores [26, 30, 31, 60], one core with heterogeneous backends [37], or a core with variable parameters [4, 23] to adapt to the running application at a coarse granularity for better efficiency. We quantitatively compared to a coarse-grained heterogeneous approach [37] in §V 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 heterogeneity might lead to further gains, which is a promising path for future work to explore.

Atomic Block-Based Execution: Many past works (e.g., [8, 9, 25, 38, 39, 42, 47, 53, 57]) exploited the notion of large atomic code blocks to improve performance, efficiency and design simplicity. HBA borrows the notion of block atomicity and uses it as a mechanism to exploit fine-grained heterogeneity.

Other Related Works: [5, 40] propose pre-scheduling instructions to save complexity or improve parallelism. HBA takes advantage of the benefits of prescheduling by reusing instruction schedules. We adapt the ideas of dynamic pipeline narrowing and "software-based dead write elision" respectively from [27] and [49] to HBA. Note that none of these past works exploit heterogeneity as HBA does.

VII. Conclusion

This paper introduces the *Heterogeneous Block Architecture (HBA)* to improve energy efficiency of modern cores while maintaining performance. HBA combines fine-grained heterogeneity, atomic code blocks, and block-based instruction scheduling to adapt each fine-grained (tens of instructions long) code block to the execution backend that best fits its characteristics. Our extensive evaluations of an initial HBA design that can dynamically schedule atomic code blocks to out-of-order and VLIW/in-order execution backends using simple heuristics demonstrate that this HBA design (i) greatly

improves energy efficiency compared to four state-of-the-art core designs and (ii) enables new power-performance tradeoff points in core design. We believe HBA provides a flexible execution substrate for exploiting fine-grained heterogeneity in core design, and hope that future work will investigate other, more aggressive, HBA designs with more specialized backends (e.g., SIMD, fine-grained reconfigurable, and coarse-grained reconfigurable logic), leading to new core designs that are even more energy-efficient and higher-performance.

ACKNOWLEDGMENTS

We especially thank the members of the SAFARI research group for helpful feedback while developing this work. We thank many of the current and previous reviewers of this paper as well as David Hansquine for their valuable feedback and suggestions. This work was supported by the Intel URO Swiss Army Processor Program grant, the Qualcomm Innovation Fellowship Program, gifts from Oracle, and NSF Awards CCF-1147397 and CCF-1212962. We also thank our industrial partners and the Intel Science and Technology Center for the support they provide. Chris Fallin was supported by an NSF Graduate Fellowship. Onur Mutlu is supported by an Intel Early Career Faculty Honor Program Award and an IBM Faculty Partnership Award.

REFERENCES

[1] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *AFIPS*, 1967.

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

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

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

[5] S. Banerjia, S. Sathaye, K. Menezes, and T. Conte, "MPS: Miss-path scheduling for multiple-issue processors," *IEEE TC*, 1998.

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

[7] B. Black and J. P. Shen, "Turboscalar: A high frequency high IPC microarchitecture," *WCED*, 2000.

[8] D. Burger, S. W. Keckler, K. S. KcKinley, M. Dahlin *et al.*, "Scaling to the end of silicon with EDGE architectures," *IEEE Computer*, 2004.

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

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

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

[12] D. Deleganes *et al.*, "Designing a 3 GHz, 130 nm, Intel Pentium 4 processor," in *IEEE Sym. on VLSI Circuits*, 2002.

[13] P. Denning, "Working sets past and present," *IEEE TSE*, 1980.

[14] H. Esmaeilzadeh, E. Blem, R. S. Amant *et al.*, "Dark silicon and the end of multicore scaling," *ISCA-38*, 2011.

[15] S. Eyerman and L. Eeckhout, "Modeling critical sections in Amdahl's law and its implications for multicore design," *ISCA*, 2010.

[16] C. Fallin, C. Wilkerson, and O. Mutlu, "The Heterogenous Block Architecture," SAFARI Technical Report No. 2014-001, Mar 2014.

[17] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," *ISCA*, 2007.

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

[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," *CF*, 2005.

[21] M. Goshima *et al.*, "A high-speed dynamic instruction scheduling scheme for superscalar processors," *MICRO-34*, 2001.

[22] M. K. Gowan *et al.*, "Power Considerations in the Design of the Alpha 21264 Microprocessor," *DAC*, 1998.

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

[24] S. Gupta *et al.*, "Bundled execution of recurring traces for energy-efficient general purpose processing," *MICRO-44*, 2011.

[25] E. Hao, P.-Y. Chang, M. Evers, and Y. N. Patt, "Increasing the instruction fetch rate via block-structured instruction set architectures," *MICRO*, 1996.

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

[27] B. V. Iyer, J. G. Beu, and T. M. Conte, "Length adapative processors: A solution for energy/performance dilemma in embedded systems," *INTERACT*, 2009.

[28] J. A. Joao, M. A. Suleman *et al.*, "Bottleneck identification and scheduling in multithreaded applications," *ASPLOS-XVII*, 2012.

[29] J. A. Joao, M. A. Suleman *et al.*, "Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs," *ISCA-40*, 2013.

[30] Khubaib *et al.*, "MorphCore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP," *MICRO-45*, 2012.

[31] C. Kim, S. Sethumadhavan, M. S. Govindan *et al.*, "Composable lightweight processors," *MICRO-40*, 2007.

[32] A. Klaiber, "The technology behind Crusoe processors," 2000.

[33] R. Kumar, K. I. Farkas, N. P. Jouppi *et al.*, "Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction," *MICRO-36*, 2003.

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

[35] C. Lee *et al.*, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," *MICRO-30*, 1997.

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

[37] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman *et al.*, "Composite cores: Pushing heterogeneity into a core," *MICRO-45*, 2012.

[38] S. W. Melvin *et al.*, "Hardware support for large atomic units in dynamically scheduled machines," in *MICRO*, 1988.

[39] S. W. Melvin and Y. Patt, "Enhancing instruction scheduling with a block-structured ISA," *IJPP*, 1995.

[40] P. Michaud and A. Seznec, "Data-flow prescheduling for large instruction windows in out-of-order processors," *HPCA-7*, 2001.

[41] T. N. Mudge, "Power: a first-class architectural design constraint," *IEEE Computer*, 2001.

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

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

[44] O. Palomar *et al.*, "Reusing cached schedules in an out-of-order processor with in-order issue logic," *ICCD*, 2009.

[45] S. J. Patel, "Trace cache design for wide issue superscalar processors," Ph.D. dissertation, University of Michigan, 1999.

[46] S. J. Patel *et al.*, "Increasing the size of atomic instruction blocks using control flow assertions," *MICRO-33*, 2000.

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

[48] H. Patil *et al.*, "Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation," *MICRO-37*, 2004.

[49] B. R. Rau, C. D. Glaeser, and R. L. Picard, "Efficient code generation for horizontal architectures: Compiler techniques and architectural support," *ISCA*, 1982.

[50] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren, "Supporting dynamic data structures on distributed memory machines," *ACM TOPLAS*, 1995.

[51] R. Rosner *et al.*, "Power awareness through selective dynamically optimized traces," *ISCA-31*, 2004.

[52] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: A low latency approach to high bandwidth instruction fetching," *MICRO-29*, 1996.

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

[54] P. Sassone, J. Rupley, E. Brekelbaum *et al.*, "Matrix scheduler reloaded," *ISCA-34*, 2007.

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

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

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

[58] M. A. Suleman *et al.*, "ACMP: Balancing hardware efficiency and programmer efficiency," HPS Tech. Report TR-HPS-2007-001, 2007.

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

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

[61] K. van Craeynest, A. Jaleel, L. Eeckhout *et al.*, "Scheduling heterogeneous multi-cores through Performance Impact Estimation (PIE)," *ISCA-39*, 2012.

[62] G. Venkatesh, J. Sampson, N. Goulding *et al.*, "Conservation cores: reducing the energy of mature computations," *ASPLOS-XV*, 2010.

[63] C. Villavieja, J. A. Joao *et al.*, "Yoga: A hybrid dynamic VLIW/OoO processor," HPS Tech. Report TR-HPS-2014-001.

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