CG-OoO: Energy-Efficient Coarse-Grain Out-of-Order Execution Near In-Order Energy with Near Out-of-Order Performance

MILAD MOHAMMADI, Stanford University

TOR M. AAMODT, University of British Columbia

WILLIAM J. DALLY, NVIDIA / Stanford University

We introduce the Coarse-Grain Out-of-Order (CG-OoO) general-purpose processor designed to achieve close to In-Order (InO) processor energy while maintaining Out-of-Order (OoO) performance. CG-OoO is an energy-performance-proportional architecture. Block-level code processing is at the heart of this architecture; CG-OoO speculates, fetches, schedules, and commits code at block-level granularity. It eliminates unnecessary accesses to energy-consuming tables and turns large tables into smaller, distributed tables that are cheaper to access. CG-OoO leverages compiler-level code optimizations to deliver efficient static code and exploits dynamic block-level and instruction-level parallelism. CG-OoO introduces Skipahead, a complexity effective, limited out-of-order instruction scheduling model. Through the energy efficiency techniques applied to the compiler and processor pipeline stages, CG-OoO closes 62% of the average energy gap between the InO and OoO baseline processors at the same area and nearly the same performance as the OoO. This makes CG-OoO 1.8 $\times$ more efficient than the OoO on the energy-delay product inverse metric. CG-OoO meets the OoO nominal performance while trading off the peak scheduling performance for superior energy efficiency.

CCS Concepts: • **Computer systems organization** $\rightarrow$ *Superscalar architectures*;

Additional Key Words and Phrases: Energy efficiency, CPU architecture, block-level execution

ACM Reference format:

Milad Mohammadi, Tor M. Aamodt, and William J. Dally. 2017. CG-OoO: Energy-Efficient Coarse-Grain Out-of-Order Execution Near In-Order Energy with Near Out-of-Order Performance. *ACM Trans. Archit. Code Optim.* 14, 4, Article 39 (December 2017), 26 pages.

<https://doi.org/10.1145/3151034>

1 INTRODUCTION

Significant achievements have been made in improving the energy and performance properties of the Out-of-Order (OoO) execution model in recent years (Czechowski et al. 2014). However, the relationship between OoO energy and performance remains superlinear. In this design paradigm, achieving an optimal energy-performance point is typically done through dynamic voltage and frequency scaling (DVFS) techniques (Watanabe et al. 2010; Azizi et al. 2010; Zyuban 2000). As the energy-saving benefits of DVFS diminish with technology scaling (Le Sueur and Heiser

Authors' addresses: M. Mohammadi, Stanford University, Computer Systems Laboratory, Gates Room 241, Stanford, CA 94305; email: milad@cs.stanford.edu; T. M. Aamodt, Department of Electrical and Computer Engineering, University of British Columbia, 2332 Main Mall, Vancouver, BC, V6T 1Z4, Canada; email: aamodt@ece.ubc.ca; W. J. Dally, Stanford University, Computer Systems Laboratory, Gates Room 301, Stanford, CA 94305; email: dally@stanford.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 ACM 1544-3566/2017/12-ART39 \$15.00

<https://doi.org/10.1145/3151034>

2010), lower-complexity architectures, with reduced wire lengths and small/distributed hardware structures, become more favorable to achieve energy efficiency at the desired performance point.

To improve processor performance, complexity, and scalability beyond that of the OoO model, several coarse-grained microarchitectures, such as block-structured instruction set architectures (ISAs), multiscalar processor (Sohi et al. 1995), trace processor (Rotenberg et al. 1997), and TRIPS/EDGE (Sankaralingam et al. 2003; Burger et al. 2004), have been proposed. In this work, we synthesize a new design point using the right mix of pre-existing coarse-grained techniques (elaborated in Section 2, Table 1) as well as new techniques (Table 1, columns 8 and 9) to achieve superior energy efficiency compared to the OoO processor while maintaining near the same performance level. To do so, Coarse-Grain Out-of-Order (CG-OoO) draws from compiler-managed block-based ISAs (Sprangle and Patt 1994; Tseng and Patt 2008), complexity-effective designs (Palacharla et al. 1997; Rotenberg et al. 1997), hierarchical and distributed architecture designs (Sankaralingam et al. 2003; Burger and Keckler 2005), energy-aware architectures (Watanabe et al. 2010; Zyuban 2000), in-order and out-of-order hybrid issue models (Sankaralingam et al. 2003), block-level fetch and predict (Zmily and Kozyrakis 2005; Reinman et al. 1999a), and block-level commit architectures (Tendler et al. 2002).

Studies show that the sources of OoO high complexity and superior performance compared to the In-Order (InO) processor are effective control speculation (88% performance gain) and dynamic scheduling (10% gain) (McFarlin et al. 2013). We find that the majority of the energy gap between the two processor models is also due to these two mechanisms. In this article, we present a novel architecture that reduces the energy consumption of speculation and dynamic scheduling by about 36% compared to the OoO architecture while maintaining nearly the same nominal performance and area.¹

Speculation and dynamic scheduling are most effective during unpredictable dynamic events such as unpredictable cache misses (McFarlin et al. 2013; Sleiman and Wenisch 2016). To obtain high performance in the face of such events, current OoO implementations perform speculation and dynamic scheduling on all operations. In this article, we limit the energy impact of these mechanisms by reducing redundant accesses to key hardware structures and by rearchitecting large and expensive-to-access structures.

Our study is built off of four high-level observations. First, conventional OoO implementations incur energy consumption across all pipeline stages. Thus, an energy-efficient architecture should reduce energy of each stage. Second, conventional OoO execution relies on tight functional dependencies between stages requiring a solution to enable energy efficiency across all stages. Third, complexity-effective microarchitectures enable energy-saving opportunities (e.g., ILDP (Kim and Smith 2002), Complexity-Effective (Palacharla et al. 1997)). A block-level execution model, like CG-OoO, enables energy efficiency by simplifying complex, energy-consuming modules throughout the pipeline stages. Fourth, since dynamic scheduling and speculation techniques mainly benefit unpredictable dynamic events, they should be applied to instructions selectively.

This article describes an execution model, which we call CG-OoO, centered around clustering instructions. CG-OoO consists of three main components:

First, CG-OoO employs a *block-level control speculation* model that saves speculation energy throughout the pipeline stages. “Block level” refers to treating a block of instructions as a unit of computation flowing through the pipeline. CG-OoO supports block-level control speculation, fetch, dispatch, and commit and allows register renaming bypass. Several front-end² architectures

¹This work is published in Milad’s Ph.D. thesis and is available at the Stanford Online Library, and an initial version of this manuscript is available on ArXiv.

²CG-OoO is an end-to-end energy-efficient core architecture as opposed to a front-end architecture.

have shown that block-level speculation can be done with high accuracy and low energy cost (Zmily and Kozyrakis 2006; Reinman et al. 1999b; Hao et al. 1998).

Second, CG-OoO utilizes a *hierarchy of scheduling techniques* centered around clustering instructions. Static instruction scheduling organizes instructions at basic-block-level granularity to reduce stalls. The CG-OoO dynamic block scheduler dispatches multiple code blocks concurrently where each block allows limited out-of-order instruction issue using a complexity-effective structure named *Skipahead*. Skipahead accomplishes this by performing dynamic dependency checking between a very small collection of instructions at the head of each code block. Section 4.4.1 discusses the Skipahead microarchitecture. Skipahead is designed to meet the OoO average scheduling performance and trade off OoO peak scheduling performance with energy efficiency (see Section 6.2.3).

Third, CG-OoO uses a distributed *register file hierarchy* to allow static allocation of short-lived registers within a block and dynamic allocation of long-lived registers. We will discuss how this model enables energy-efficient register file accesses and reduces register renaming events.

This article makes the following contributions:

- It proposes Skipahead, an energy-efficient, limited out-of-order instruction scheduling model.
- It evaluates the integration of block-level predict, fetch, dispatch, commit, and dynamic scheduling for improving energy efficiency.
- It evaluates the use of a hierarchal register file model to reduce energy consumption and to reduce the number of register renaming events observing that not all operations require renaming.

The rest of this article is organized as follows. Section 2 presents the related work, Section 3 describes the CG-OoO execution model, Section 4 discusses the processor architecture, Section 5 presents the evaluation methodology, Section 6 provides the evaluation results, and Section 7 concludes the article.

2 OVERVIEW AND RELATED WORK

CG-OoO aims to design an energy-efficient, single-threaded processor core³ through targeting a design point where the complexity is nearly as simple as an In-Order and instruction-level parallelism (ILP) is paramount. In doing so, CG-OoO is partly inspired by some of the concepts and techniques in the literature.

Table 1 compares several high-level design features that distinguish the CG-OoO processor from the previous literature. Unlike others, CG-OoO's objective is energy-efficient computing (column 1), thereby designing several complexity-effective (column 3), energy-aware techniques, including an efficient register file hierarchy (column 10), a block-level control speculation, and a static and dynamic block-level instruction scheduler (columns 6, 7, 9) coupled with an issue model named *Skipahead* (column 8). Skipahead allows out-of-order instruction scheduling on two to five instruction slots at the head of an instruction queue rather than the entire queue. CG-OoO is a distributed instruction-queue model (column 1) that clusters execution units with instruction queues and register files to achieve an energy-performance-proportional solution (column 5).⁴

Braid (Tseng and Patt 2008) clusters static instructions at sub-basic-block granularity. The braid compiler employs program profiling to construct sub-basic-block dataflow chains. This execution

³Studying the energy efficiency of the memory hierarchy remains outside of the scope of this work.

⁴Not to be confused with energy-proportional designs (Barroso and Hölzle 2007). Energy-performance-proportional scaling refers to linear change in energy as the processor configuration allows higher performance (see Figure 26).

Table 1. CG-OoO Architecture High-Level Design Features Compared to the Previous Literature

DESIGN	1. Distributed μ -Architecture / Coarse-Grain	2. Energy Modeling	3. Complexity-Effective Design	4. Profiling NOT Done	5. Pipeline Clustering	6. Static & Dynamic Scheduling Hybrid	7. Block-Level Out-of-Order Scheduling	8. Skipahead Scheduling	9. Basic-Block-Level Predict, Fetch, Dispatch, Commit	10. Register File Hierarchy
CG-OoO	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Braid (Tseng and Patt 2008)	✓		✓			✓	✓			✓
WiDGET (Watanabe et al. 2010)	✓	✓	✓	✓	✓					
TRIPS (Sankaralingam et al. 2003; Burger and Keckler 2005)	✓		✓		✓	✓	✓			✓
Multiscalar (Sohi et al. 1995)	✓			✓		✓	✓			
CE (Palacharla et al. 1997)	✓		✓	✓	✓					
TP (Rotenberg et al. 1997)	✓		✓				✓			✓
MorphCore (Khubaib et al. 2012)		✓		✓			✓			
BOLT (Hilton and Roth 2010)		✓		✓			✓			
iCFP (Hilton et al. 2009)			✓	✓						
ILDp (Kim and Smith 2002)	✓		✓			✓				✓
POWER4 (Tendler et al. 2002)				✓			✓			
WaveScalar (Swanson et al. 2003)	✓		✓	✓	✓		✓			
Outrider (Crago and Patel 2011)				✓		✓	✓			
HSW (Brekelbaum et al. 2002)			✓	✓	✓					

model performs *instruction-level*, branch prediction, issue, and commit. Each braid execution unit runs in order. WiDGET (Watanabe et al. 2010) is an energy-proportional grid execution design consisting of a decoupled thread context management and a large set of simple execution units. WiDGET performs dynamic instruction-level data dependency detection to schedule instructions. In contrast to these proposals, CG-OoO is centered around clustering basic-block operations statically such that control speculation, fetch, dispatch, commit, and squash are done at block granularity (column 9). CG-OoO leverages energy-efficient, concurrent out-of-order issue from each code block (columns 7 and 8).

Multiscalar (Sohi et al. 1995) evaluates a multiprocessing unit capable of steering coarse-grained code segments, often larger than a basic block, to its processing units. It forwards the register values produced by a computation unit to the next one, causing data communication across its

register files. TRIPS and EDGE (Sankaralingam et al. 2003; Burger et al. 2004) are high-performance, dataflow, wide-issue, grid-processing architectures with many Execution Nodes (ENs) that use static instruction scheduling in space and dynamic (out-of-order) scheduling in time. An EN instruction buffer allows out-of-order scheduling across all its operations. ENs do not hold dedicated register files; instead, they share several register banks. TRIPS relies on large Hyperblocks (Mahlke et al. 1992) to map instructions to a grid of ENs. Hyperblocks use static branch predication to group basic blocks that are connected together through weakly biased branches. Static predication is not adaptive to program runtime behavior and leads to useless work. The TRIPS compiler uses program profiling to construct Hyperblocks. Palacharla et al. (1997) support a distributed instruction window model that simplifies the wake-up logic, issue window, and forwarding logic. In this article, instruction scheduling and steering is done at instruction granularity. Trace Processors (Rotenberg et al. 1997) is an instruction flow design based on dynamic code trace processing. The register file hierarchy in this work consists of several local register files and a global register file. ILDP (Kim and Smith 2002) is a distributed processing architecture that consists of a hierarchical register file built for communicating short-lived registers locally and long-lived registers globally. ILDP uses profiling and In-Order scheduling from each processing unit. In contrast to these proposals, the CG-OoO compiler employs neither program profiling (column 4) nor control predication to combine basic blocks; it relies on clustering operations at the basic-block level (column 9). CG-OoO uses local and segmented global register files to reduce data movement and storage energy (column 10). CG-OoO employs a limited out-of-order issue model (column 8).

iCFP (Hilton et al. 2009) addresses the head-of-queue⁵ blocking problem in the InO processor by building an execution model that, on every cache miss, checkpoints the program context and steers miss-dependent instructions to a side buffer enabling miss-independent instructions to make forward progress. CFP (Srinivasan et al. 2004) addresses the same problem in an OoO processor. Similarly, BOLT (Hilton and Roth 2010), Flea Flicker (Barnes et al. 2005), and Runahead Execution (Mutlu et al. 2003) are high-ILP, high-MLP,⁶ latency-tolerant architectures designed for energy-efficient out-of-order execution. All of these architectures follow the runahead execution model. BOLT uses a slice buffer design that utilizes minimal hardware resources. CG-OoO solves the head-of-queue scheduling problem through a hierarchy of energy-efficient solutions including the Skipahead (Section 4.4.1) scheduler (column 7).

POWER4 (Tendler et al. 2002) dynamically clusters instructions into groups of up to five Internal Instructions. Each group is dispatched in order and issued out of order. Each instruction group is committed as a whole.

WaveScalar (Swanson et al. 2003) and SEED (Nowatzki et al. 2015) are out-of-order dataflow architectures. The former focuses on solving the problem of long wire delays by bringing computation close to data. The latter is a complexity-effective design that groups data-dependent instructions dynamically and manages control flow using *switch instructions*. MorphCore (Khubaib et al. 2012) is an InO, OoO hybrid architecture designed to enable single-threaded energy efficiency. It utilizes either core depending on the program state and resource requirements. It uses dynamic instruction scheduling to execute and commit instructions. In contrast to the aforementioned, CG-OoO is a single-threaded, block-level, energy-efficient design that addresses the long wire delay problem through clustering execution units, register files, and instruction queues close to one another. CG-OoO is end-to-end coarse-grain, and code blocks do not need additional instructions to manage control flow.

⁵Stall of a ready operation behind another stalling operation in a first-in-first-out (FIFO) queue.

⁶MLP: Memory-level parallelism.

Fig. 1. The CG-OoO {BWs, EUs} cluster network.

Outrider (Crago and Patel 2011) and Load Slice Core (Carlson et al. 2015) address the head-of-queue stall problem by separating the instruction flow for memory and compute operations such that head-of-queue latency is minimized. Outrider requires compiler support to construct instruction strands to improve upon the performance of In-Order SMT cores. Load Slide Core focuses on memory hierarchy parallelism to substantially improve the performance of the In-Order processor model while retaining its energy efficiency.

HSW (Brekelbaum et al. 2002) proposes a complexity-effective, out-of-order design that extends the effective size of the OoO instruction window through a two-level instruction scheduling window; latency-critical operations flow into a small instruction window, while the rest flow into a second, larger instruction window. Instruction steering is done dynamically, and the instruction scheduler looks up both instruction windows to find ready operations.

Czechowski et al. (2014) discuss the energy efficiency techniques used in the recent generations of the Intel CPU architectures (e.g., Core i7, Haswell) including Micro-op cache, Loop cache, and Single Instruction Multiple Data (SIMD) ISA. This article questions the inherent energy efficiency attributes of the OoO model and presents a solution 42% more energy efficient than the baseline OoO. The energy efficiency techniques discussed in Czechowski et al. (2014) may be applied to the CG-OoO model to achieve additional gains.

3 CG-OOO ARCHITECTURE

The goal of the CG-OoO processor is to reach near the energy of the InO at the performance of OoO. This section introduces the CG-OoO as a block-level execution model that leverages a hierarchy of solutions (software and hardware) to save energy. Section 3.4 provides an execution flow example.

CG-OoO consists of multiple instruction queues, called Block Windows (BWs), each holding a dynamic block of code and issuing instructions concurrently. Each BW owns a dedicated register file and supports a *limited* out-of-order instruction scheduling model named *Skipahead*. BWs share execution units (EU) to issue instructions (Figure 1). Each EU consists of an integer, a floating point, and an address generation unit. Several BWs and EUs are grouped to form execution clusters. CG-OoO uses compiler support to group and statically schedule instructions.

3.1 Hierarchical Design

3.1.1 Hierarchical Architecture. CG-OoO groups instructions into code blocks fetched, dispatched, and committed together. In this study, a code block is defined to be a basic block (with single entry and single exit). At runtime, each dynamic block is processed from a dedicated BW. To manage architecture scalability as well as data communication energy and latency, BWs and EUs

are grouped together to form *clusters*. Figure 1 shows CG-OoO clusters highlighted; thin wires, in blue, enable data forwarding between EUs. Similar scalable architectures were previously studied by Salverda and Zilles (2008), Watanabe et al. (2010), and Chandrakasan et al. (1992). CG-OoO extends this concept to energy-efficient, block-level execution through evaluating a three-cluster architecture configuration.

3.1.2 Hierarchical Instruction Scheduling. We use static instruction list scheduling on each basic block to improve performance and energy

- (a) by optimizing the schedule of predictable instructions along the critical path,
- (b) by improving MLP via hoisting memory operations to the top of basic blocks, and
- (c) by minimizing wasted computation due to memory misspeculation (Section 3.2.1). Our list scheduler improves the -O3 gcc code by ordering instructions to meet the microarchitectural requirements of the CG-OoO model as opposed to the native x86 processor on which gcc was run. It assumes L1-cache latency for memory operations.

BWs in each cluster schedule instructions concurrently to hide each other's head-of-queue stalls. We call this scheduling model block-level parallelism (BLP). Furthermore, each BW supports a complexity-effective, limited out-of-order instruction issue model (Section 4.4.1) to address unpredictable cases where coarse-grained scheduling cannot provide enough ILP. These techniques combined help save energy by limiting the processor scheduling granularity to the program runtime needs (Section 3.4 shows an example).

3.1.3 Hierarchical Register Files. The CG-OoO register file hierarchy consists of a *Global Register File (GRF)* and *per-BW Local Register Files (LRFs)*. The GRF provides a set of architecturally visible registers that are dynamically managed, while LRF is statically managed, small, and energy efficient. The GRF is used for data communication across BWs, while LRF is used for data communication within each BW. Each BW has its dedicated LRF. As shown in Section 6.2.2, 30% of data communication (register→register and register↔memory) is done through LRFs. To further save energy, the GRF is segmented and distributed among BWs. GRF segmentation does not rely on a block-level execution model and may be used independently. Similar register file models are studied in Kim and Smith (2002), Tseng and Patt (2008), Rotenberg et al. (1997), Tseng and Asanović (2003), and Seznec et al. (2002b). Some ARM architectures bank the register file for purposes such as improving thread context switching efficiency (Tool 2015). CG-OoO evaluates them from an energy standpoint.

BWs can access any GRF segment. The CG-OoO register rename algorithm employs a renaming heuristic that minimizes the physical distance between a global register segment and the operation writing to it. Placing a GRF segment next to each BW saves energy when operations read/write global operands from/to the closest segment.

Communication among BW components (e.g., global register segments, wakeup logic) within the same cluster takes one cycle and is enabled through point-to-point links. Communication among BW components in different clusters takes two cycles and is enabled through a crossbar (Watanabe et al. 2010).

3.2 Block-Level Speculation

OoO processors avoid fetch stall cycles by performing Branch Prediction Unit (BPU) lookups immediately before *every fetch* irrespective of the fetched instruction types (Seznec et al. 2002a); this leads to excessive speculation energy cost due to redundant BPU lookup traffic by noncontrol instructions (Mohammadi et al. 2015). CG-OoO supports energy-efficient, block-level fetch and predict using a compiler-generated operation named head that

Fig. 2. The head instruction format.

Fig. 3. A simple do-while loop program and its assembly code.

- (a) specifies the start of a new code block,
- (b) accesses the BPU to predict the next code block, and
- (c) triggers BW allocation to steer the upcoming code block operations (Figure 4).

Immediately after Fetch, instructions are checked to identify head operations. The BPU is only accessed when a head is detected (as later shown in Figure 6(c)). Because block speculation is an early prediction mechanism, head is often ahead of its branch by at least a cycle. A pipeline bubble may occur if the current block fetch completes before the next code block address is predicted. In our evaluations, less than 1% delay is introduced due to bubbles on average.

Figure 2 shows the head instruction fields:

- (a) opcode,
- (b) control instruction presence bit,
- (c) block size,⁷ and
- (d) control instruction least significant address bits.

The example code in Figure 3 shows head has HasCtrl=1'b1, indicating a control operation ends the basic-block. If it has HasCtrl=1'b0, BPU lookup is disabled to save energy. In Figure 3, local and global operands are identified by r and g prefixes, respectively.

3.2.1 Squash Model. CG-OoO supports block-level speculative control and memory squash. Upon control misprediction, the front end stalls fetching new instructions, all code blocks younger than the mis-speculated control operation are flushed, and the remaining code blocks are retired. The data produced by wrong-path blocks are automatically discarded as such blocks never retire. Once the Block Re-Order Buffer (BROB) is clear of wrong-path blocks, the processor resumes normal execution.

3.3 CG-OoO Block-Level ISA

The compiler adopts a fixed-width ISA that closely models the MIPS ISA (Price 1995). The CG-OoO ISA supports *Local Register* as well as *Global Register* operands as discussed in Section 3.1.3, and introduces the head instruction to annotate code blocks as discussed in Section 3.2. The following hardware resources are accounted for by the compiler to construct code blocks: (1) BW instruction queue size, (2) number of available local registers per BW, and (3) number of global write register operand fields per BROB entry.

⁷The compiler partitions code blocks larger than 32 instructions. Bird et al. (2007) shows that the average size of basic blocks in the SPEC CPU 2006 integer and floating-point benchmarks are five and 17 operations, respectively.

Fig. 4. CG-OoO processor pipeline stages.

Fig. 5. (a) The CG-OoO architecture models. (b) Cycle-by-cycle flow of example instructions in CG-OoO.

The BW Instruction Queue size determines the allowable number of operations per code block. Accordingly, the compiler breaks large basic blocks to accommodate the hardware requirements. Section 5 lists the hardware resource sizes for BWs.

Because local registers are expressed via the ISA encoding, the ISA specifies the LRF size (i.e., the number of local register specifier bits in the encoding). As a result, the compiler terminates a block early once it runs out of local registers. The remaining operations form a new block following the original one. CG-OoO supports 20 local registers per block. The same discussion applies to the global write registers per block; these registers are limited by the number of available register identifier fields per BROB entry (i.e., 10 entries as shown in Section 4.6). Given the ISA dependence on the CG-OoO microarchitectural features, backward compatibility can be obtained through dynamic binary translation (Dally 2011; Dehnert et al. 2003; Apple 2006).

3.4 CG-OoO Program Execution Flow

This section discusses the CG-OoO execution model through a code example. Figure 4 shows the CG-OoO pipeline. The highlighted stages differ from the traditional OoO. Control speculation, dispatch, and commit are at block granularity, and rename is only used for global operands. Section 4 discusses how each stage saves energy.

Figure 5(a) illustrates a two-wide superscalar CG-OoO. The instruction scheduler issues one instruction per BW per cycle to the two EUs. The code in BWs is two consecutive iterations of the aforementioned do-while loop. Figure 5(b) shows the cycle-by-cycle flow of instructions through the CG-OoO pipeline. Instructions in iterations 1 and 2 are green and red, respectively. It also shows the contents of BW0, BW1, and the BROB. Here, lw has four-cycle latency, and all others have one-cycle latency.

In cycle 1, {head.1, add.1} instructions are fetched from the instruction cache. In cycle 2, the immediate field of head.1 is forwarded to the BPU. In cycle 3, head.1 speculates the next code block *before* the control operation, bne.1, is fetched; furthermore, the Block Allocator assigns BW0 to the instructions following head.1, and BROB reserves an entry for head.1 to store the runtime status of its instructions. In cycle 4, BW0 receives its first instruction. In cycle 5, add.1 is issued while more instructions join BW0. In cycle 10, the last instruction of iteration 1 leaves BW0. In cycle 11, BW0 is available to hold new code blocks. In cycle 13, head.1 is retired as all its instructions complete execution; at this point, all data generated by the block operations will be marked nonspeculative.

Fig. 6. CG-OoO front-end microarchitecture.

4 CG-OOO Microarchitecture

This sections presents the CG-OoO pipeline microarchitecture details and highlights their energy-saving attributes. These stages save energy by utilizing several complexity-effective techniques through

- (a) the use of small tables,
- (b) reduced number of table accesses, and
- (c) hardware-software hybrid instruction scheduling.

4.1 Branch Prediction

Figure 6(a) shows the microarchitectural details of the branch prediction stage in the CG-OoO processor; it consists of the Branch Predictor (BP) (Seznec et al. 2002a), Branch Target Buffer (BTB), Return Address Stack (RAS), and Next Block-PC. Equation (1) shows the Next Block-PC computation relationship:

$$PC_{Next-head} = PC_{head} + fall-through-block-offset. \quad (1)$$

The fall-through-block-offset is the immediate field of the head instruction shown in Figure 2. In the CG-OoO model, only head PCs access the BPU. To detect head operations prior to decode, the head Opcode Detector logic determines if the fetch group contains a head (Figure 6(c)). Upon lookup, a head PC is used to predict the next head PC. Speculated PCs are pushed into a FIFO queue, named *Block PC Buffer*, dedicated to communicate block addresses to Fetch (Figure 6(b)).

4.2 Fetch Stage

Figure 7(a) illustrates a control flow graph with five basic blocks. Each block is marked with its head identifier, *h*, at the top, and its control operation identifier (if any), *c*, at the bottom. Figure 7(b) illustrates the mapping of these basic blocks to the instruction cache, where each box represents

Fig. 7. (a) A control flow graph (CFG) with five blocks labeled B1 to B5. (b) Mapping of CFG operations onto a four-wide instruction cache.

an instruction and each set of adjacent boxes with the same color represent a fetch group.⁸ Entries marked I represent noncontrol, nonhead operations in each basic block in Figure 7(a). As shown in Figure 7(b), more than one head may exist in a fetch group. Fetch and BPU handle such cases by queuing head operations.

The Block PC Buffer holds either a next-PC address or a 64'b0, where the former is the next code block to fetch from the cache, and the latter is a hint that next-PC is unknown. An unknown PC happens when a head operation, hh, is predicted *not-taken* and hh itself is not yet available. Recall, the predictor needs to have the fall-through-block-offset of hh to predict the next block. In such cases, Fetch assumes the fall-through block is adjacent to the hh block in memory; so, it continues fetching the next block while the fall-through block address for hh is computed.

4.3 Decode and Register Rename Stages

The Decode microarchitecture follows that of the conventional OoO except for its additional functionality to identify global and local register operands by appending a 1-bit flag, named *Register Rename Flag (RRF)*, next to each register identifier. If an instruction holds a global operand, it accesses the register rename (RR) tables for its physical register identifier; otherwise, it would skip RR lookup (Figure 6(d)). Skipping the register rename stage reduces the renaming lookup energy by 14% on average. This saving is realized due to our block-level execution model. Our RR evaluations use the Merged Rename and Architectural Register File model discussed in Kessler et al. (1998), Gwennap (1997), and Hinton et al. (2001).

4.4 Issue Stage

Before discussing the CG-OoO issue model, let us visit the Block Window microarchitecture shown in Figure 8(a). It consists of an Instruction Queue (IQ); a Skipahead Buffer (SB), which is similar to a small bank of reservation stations;⁹ a dedicated LRF; a GRF segment; and a number of EUs. IQ is a FIFO that holds code block instructions. SB is a *small buffer* that holds instructions waiting to be issued by the Instruction Scheduler in a content accessible memory (CAM) array. SB pulls instructions from the IQ and waits for their operands to become ready for issue. The CG-OoO issue model allows register file accesses only to operations in the SB, thereby

⁸This section assumes no fetch alignment (Mahin et al. 1997).

⁹Because CG-OoO supports compiler-managed local registers, SB tracks true and false dependencies to mark instructions ready. A reservation station tracks true dependencies only.

Fig. 8.

- (a) avoiding the OoO postissue register file read cycle and
- (b) saving the preissue large data storage overhead (González et al. 2010) by only storing operands in SBs.

Because the number of operations in all SBs is a fraction of all in-flight instructions, this model is as fast as the OoO preissue model, and more energy efficient than both models. Section 6.2.3 compares the energy consumption of the CG-OoO issue model against that of the baseline OoO.

4.4.1 Skipahead Instruction Scheduler. The Skipahead model allows *limited out-of-order* issue where the term *limited* means out-of-order issue is restricted only to instructions in SB, a subset of BW operations. An instruction, *Ins*, in SB may be issued out of order when it has neither true nor false dependency on other SB instructions ahead of it. Figure 8(b) shows the simple XOR logic used to check dependencies in an SB. Assuming a three-entry SB, Figure 9 shows an example code sequence where instructions are issued as {1, 3} followed by {2, 4}. Before issuing 3, its operands are dependency checked against those of 2. In Section 6, we show that a limited OoO issue model like Skipahead can cover the performance gap between the traditional InO and OoO execution models at a fraction of the energy cost.

The Skipahead model improves the CG-OoO performance by 36% (Section 6.1) while significantly limiting the select and wakeup logic energy demands. Upon completion, instructions with global write operands send a wakeup signal to all SBs through a crossbar link; on the other hand, instructions with local write operands send a wakeup signal only to their own SB. When a wakeup message consumer is not yet in the Skipahead Buffer (i.e., it is in IQ), it ignores the broadcast message. Later on, when it joins the Skipahead Buffer, the instruction accesses the register file to read its already available operand. The wakeup unit presents three sources of energy efficiency:

- (a) Each BW utilizes a small SB storage space to hold operand *data*. In contrast, the OoO Instruction Queue maintains ready operand data for all operations.
- (b) The wakeup unit searches small CAM tables for *source operands*. For instance, in a CG-OoO processor with eight BWs, each with three SB entries, the wakeup unit accesses 24 CAM entries in total. The OoO baseline assumes 64 (Mutlu et al. 2003; Corporation 2009) in-flight operations in Instruction Window to search for ready operands.
- (c) Local write operands wake up source operands associated with their own BW *only*.

1.	sll	r1, r2, #3
2.	lw	r0, r1
3.	add	r3, r4, #1
4.	bne	r2, r3, LOOP

Fig. 9. A code snippet with two data dependencies.

Block SN	BlkSize	GW0	GW1	GW2	GW3	GW4	GW5	GW6	GW7	GW8	GW9
----------	---------	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Fig. 10. A Block Reorder Buffer (BROB) entry.

This design architecture is 35% more energy efficient than the case where wakeup broadcasts results to all in-flight instructions.

4.5 Memory Stage

CG-OoO is focused on presenting an energy-efficient processor core while using identical memory architecture models for all three processors. Similar to the OoO processor, CG-OoO uses a unified load-store-unit (LSU) architecture that operates at instruction granularity and handles up to one pipelined cache access per cycle. In this model, a squash is triggered when an sw conflicts with a younger lw, at which point the block holding the lw is flushed; this means *useful* instructions older than lw are also squashed. For instance, in Figure 9, if operation 2 were to trigger a memory mis-speculation event, the entire block, including operation 1, would be squashed. Flushing useful operations is called *wasted squash*, which the compiler reduces by hoisting memory operations toward the top of basic blocks. To prevent the mis-speculation from happening again during re-execution, the mis-speculated block is executed in order.

Efficient memory speculation models such as NoSQ (Sha et al. 2006) can further improve processor energy efficiency by replacing associative LSU lookups with indexed lookups, and by converting predictable store-load communications to register communications. Segmentation of LSU into smaller, cluster-level units further improves the CG-OoO energy efficiency and scalability. LSU segmentation does not rely on a block-level execution model and may be used independently (Park et al. 2003). Other memory architecture optimizations such as data prefetching bring similar benefits to both OoO and CG-OoO. The design of a scalable and energy-efficient memory system architecture is outside of the scope of this article.

4.6 Write-Back and Commit Stage

Figure 10 shows the contents of a BROB entry; it holds the block sequence number, block size, and block global write (GW) register operand identifiers. BlkSize is initialized by the corresponding head operation. The GW fields are updated by instructions with global write registers as they are steered from the Register Rename stage to their BW. The compiler controls the number of global write operands per code block.

4.6.1 Write-Back Stage. Once an instruction completes, it writes its results into either a designated register file entry (global or local) or the store queue. In Figure 10, BlkSize is decremented upon each instruction complete; once its value is zero, the corresponding *block is completed*. When a block completes, its corresponding BW is marked available for new block dispatch, and all its storage structures (LRF, SB, IQ) are labeled invalid.

4.6.2 Commit Stage. A block is committed when it is *completed* and is at the head of the BROB. During commit, the global registers modified by the block are marked *Architectural* using the GW fields in BROB. By design, no two global write registers in a code block correspond to the same static register operand.¹⁰ This allows the global write registers of a code block to concurrently

¹⁰The mutual exclusivity of global write registers within a code block is due to the presence of local registers.

retire to GRF. For the baseline ROB, however, the write registers of two concurrently committing instructions may conflict and require serialization. Code blocks are properly serialized to commit in order. Local registers are not tracked in the BROB as they are statically allocated, and when a block is incorrectly speculated, these registers are squashed with the block.

Upon commit, sw operations in the committing code block retire; in doing so, the Store-Queue “commit” pointer moves to the youngest sw belonging to the committing block. This sw is found via searching for the youngest store operation whose Block SN matches that of the committing block. Note, our LSU holds a Block SN column.

Exceptions are handled at block granularity. Upon an exception, instructions are executed in order. Once recovered from the exception, the processor resumes normal execution.

To handle block-level commit in multiprocessor CG-OoO, additional compiler support is required to guarantee memory consistency; requirements vary according to the chosen model. For instance, to support total-store-order (TSO), the compiler should break basic blocks at atomic operations. This enables correct handling of implicit fences associated with atomics. Also, the CG-OoO block-level commit model does not permit supporting Sequential Consistency (SC). Multiprocessor CG-OoO is outside of the scope of this contribution.

Checkpoint-based processors (Akkary et al. 2003; Cristal et al. 2002, 2004) propose a general concept applicable to many architectures (e.g., iCFP (Hilton et al. 2009)). While outside of the scope of this work, coarse-grain checkpoint-based processing is promising for extending the energy efficiency of CG-OoO.

4.7 Squash Handling

CG-OoO handles squash events through the following steps:

- (a) The BPU history queue and Block PC Buffer flush the content corresponding to wrong-path blocks. The code block PC resets to the start of the right path; in case of a control mis-speculation, the right path is the opposite side of the control operation, and in case of a memory mis-speculation, it is the *start* of the same code block.
- (b) All BWs holding code blocks younger than the mis-speculated operation flush their IQ and Skipahead Buffer and mark LRF registers invalid.
- (c) LSU flushes operations corresponding to the code block younger than the mis-speculated operation by comparing the mis-speculated Block SN against that of memory operations.
- (d) BROB flushes code block entries younger than the mis-speculated operation. The remaining blocks complete execution and commit.

5 METHODOLOGY

The evaluation setup consists of an in-house compiler, simulator, and energy model. The compiler performs *Local Register Allocation* and *Global Register Allocation* as well as *Static Block-Level List Scheduling* for each program basic block. The simulator consists of a Pin-based functional emulator attached to a timing simulator (Luk et al. 2005). The emulator produces an identical set of dynamic operations irrespective of the simulated architecture. The emulator supports wrong-path execution. The dynamic micro-operations produced by the emulator are converted into the internal ISA before processing by the timing simulator. Table 2 outlines the configurations used by the simulator to support the timing and energy model for the CG-OoO, OoO, and InO processors. The simulator uses the cache and memory model in Das et al. (2015). All evaluations support instruction fetch alignment. They also support data forwarding between EUs. Evaluations use the PinPoints tool, based on SimPoints (Hamerly et al. 2005; Patil et al. 2004). Warmup is done on the

Table 2. System Parameters for Each Individual Core

Shared Parameters		
Technology Node, System Clock Frequency	22nm, 2GHz	
L1-I, assoc, latency	32KB, 8, 4 cycles	
L1, assoc, latency	32KB, 8, 4 cycles	
L2, assoc, latency	256KB, 8, 12 cycles	
L3, assoc, latency	4MB, 8, 40 cycles	
Main memory latency	100 cycles	
Instruction Fetch Width	1 to 8	
Branch Predictor	Hybrid - 20-cycle branch mispredict	
- G-Share, Bimodal, Meta, Global Pattern Hist	8Kb, 8Kb, 8Kb, 13b	
BTB, assoc, tag	4,096 entries, 8, 16b	
Load/Store Queues	64, 32 entries, CAM	
Architectural Register Count	70	
InO Processor		
Pipeline Depth	7 cycles	
Instruction Queue	8 entries, FIFO	RAM, 4R-4W Ports
Register File	70 entries (64bit)	RAM, 8R-4W Ports
Execution Unit	1-8 wide	
OoO Processor		
Pipeline Depth	13 cycles	
Instruction Queue	64 entries	RAM/CAM, 4R-4W Ports
Register File	250 entries (64 bits)	RAM, 8R-4W Ports
Execution Unit	1-8 wide	
Reorder Buffer (ROB)	180 entries	RAM, 4R-4W Ports
CG-OoO Processor		
Pipeline Depth	13 cycles	
Instruction Queue / BW	4 parallel FIFO queues, 4 entries each	RAM, 1R-1W Ports
Skipahead Buffer / BW	2-5 entries	RAM/CAM, Up to 4R-4W Ports
Execution Unit / BW	1-8 wide	
GRF, LRF / BW	250, 20 entries (64 bits)	RAM, 2R-2W Ports
Block Reorder Buffer (BROB)	16 entries	RAM, 4R-4W Ports
Number of BWs	3-18	
Number of Clusters	1-3	

6 million instructions prior to SimPoints. We chose the SPEC Int 2006 benchmark suites for their inherently limited instruction-level parallelism (Henning 2006).

5.1 Energy Model

Similar to the approach in McPAT (Li et al. 2009), our energy model produces per-access energy numbers for the simulator to use for computing the total energy of the hardware units in Table 2. Column 3 lists the design configuration for the main hardware structures modeled. This model supports tables, caches, wires, stage registers, and execution unit energies and areas by extending the energy model in Das et al. (2015). It estimates per-access dynamic energy and per-cycle static energy consumption. Other logical blocks in the processor (e.g., control modules) are assumed to have similar energy costs for the baseline OoO and the CG-OoO, and therefore have a secondary effect on the overall energy difference.

RAM tables are modeled as standard (multiported) SRAM units accessed through decoder and read through sense amplifiers. Static and dynamic energy are generated using SPICE. All steps including area estimation, energy scaling for different port configurations, and cache structures are done in SPICE. Similarly, CAM tables are designed as standard SRAM units accessed through a driver input module and read through sense amplifiers. To evaluate the energy and area of pipeline stage registers, 6-NAND gate-positive edge-triggered flip-flops (FFs) are simulated in SPICE.

Different 64-bit execution units including the add, multiply, and divide units for arithmetic and floating-point operations are developed in Verilog and simulated in the Design Compiler

Fig. 11. Performance of InO and CG-OoO normalized to OoO.

(Azizi et al. 2010). The Design Compiler provides per-operation energy numbers for each unit. HotSpot (Huang et al. 2006) is used for optimizing floorplan area and wire lengths. To extract wire energy numbers, we upgraded HotSpot to report wire energy using its wire length model. The energy per access used for wires is 80fJ/b-mm at the 22nm technology node (Cao et al. 2002). The simulator assumes all wires have a 0.5 activity factor;¹¹ so, when the simulator drives a wire, its energy consumption is incremented by half of its per-access energy.

6 EVALUATION

The section evaluates the performance and energy characteristics of a three-cluster CG-OoO model with four EUs, four BWs per cluster, and five-entry Skipahead Buffers. This configuration consumes 64% of the OoO energy at near the same performance on SPEC Int 2006 benchmarks. This implies CG-OoO closes 62% of the energy gap between the InO and OoO processors. This section quantifies the energy and performance attributes of CG-OoO and the energy-saving contribution of each pipeline stage.

6.1 CG-OoO Performance Evaluation

Figure 11(a) uses a 4-wide OoO superscalar processor as the baseline for illustrating the relative performance of a 4-wide InO and CG-OoO (4-wide front-end, Skipahead-5, four EUs, four BWs, arranged as one cluster) processor. In this case, the CG-OoO harmonic mean performance is 21% lower than the OoO baseline. Performance results are measured in terms of instructions per cycle (IPC). In Figure 11(b), the same 4-wide InO and OoO configurations are compared against a CG-OoO model with a 4-wide front-end, 12 EUs, 12 BWs, Skipahead-5, spread across three clusters. In this configuration, the CG-OoO average ILP reaches within 2% of the OoO. Similar results are obtained when comparing this CG-OoO configuration against a 4-wide OoO with 12 EUs. Figure 11(b) indicates that the higher availability of resources allows exploiting higher ILP for some benchmarks including Hmmer, Bzip2, and Libquantum.

The first source of performance gain is *static block-level list scheduling*. Figure 12 shows the effect of static scheduling on performance. On average, static scheduling increases the CG-OoO performance by 13%. Because the entire instruction queue is visible to the OoO scheduler, the list

¹¹Here, activity factor refers to a fraction of a toggle.

Fig. 12. Effect of static block-level list scheduling on CG-OoO performance. The Skipahead 5 dynamic scheduling model is used for both results (see Figure 13).

Fig. 13. The performance attribute of the Skipahead model.

scheduler does not offer performance improvements to OoO. On the other hand, CG-OoO utilizes a limited out-of-order scheduling model that gains performance improvements with a better static code schedule; this optimization provides scheduling visibility to high-latency operations early.

The next source of performance gain is through BLP. To illustrate the contribution of BLP, let us assume each BW can issue up to four operations in order; that is, if an instruction at the head of a BW queue is not ready to issue, younger, independent operations in the same queue do not issue. Other BWs, however, can issue ready operations to hide the latency of the stalling BW. We find that six BWs are in flight on average. The *No Skipahead* bar in Figure 13 refers to this setup. It shows that, on average, 18% of the performance gap between the InO and OoO is closed through BLP. Benchmarks like Perl and Sjeng exhibit better performance for the InO model. This is because the InO processor has a shallower pipeline depth (seven cycles) compared to the CG-OoO processor (13 cycles), allowing faster control mis-speculation recovery.

The last source of performance improvement in CG-OoO is the Skipahead model where *limited* out-of-order instruction scheduling within each BW is provided. McFarlin et al. (2013) note that the superior performance of OoO is mainly attributed to its ability to effectively find useful work to hide head-of-queue stalls. CG-OoO reaches this goal by illustrating that stall-independent operations are found within a small distance from the stalling operation. Skipahead enables limited out-of-order scheduling to find stall-independent operations by bypassing head-of-queue stalls. This feature leverages the Skipahead Buffer tables. Figure 13 shows the performance gain obtained via varying the number of SB entries. Without Skipahead, 18% of the gap between OoO and InO is closed. *Skipahead 2* refers to a two-entry SB, which closes 59% of the performance gap. *Skipahead-5* (i.e., five-entry SB) closes 98% of the performance gap, and *Skipahead-8* closes the entire performance gap. These results indicate a significant performance gain as the Skipahead buffer size grows from 2 to 5, and a marginal performance gain beyond. All CG-OoO results use the statically list scheduled code.

Figure 14 shows the CG-OoO performance as the processor front-end width varies from 1 to 8. Comparing the harmonic mean results for the OoO and CG-OoO shows that the CG-OoO processor is superior on narrower designs. A wider front end delivers more dynamic operations to the back end. Because the OoO model has access to all in-flight operations, it can exploit a larger effective instruction window. Despite the larger number of in-flight operations, CG-OoO

Fig. 14. The speedup of the OoO and CG-OoO for different front-end widths.

maintains a limited view to the in-flight operations, making an 8-wide CG-OoO not much superior to its 4-wide counterpart.

WiDGET reports 0.7 average speedup, on SPEC Int 2006, for an 8EU architecture with one instruction buffer per EU (Watanabe et al. 2010).¹² The average performance of CG-OoO with eight EUs and one Block Window per EU is 0.79. WiDGET performs instruction clustering at the Steer stage (without list scheduling) and adopts a unified register file. CG-OoO performs instruction clustering and list scheduling at compile time and adopts LRFs and segmented-GRFs to optimize data communication latency.

6.2 CG-OoO Energy Evaluation

In this section, the source of energy saving within each pipeline stage is discussed. Overall, CG-OoO closes 62% of the energy gap between the InO and OoO processors at the same area as the OoO baseline. Energy results are measured in terms of energy per instruction (EPI). Figure 15(a) shows the total energy level for the CG-OoO, OoO, and InO processors; Figure 15(b) shows the harmonic mean energy breakdown for different pipeline stages. All benchmarks follow a similar energy breakdown trend as the harmonic mean. This figure highlights the overall energy savings in *Branch Prediction*, *Register Rename*, *Issue*, *Register File access*, and *Commit* stages. These gains are further quantified throughout the rest of this section. The *Fetch* stage energy is 9.5% higher than OoO, primarily due to the addition of head operations to the binary. The *Execute* stage energy is 2.5% higher than OoO, mainly due to the static energy of the additional eight EUs in CG-OoO. Figure 15(c) shows the energy-saving benefits of the CG-OoO core excluding the cache and memory. It shows 43% average energy savings for the CG-OoO core compared to OoO baseline with similar performance.

Figure 16 shows the inverse of the energy-delay (ED) product indicating the favorable energy-delay characteristics of the CG-OoO over OoO for all benchmarks, even those that fall short of the OoO performance such as Libquantum and Mcf. The CG-OoO is 1.8× more efficient than the OoO on average. Figure 17 shows the static and dynamic energy breakdown for different benchmarks relative to the OoO baseline. On average, static energy is about 6% and 10% of the total energy for OoO and CG-OoO, respectively. The extra static energy for CG-OoO is mainly due to the availability of 12 BW structures and wider execute pipeline registers.

6.2.1 Block-Level Branch Prediction. Block-level branch prediction is primarily focused on saving energy by accessing the branch prediction unit at block granularity rather than fetch-group

¹²Caveat: the discussed result is drawn directly from Figure 12 of WiDGET without modification. This figure provides an intermediate evaluation to make an argument for using multiple instruction buffers per EU.

(a) Normalized EPI of processors (cache energy included).

(b) Harmonic mean EPI breakdown of all processors.

(c) Normalized EPI of processors (cache energy excluded).

Fig. 15. CG-OoO, OoO, and InO energy per instruction (EPI).

Fig. 16. The CG-OoO inverse of energy-delay product normalized to OoO.

granularity. Figure 18 shows the average block sizes for SPEC Int 2006 benchmarks. For a benchmark application with average block size of eight running on a 4-wide processor, this translates to a roughly $2\times$ reduction in the number of accesses to the BPU tables. Figure 19 shows the relative energy per instruction for the CG-OoO model compared to the OoO baseline. On average, Block-Level BP is 34% more energy efficient than the OoO model. Hmmer shows a 70% reduction in branch prediction energy because of its larger average code block size.

Fig. 17. Static and dynamic EPI normalized to OoO.

Fig. 18. Average code block size of SPEC Int 2006 benchmarks.

Fig. 19. The BPU access EPI normalized to OoO.

6.2.2 Register File Hierarchy. The CG-OoO register file hierarchy contributes to the processor energy savings in four different ways, each of which is discussed here:

- LRFs are low-energy tables.
- Segmented GRF reduces access energy.
- Local operands bypass register renaming.
- Register renaming is optimized to reduce on-chip data movement.

Local registers are statically managed and account for 30% of the total data communication. The 20-entry LRF energy per access is about 25× smaller than that of a unified, 250-entry register file in the baseline OoO processor. The LRF has two read and two write ports, and the unified register file has eight read and four write ports. In addition, since each BW holds an LRF near its instruction window and execution units, operand reads and writes take place over shorter average wire lengths. LRFs enable additional energy saving by avoiding local write-operand wakeup broadcasts. Figure 20 shows the contribution of the local register file energy compared to the OoO baseline; it shows an average 26% reduction in register file energy consumption due to local register accesses. The global register file used in both OoO and CG-OoO has 250 entries. While the use of local registers enables building a smaller global register file without a noticeable impact on performance, our experiments use equal global register file sizes for consistent energy and performance modeling between CG-OoO and OoO.

To reduce the access energy overhead of a unified register file, and to increase the aggregate number of ports in the CG-OoO, this processor breaks the global register file (GRF) into 12 segments. Each segment is placed next to a BW. Figure 20 also shows the contribution of the global register file access energy compared to the OoO baseline; it shows an average 92% reduction in

Fig. 20. The register file (RF) access EPI normalized to OoO. The CG-OoO RF hierarchy is 95% more efficient than that of OoO. S-GRF 12 shows the energy of a CG-OoO GRF with 12 segments, and RF shows the energy of an InO RF with the same number of ports as OoO.

Fig. 21. The S-GRF access EPI normalized to one Segment GRF.

Fig. 22. The register rename access EPI normalized to OoO.

Fig. 23. The instruction issue EPI normalized to OoO.

the global register file energy consumption due to segmentation. Figure 21 compares the case of a unified GRF, one GRF segment per cluster (for a three-cluster CG-OoO), and one GRF segment per BW. As the number of register segments increases, energy consumption decreases proportionally. GRF and wakeup/bypass communications across BW components within a cluster and across different clusters take one and two cycles, respectively. Furthermore, because local register operands are statically allocated, they do not require register renaming. Thus, 14% average energy reduction is observed in the register rename stage (Figure 22).

6.2.3 Instruction Scheduling. The CG-OoO processor introduces the Skipahead issue model. In OoO and CG-OoO, in-flight instructions are maintained in queues that are partly RAM and partly CAM tables (González et al. 2010). For the InO model, instructions are held in a small FIFO buffer. Figure 23 shows the energy breakdown of the dynamic scheduling hardware. On average, the

Fig. 24. The commit EPI normalized to OoO.

majority of the OoO scheduling energy (65%) is in reading and writing instructions from the RAM table; another 28% of the energy is in CAM table accesses. The “Rest” energy is consumed by stage registers and the interconnects used for instruction wakeup and select.

This figure also indicates that an 87% average reduction in the CG-OoO RAM table energy (relative to OoO RAM energy) is due to accessing smaller SRAM tables, and 94% average reduction in the CAM table energy is due to using two- to five-entry Skipahead Buffers instead of a 64-entry instruction queue CAM table in OoO. The “Rest” average energy is increased by 56% due to the additional pipeline registers present at the issue stage. Overall, the CG-OoO issue stage is 80% more efficient than OoO. This energy improvement is achieved through designing a processor that targets average execution performance, a type of design desired in embedded applications. The average number of CG-OoO in-flight instructions is 36% higher than OoO, and at the peak performance the number of in-flight instructions is 6% lower due to the limited view of the Skipahead scheduler to the pipelined instructions. BLP, however, compensates for the limited view of Skipahead. The average number of in-flight BWs is 4.6.

6.2.4 Block Reorder Buffer. The CG-OoO processor maintains program order at block-level granularity. This makes BROB more energy efficient than the baseline ROB for the following three reasons:

- Fewer BROB accesses: one entry is allocated/deallocated per code block as opposed to one entry per instruction in OoO. BROB allocations are done after decoding each head and block deallocations are done at the block commit stage.
- Lower BROB utilization: local write register identifiers are not stored in BROB as they are not dynamically register renamed.
- Cheaper BROB accesses: because the BROB is designed to maintain program order at block granularity, it is provisioned to have 16 entries rather than 180 entries used for OoO (Corporation 2009). This makes BROB 54% smaller than the baseline ROB and thereby cheaper to access.

Figure 24 shows 41% average energy savings for CG-OoO commit.

6.3 Clustering and Scaling Analysis

The CG-OoO architecture focuses on reducing processor energy through designing a complexity-effective architecture. To remain competitive with the OoO performance, this architecture supports a larger number of EUs. To achieve both objectives, CG-OoO employs a clustering design strategy that is more scalable than the OoO. A cluster consists of a number of BWs sharing a few EUs. This section illustrates the effect of changing these resources on a three-cluster CG-OoO processor.

Figures 25(a) and 25(b) show the normalized average performance (IPC) and energy (EPI) of SPEC Int 2006 benchmarks versus the number of BWs per cluster for various numbers of EUs per cluster. The speedup figure shows that two of the configurations with six BWs per cluster reach the performance of OoO. All clustering models exhibit lower energy consumption overhead compared

Fig. 25. Normalized performance and energy for various clustering configurations, all assuming a three-cluster processor. Multiplying the numbers in the figure by 3 gives the total number of BWs and EUs.

Fig. 26. The energy-performance proportionality attribute of CG-OoO is illustrated here. The CG-OoO design points correspond to those in Figure 25.

to the OoO design. The most energy-efficient configuration is the one with {1 BW, 1 EU} per cluster; it is 56% more energy efficient than the OoO, but only at 63% of the OoO performance. The most high-performance configuration evaluated is the one with {6 BW, 8 EU} per cluster; it is 28% more energy efficient than the OoO at the same performance level. The design configuration evaluated throughout this article corresponds to the {4 BW, 4 EU}-per-cluster configuration where the OoO and CG-OoO areas match.

Figure 26 shows the CG-OoO energy-performance (EP) characteristics plotting all the cluster configurations presented earlier. The highest EP point, in the plot, refers to the {6 BW, 8 EU}-per-cluster configuration, and the lowest EP point refers to the {1 BW, 1 EU}-per-cluster configuration. The triangle with black edges corresponds to the {4 BW, 4 EU}-per-cluster configurations evaluated in the previous sections. As the processor resource complexity decreases, energy and performance properties drop proportionally. That is, depending on the system and runtime requirements, processor recourses can be turned on/off to deliver the right level of performance at a proportional energy cost. Beyond a certain scaling point, the wakeup/select and load-store unit wire latencies become so large that the CG-OoO energy-performance proportionality breaks. Identifying the breaking point is outside the scope of this work.

7 CONCLUSION

Unlike most previous studies, CG-OoO is an execution model designed from an energy efficiency standpoint. CG-OoO closes 62% of the average energy gap between the InO and OoO baseline processors at the same area and nearly the same performance as the OoO. CG-OoO is designed to meet the OoO nominal performance while trading off the peak scheduling performance for

superior energy efficiency. The key energy efficiency enablers of this processor are (1) its end-to-end block-level, complexity-effective design and (2) its use of compiler assistance for code clustering and instruction scheduling. CG-OoO evaluates an energy-efficient block-level speculation and dynamic scheduling model. It proposes Skipahead, an energy-efficient, limited out-of-order instruction issue model that decouples the window of instructions exposed to the dynamic scheduler from the window of in-flight instructions. Furthermore, CG-OoO evaluates an energy-efficient register file hierarchy that reduces register renaming events. Scaling the CG-OoO resources exhibits energy-performance proportionality.

REFERENCES

- Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. 2003. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003 (MICRO-36)*. IEEE, 423–434.
- Apple. 2006. Apple Rosetta Software. Retrieved September 20, 2017, from <https://www.apple.com/la/rosetta>.
- Omid Azizi, Aqeel Mahesri, Benjamin C. Lee, Sanjay J. Patel, and Mark Horowitz. 2010. Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis. In *ACM SIGARCH Computer Architecture News*, Vol. 38. ACM, 26–36.
- Ronald D. Barnes, Shane Ryoo, and Wen-mei W. Hwu. 2005. Flea-flicker multipass pipelining: An alternative to the high-power out-of-order offense. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 319–330.
- Luiz André Barroso and Urs Hölzle. 2007. The case for energy-proportional computing. *Computer* 12 (2007), 33–37.
- Sarah Bird, Aashish Phansalkar, Lizy K. John, Alex Mericas, and Rajeev Indukuru. 2007. Performance characterization of SPEC CPU benchmarks on Intel's core microarchitecture based processor. In *SPEC Benchmark Workshop*.
- Edward Brekelbaum, Jeff Rupley, Chris Wilkerson, and Bryan Black. 2002. Hierarchical scheduling windows. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society Press, 27–36.
- Doug Burger and Stephen W. Keckler. 2005. 19.5: Breaking the GOP/Watt barrier with EDGE architectures. In *GOMACTech Intelligent Technologies Conference*.
- Doug Burger, Stephen W. Keckler, Kea McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, and William Yoder. 2004. Scaling to the end of silicon with EDGE architectures. *Computer* 37, 7 (2004), 44–55.
- Y. U. Cao, Takashi Sato, Dennis Sylvester, Michael Orshansky, and C. Hu. 2002. Predictive technology model. Retrieved from <http://ptm.asu.edu>.
- Trevor E. Carlson, Wim Heirman, Osman Allam, Stefanos Kaxiras, and Lieven Eeckhout. 2015. The load slice core microarchitecture. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 272–284.
- Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. 1992. Low-power CMOS digital design. *IEICE Transactions on Electronics* 75, 4 (1992), 371–382.
- Intel Corporation. 2009. Intel 64 and IA-32 Architectures Optimization Reference Manual.
- Neal Clayton Crago and Sanjay Jaram Patel. 2011. OUTRIDER: Efficient memory latency tolerance with decoupled strands. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 117–128.
- Adrian Cristal, Daniel Ortega, Josep Llosa, and Mateo Valero. 2004. Out-of-order commit processors. *Software, IEEE Proceedings*. IEEE, 48–59.
- Adrian Cristal, Mateo Valero, J. L. Llosa, and Antonio Gonzalez. 2002. *Large Virtual ROB's by Processor Checkpointing*. Technical Report UPC-DAC-2002-39, Universitat Politècnica de Catalunya.
- Kenneth Czechowski, Victor W. Lee, Ed Grochowski, Ronny Ronen, Ronak Singhal, Richard Vuduc, and Pradeep Dubey. 2014. Improving the energy efficiency of big cores. In *Proceeding of the 41st Annual International Symposium on Computer Architecture*. IEEE Press, 493–504.
- Bill Dally. 2011. Project denver. *Processor to Usher in New Era of Computing*. Retrieved from <http://blogs.nvidia.com/blog/2011/01/05/project-denver-processor-to-usher-in-new-era-of-computing/>.
- Subhasis Das, Tor M. Aamodt, and William J. Dally. 2015. SLIP: Reducing wire energy in the memory hierarchy. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, 349–361.
- James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. 2003. The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. IEEE Computer Society, 15–24.

- Antonio González, Fernando Latorre, and Grigorios Magklis. 2010. Processor microarchitecture: An implementation perspective. *Synthesis Lectures on Computer Architecture* 5, 1 (2010), 1–116.
- Linley Gwennap. 1997. MIPS R12000 to hit 300 MHz. *Microprocessor Report* 11, 13 (1997), 1.
- Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. SimPoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism* 7, 4 (2005), 1–28.
- Eric Hao, Po-Yung Chang, Marius Evers, and Yale N. Patt. 1998. Increasing the instruction fetch rate via block-structured instruction set architectures. *International Journal of Parallel Programming* 26, 4 (1998), 449–478.
- John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.
- Andrew Hilton, Santosh Nagarakatte, and Amir Roth. 2009. iCFP: Tolerating all-level cache misses in in-order processors. In *Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture, 2009 (HPCA'09)*. IEEE, 431–442.
- Andrew Hilton and Amir Roth. 2010. BOLT: Energy-efficient out-of-order latency-tolerant execution. In *Proceedings of the 2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA'10)*. IEEE, 1–12.
- Glenn Hinton, Dave Sager, Mike Upton, and Darrell Boggs. 2001. The microarchitecture of the Pentium® 4 processor. *Intel Technology Journal*. Citeseer.
- Wei Huang, Shougata Ghosh, Sivakumar Velusamy, Karthik Sankaranarayanan, Kevin Skadron, and Mircea R. Stan. 2006. HotSpot: A compact thermal modeling methodology for early-stage VLSI design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 14, 5 (2006), 501–513.
- Richard E. Kessler, Edward J. McLellan, and David A. Webb. 1998. The alpha 21264 microprocessor architecture. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors, 1998 (ICCD'98)*. IEEE, 90–95.
- K. Khubaib, M. Aater Suleman, Milad Hashemi, Chris Wilkerson, and Yale N. Patt. 2012. Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. IEEE, 305–316.
- H.-S. Kim and James E. Smith. 2002. An instruction set and microarchitecture for instruction level distributed processing. In *Proceedings of the 29th Annual International Symposium on Computer Architecture, 2002*. IEEE, 71–81.
- Etienne Le Sueur and Gernot Heiser. 2010. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*. 1–8.
- Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the IEEE/ACM Symposium on Microarchitecture (MICRO'09)*. IEEE, 469–480.
- Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices* 40, 6 (2005), 190–200.
- Stephen W. Mahin, Stephen M. Connor, Stephen J. Ciavaglia, Lyman H. Moulton III, Stephen E. Rich, and Paul D. Kartschoke. 1997. Superscalar instruction pipeline using alignment logic responsive to boundary identification logic for aligning and appending variable length instructions to instructions stored in cache. April 29 1997. US Patent 5,625,787.
- Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. 1992. Effective compiler support for predicated execution using the hyperblock. In *ACM SIGMICRO Newsletter*, Vol. 23. IEEE Computer Society Press, 45–54.
- Daniel S. McFarlin, Charles Tucker, and Craig Zilles. 2013. Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism? In *ACM SIGPLAN Notices*, Vol. 48. ACM, 241–252.
- Milad Mohammadi, Song Han, Tor M. Aamodt, and William J. Dally. 2015. On-demand dynamic branch prediction. *IEEE Computer Architecture Letters* 14, 1 (2015), 50–53.
- Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture, 2003 (HPCA-9'03)*. IEEE, 129–140.
- Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. 2015. Exploring the potential of heterogeneous von Neumann/dataflow execution models. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, 298–310.
- Subbarao Palacharla, Norman P. Jouppi, and James E. Smith. 1997. *Complexity-Effective Superscalar Processors*. Vol. 25. ACM.
- Il Park, Chong Liang Ooi, and T. N. Vijaykumar. 2003. Reducing design complexity of the load/store queue. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 411.
- Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. 2004. Pinpointing representative portions of large Intel® Itanium® programs with dynamic instrumentation. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 81–92.

- Charles Price. 1995. MIPS IV instruction set.
- Glenn Reinman, Todd Austin, and Brad Calder. 1999a. A scalable front-end architecture for fast instruction delivery. In *ACM SIGARCH Computer Architecture News*, Vol. 27. IEEE Computer Society, 234–245.
- Glenn Reinman, Todd Austin, and Brad Calder. 1999b. A scalable front-end architecture for fast instruction delivery. In *ACM SIGARCH Computer Architecture News*, Vol. 27. IEEE Computer Society, 234–245.
- Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. 1997. Trace processors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture, 1997*. IEEE, 138–148.
- Pierre Salverda and Craig Zilles. 2008. Fundamental performance constraints in horizontal fusion of in-order cores. In *Proceedings of the IEEE 14th International Symposium on High Performance Computer Architecture, 2008 (HPCA'08)*. IEEE, 252–263.
- Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. 2003. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture, 2003*. IEEE, 422–433.
- André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. 2002a. Design tradeoffs for the alpha EV8 conditional branch predictor. In *Proceedings of the IEEE/ACM Symposium on Computer Architecture (ISCA'02)*. 295–306.
- André Seznec, Eric Toullec, and Olivier Rochecouste. 2002b. Register write specialization register read specialization: A path to complexity-effective wide-issue superscalar processors. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society Press, 383–394.
- Tingting Sha, Milo M. K. Martin, and Amir Roth. 2006. Nosq: Store-load communication without a store queue. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 285–296.
- Faissal M. Sleiman and Thomas F. Wenisch. 2016. Efficiently scaling out-of-order cores for simultaneous multithreading. In *Proceedings of the IEEE/ACM Symposium on Computer Architecture (ISCA'16)*. 431–443.
- Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. 1995. Multiscalar processors. In *ACM SIGARCH Computer Architecture News*, Vol. 23. ACM, 414–425.
- Eric Sprangle and Yale Patt. 1994. Facilitating superscalar processing via a combined static/dynamic register renaming scheme. In *Proceedings of the 27th Annual International Symposium on Microarchitecture, 1994 (MICRO-27)*. IEEE, 143–147.
- Srikanth T. Srinivasan, Ravi Rajwar, Haitham Akkary, Amit Gandhi, and Mike Upton. 2004. Continual flow pipelines. *ACM SIGPLAN Notices* 39, 11 (2004), 107–119.
- Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. 2003. WaveScalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 291.
- Joel M. Tendler, J. Steve Dodson, J. S. Fields, Hung Le, and Balaram Sinharoy. 2002. POWER4 system microarchitecture. *IBM Journal of Research and Development* 46, 1 (2002), 5–25.
- Keil Tool. 2015. ARM registers. Retrieved August 24, 2015, from http://www.keil.com/support/man/docs/armasm/armasm_dom1359731128950.htm.
- Francis Tseng and Yale N. Patt. 2008. Achieving out-of-order performance with almost in-order complexity. In *35th International Symposium on Computer Architecture, 2008 (ISCA'08)*. IEEE, 3–12.
- Jessica H. Tseng and Krste Asanović. 2003. Banked multiported register files for high-frequency superscalar microprocessors. *ACM SIGARCH Computer Architecture News* 31, 2 (2003), 62–71.
- Yasuko Watanabe, John D. Davis, and David A. Wood. 2010. WiDGET: Wisconsin decoupled grid execution tiles. In *ACM SIGARCH Computer Architecture News*, Vol. 38. ACM, 2–13.
- Ahmad Zmily and Christos Kozyrakis. 2005. Energy-efficient and high-performance instruction fetch using a block-aware ISA. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*. ACM, 36–41.
- Ahmad Zmily and Christos Kozyrakis. 2006. Simultaneously improving code size, performance, and energy in embedded processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 224–229.
- Victor V. Zyuban. 2000. *Inherently Lower-Power High-Performance Superscalar Architectures*. Ph.D. Dissertation. University of Notre Dame.

Received January 2017; revised September 2017; accepted October 2017