

# RingBOOM: An Implementation of a Novel High-Performance Banked Microarchitecture

*Benjamin Korpan*



Electrical Engineering and Computer Sciences  
University of California at Berkeley

Technical Report No. UCB/Eecs-2020-177

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/Eecs-2020-177.html>

August 21, 2020

Copyright © 2020, by the author(s).  
All rights reserved.

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

---

**RingBOOM: An Implementation of a Novel High-Performance Banked  
Microarchitecture**

by Ben Korpan

---

Submitted to the Department of Electrical Engineering and Computer Sciences,  
University of California at Berkeley, in partial satisfaction of the requirements for the  
degree of **Master of Science, Plan II**.

Approval for the Report:

**Committee:**

---

Professor Krste Asanovic  
Research Advisor

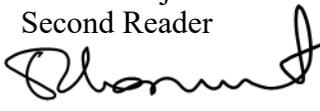
Signature: 

Date: 8/20/2020

\*\*\*\*\*

---

Professor Borivoje Nikolic  
Second Reader

Signature: 

Date: 8/20/2020

# RingBOOM: An Implementation of a Novel High-Performance Banked Microarchitecture

Ben Korpan

bkorpan@berkeley.edu

University of California, Berkeley

## ABSTRACT

Out-of-order superscalar microarchitectures are composed of structures which scale poorly with pipeline width in power, cycle time, and area. Many techniques have been proposed to reduce the cost of these structures, some of which have been applied to real microprocessors. A previously proposed banked microarchitecture called "RingScalar" [1] enables large reductions in the complexity of these structures by tailoring the architecture to common patterns in instruction streams, but has never been implemented. This work focuses on an implementation ("RingBOOM") of the RingScalar microarchitecture in the open-source Berkeley Out-of-Order Machine (BOOM). IPC degradation compared to the current BOOM microarchitecture is measured: a relative IPC of 0.82 was observed. Assessment of cycle time and area is then performed with a commercial FinFET technology: RingBOOM core variants achieved synthesized frequencies of up to 2.1 GHz and were about 25% smaller than the reference design.

## 1 INTRODUCTION

Out-of-order execution was first introduced in the floating-point unit of the IBM System/360 Model 91 [2]. It later resurfaced in microprocessors as a microarchitectural means to increase performance, primarily by hiding memory latency. Early examples of its application in microprocessors include the MIPS R10000 [3] and Alpha 21264 [4]. Such microarchitectures require large, complex structures to keep track of dependencies between instructions, maintain the illusion of in-order execution by providing facilities to undo incorrect execution, and manage the allocation of entries in these structures. The poor scaling of many of these structures motivates the investigation of alternative core organizations. Note that this report focuses solely on physical register file (PRF) based designs. For background, below are important components of an 'ideal', fully-ported PRF execution core. Attention is drawn towards their scaling in an  $N$ -wide machine. The Load/Store unit is considered separate from the execution core, and will be considered in later work.

## 2 OUT-OF-ORDER MICROARCHITECTURE BACKGROUND

### 2.1 Re-Order Buffer

The re-order buffer (ROB) provides the illusion of in-order execution by allowing incorrect execution to be undone. It is a circular buffer that contains metadata for all in-flight instructions in program-order. When an exception is thrown, the ROB is able to "rollback" the state of the machine from the tail to head, reaching the correct architectural state. The ROB is also responsible for microarchitectural bookkeeping: when a physical register may no longer be accessed,

it is returned to the free list. The ROB size can be roughly expressed as  $E = S \times N$ , where  $E$  is the number of entries and  $S$  is the maximum speculation depth in cycles. Designers of wider machines will likely wish to maintain the same maximum speculation depth, so  $S$  can be viewed as a constant. Since the ROB can be trivially implemented as a collection  $N$  banks, it scales as  $O(N)$  and is not a target for optimization by this work.

### 2.2 Issue Queues

The issue unit or scheduler is composed of one or multiple issue queues: here we will consider a unified issue queue which holds all in-flight, unexecuted instructions. The queue holds these dispatched instructions in issue slots, attempting to issue them on an oldest-first basis as soon as their operands are available. This selection logic may be implemented as a cascade of single priority selectors, or a monolithic first- $N$  selector using a one-hot counter tree. As operands are often unavailable at dispatch, the issue slots must watch the register file writeback ports for the appearance of their instruction's operands. The appearance of such an operand is called a 'wakeup', because it notifies dependent instructions in the issue queues, allowing them to proceed to execution. In addition to the writeback ports, the issue slots may also perform wakeups against bypassable scheduled instructions, allowing their instructions to execute sooner than if they were to wait for the writeback of their operands. In BOOM, this is referred to as a 'fast' wakeup, while the type that occurs at writeback is a 'slow' wakeup. The dependency between issue selection and fast wakeups is referred to as the select-wakeup loop, and constitutes an important microarchitectural loop [5]. As in the ROB, the total number of issue queue entries will typically scale with  $N$ . As each entry must compare its operands to  $2N$  wakeup ports ( $N$  fast +  $N$  slow), the total number of wakeup comparators grows as  $O(N^2)$ . This may become a significant area and/or power contribution in a wide machine.

### 2.3 Physical Register File

The physical register file (PRF) is a very large component of an out-of-order execution core. At any point in time the PRF contains the committed architectural register state, as well as a larger set of in-flight results. An  $N$ -wide machine requires its PRF have at least  $2N$  read ports and  $N$  write ports. As with the above two structures, a rule of thumb indicates that the number of physical registers should be concomitant with the pipeline width, allowing an appropriate number of in-flight results. Note that the area of a register file structure tends to scale as  $O(k^2)$  for  $k$ -ports [6]. Thus, the PRF may scale as poorly as  $O(N^3)$ .

## 2.4 Bypass Network

The bypass network shortens instruction execution latency by skipping a sequential register file write and read. It accomplishes this by connecting each execution unit output each input through operand selection multiplexers. It thus scales as  $O(N^2)$ .

## 2.5 Load/Store Queues

Load and store queues maintain the in-order behavior of loads and stores in an out-of-order machine. Traditional load/store queue designs scale very poorly with pipeline width due to their read/write port requirements and use of address CAMs to detect dependencies between inflight memory operations. This work focuses solely on the execution core, with the load/store unit currently out-of-scope. Future work (section 7) may address scalable load/store queues.

## 3 PRIOR WORK

Many techniques have been proposed to combat the inherent complexity of superscalar out-of-order microarchitectures. The goal is to enable wider machines without compromising cycle time or causing an explosion in area and power consumption. These techniques generally introduce additional structural hazards beyond those posed by structure capacities in the above ideal machine.

### 3.1 Clustering

Clustering is a technique wherein a large execution core is subdivided into multiple smaller, fewer-ported segments. A common form of clustering which was employed in the Alpha 21264 [4] hinges on replication of the PRF between clusters. In a  $N$ -wide,  $k$ -cluster machine this allows a factor of  $k$  reduction in read ports on each cluster PRF, but still requires  $N$  write ports per cluster to maintain a consistent replication. Another form of clustering utilized by microarchitectures such as Multicluster [7] avoids replication by dividing the architectural and physical register files between clusters. Unlike the banked register files described in the next section, this technique requires complex logic to map instructions to clusters and provide inter-cluster communication.

### 3.2 Banked Register Files

Banked register files [6] allow a reduction in register file read ports in a centralized microarchitecture, without reliance on replication. In this scheme the set of physical registers is statically partitioned into banks; the port-count per bank can vary independently of the total number of ports required, posing a tradeoff between area and collision rate. A simple but effective scheme for handling collisions is described in [8]: groups of issued instructions which incur read or write bank conflicts are killed, along with the subsequent issue group which may have received wakeups from the killed group. This is an alternative to complex stall logic, and does not significantly effect the delay of the select-wakeup loop. However, it does add an additional pipeline stage between issue and register read, increasing the branch misprediction penalty and incurring a high bank conflict penalty (up to  $2N$  instructions).

Two optimizations are presented for this scheme: bypass-skip and read-sharing. Bypass-skip prevents the request of a register read when an operand is known to be bypassable. In the simplest implementation, this occurs if the operand was just woken by a

single-cycle latency instruction issued in the previous cycle. Read-sharing prevents bank conflicts in the case that multiple requesters access the same register: the read port which produces the operand drives both requesters.

## 3.3 RingScalar

RingScalar [1] is a proposed microarchitecture which builds on the banked register files. It avoids writeback bank collisions through its physical register allocation policy. Additionally, it allows the number of dispatch, issue and wakeup ports in the issue queues to be greatly reduced. This reduces the queue area and latency of the select-wakeup loop. Finally, it decreases the size of the bypass network, with each pipeline slot receiving and driving only a single bypass path.

An  $N$ -wide RingScalar-style machine is divided into  $N$  execution columns. Its name is derived from the unidirectional ring which wraps around the columns. The wakeup ports and bypass network form a ring: scheduled instructions wakeup dependents in the subsequent column which, if issued immediately, may receive the data from the bypass path. All of the discussed structures scale as  $O(N)$  in this scheme. However, it requires several additional crossbars which scale as  $O(N^2)$ . These may pose a problem when attempting to scale to extremely wide pipeline widths.

RingScalar forms the basis of the implemented microarchitecture. The details are further described as-implemented in section 4.

## 4 MICROARCHITECTURE

This section will discuss "RingBOOM": an implementation of the RingScalar microarchitecture. The implementation was performed in Chisel hardware construction language [9], involving several thousand lines worth of additions and modifications on a fork of the Berkeley Out-of-Order Machine (BOOM) [10]. BOOM is an open-source out-of-order implementation of the RISC-V ISA [11].

### 4.1 Rename

One tradeoff incurred by the RingScalar microarchitecture is the increased complexity of the rename and dispatch stages. This is due to the physical register allocation policy: an instruction with a busy operand must be mapped to a column following that which produces the operand. Thus, the allocation of physical registers is delayed until after the reading of the map table, busy table and column selection logic. A 2-stage "column-steering" rename pipeline is pictured in figure 2. To achieve the most aggressive frequency in section 5.2, a third stage needed to be added.

**4.1.1 Column Selection.** Column selection is performed by a module called the column arbiter, pictured in figure 1. The request mechanism is implemented by the boxes labelled '?' in the figure, and is summarized by table 1, where 'xxx' denotes a 'don't care' signal value. Zero-waiting instructions (instructions which are not waiting on either operand) are assigned to a random column, while one-waiting instructions are assigned the column following the producer of the busy operand. Two-waiting instructions must be placed in a column following one of the producers of their source operands. For simplicity, this is done by prioritizing the left operand, prs1: this is important for quick translation of store addresses to prevent memory ordering failures. Also note the intra-cycle bypassing

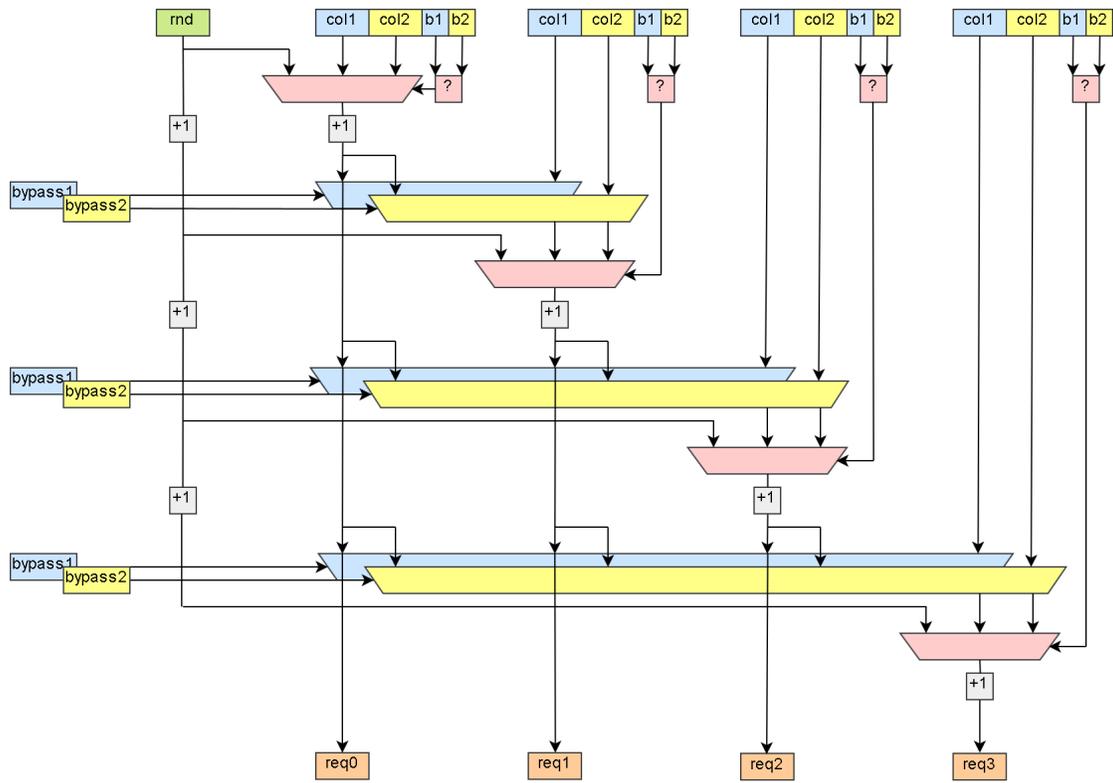


Figure 1: The Column Arbiter

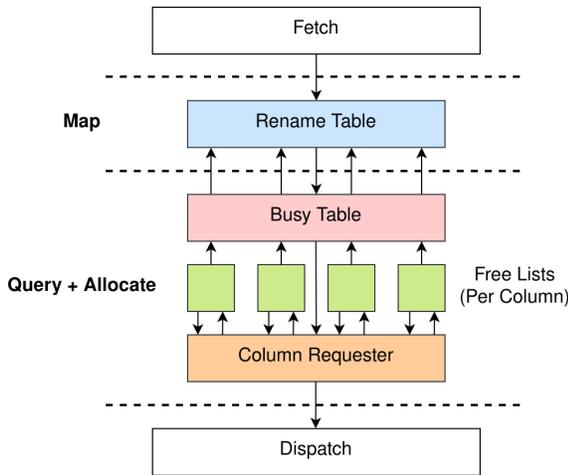


Figure 2: 2-stage Column-Steering Rename Pipeline

which is performed between column selections, ensuring each instruction sees the columns requested by older instructions. Finally, note the '+1' incrementation operations: these are implemented as rotations of one-hot vectors.

4.1.2 *Load-Dependent Operands.* To implement the load-hit bypassing scheme described in section 4.5, physical registers which

are produced by loads are kept track of in a new "load table" in parallel with the busy table. Instructions that are solely load-dependent receive their wakeup signals and bypass data directly from the load/store unit, and can therefore be flexibly assigned to any column.

Table 1: Column Request Logic

prs1 busy?	prs2 busy?	request
no	no	<i>random</i>
no	yes	<i>col2 + 1</i>
yes	xxx	<i>col1 + 1</i>

## 4.2 Dispatch

An additional pipeline stage was added following rename, giving instructions an entire cycle to dispatch into out-of-order structures. The pipeline register was implemented as a two-entry compacting queue: this prevents dispatch logic from directly backpressuring the rename pipeline, improving QoR.

During dispatch, a crossbar provides the  $N$  integer issues queues with instructions from rename. The number of instructions each queue can receive per cycle is parameterized: a value of 2 was found to achieve a combination of good IPC and QoR.

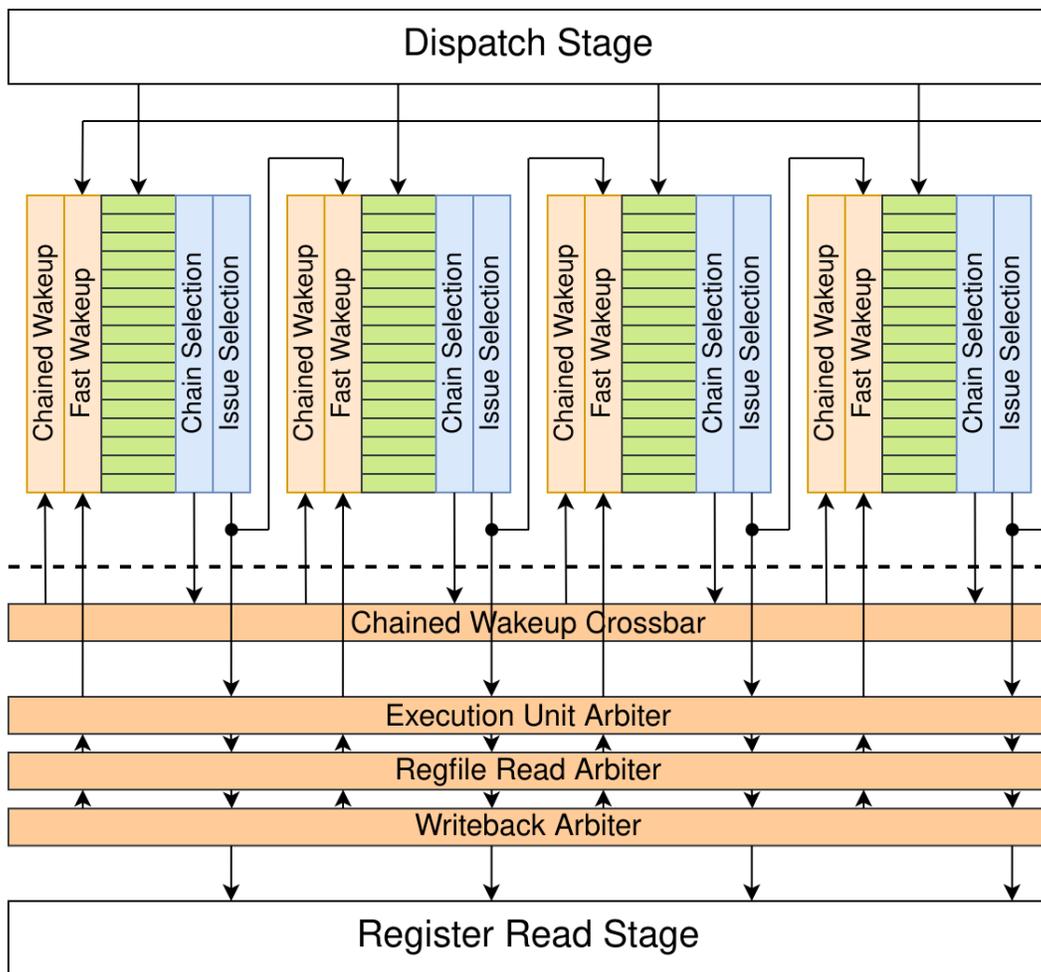


Figure 3: Scheduler with Issue Arbiters and Chained Wakeups

### 4.3 Issue

Two schemes were discussed in [1] to greatly reduce the number of required wakeup ports in the issue queues. Selected for implementation was the scheme which splits two-waiting instructions into two  $\mu$ ops at dispatch: the second  $\mu$ op is a 'dummy' which catches the wakeup of the second busy operand. This style of issue queue and issue slot are pictured in figures 3 and 4 respectively. Four wakeup ports are needed per slot: fast wakeups for bypassable operations, a speculative load port, a slow port and a port which receives the chained wakeups. Advantages of this scheme are that it both preserves the regularity of the issue queue structure and allows it to scale as  $O(N)$ .

Arbiters were implemented to prevent collisions in utilizing execution resource. These arbiters initially sat in the wakeup-select loop, simply revoking a grant and fast-wakeup signal in the case of a collision. As the arbiters became more complex with multiple load/store pipelines and more flexible read-port allocation, arbitration needed to be performed in new arbitration pipeline stage following issue selection.

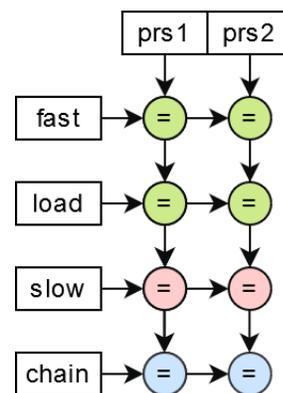


Figure 4: Issue Slot with Chained Wakeup Port

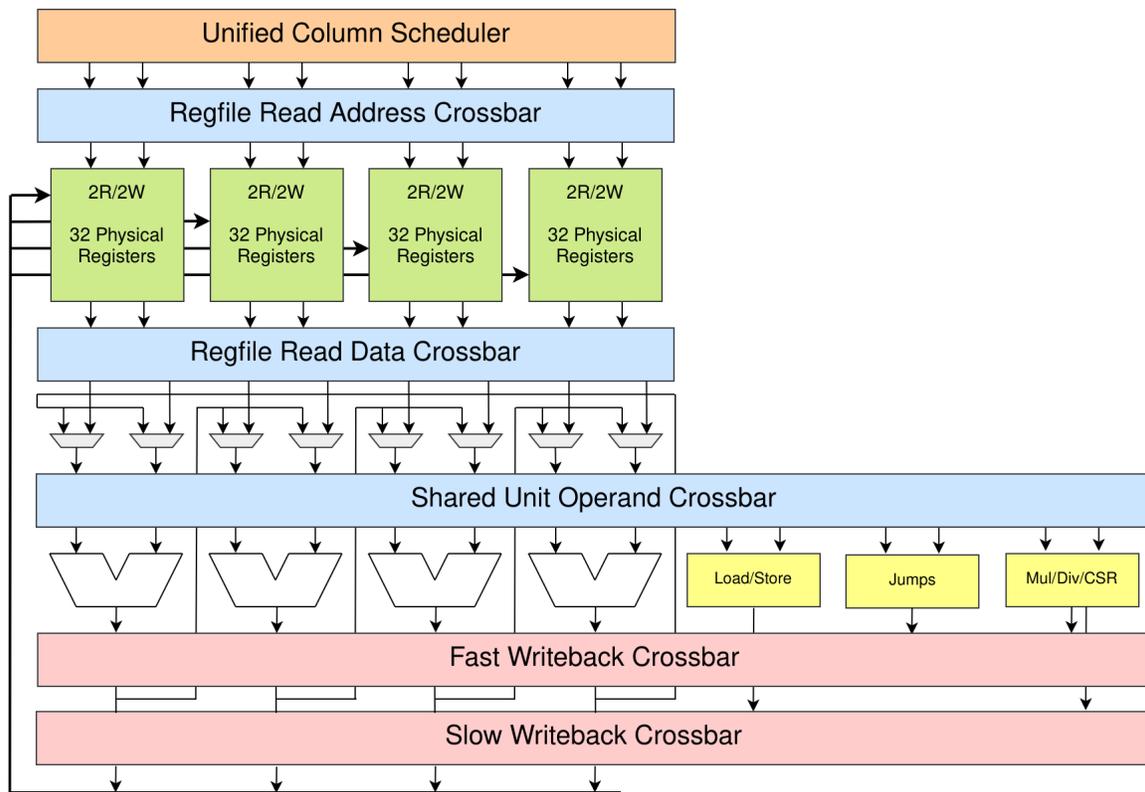


Figure 5: RingBOOM Execution Unit, Register File and Writeback Organization

#### 4.4 Register File Read

Register file read port reduction should contribute the largest area advantage offered by this microarchitecture. As with the banked register files described in section 3.2, bank conflicts must be detected and handled. This was previously being done in the issue stage to simplify implementation and minimize IPC loss. The latest version of the microarchitecture adds a dedicated pipeline stage for arbitration. An instruction which loses in arbitration is killed back into its issue slot, and any operands it woke up in the next column are reverted back to an unready state. The number of register file read ports per bank is parameterized: values of 2 and 3 were found to work well with a 4-wide pipeline. The read ports are flexibly allocated to requesting operands, and are only requested when an operand is needed and not able to be bypassed. A configuration with two read ports per column is pictured in figure 5.

#### 4.5 Bypass

Each operand register can select from its register file port or a handful of bypass paths, as pictured in figure 6. This includes a single-cycle ALU bypass path from the previous column, a path from the previous column’s fast writeback crossbar output, and load-hit bypass path(s). Notice that combinational ALU outputs each drive only one single-cycle ALU bypass path: this constitutes a speedup of an important microarchitectural loop. Additionally, note that the number of bypass paths for each operand register

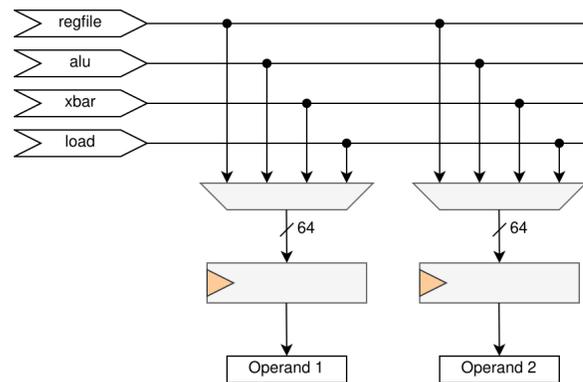


Figure 6: Bypass Selection for Execution Operand Registers

to choose from is constant with pipeline width, unless additional load-hit bypass paths are added. Currently, only one or two load-hit bypass paths are being used.

#### 4.6 Execution Units

The operand registers fan out to their respective column ALUs, as well as a shared unit input crossbar. BOOM contains a set of heterogeneous execution units, which are mostly homogeneous in their scheduling, register read and writeback port provisions.

RingScalar requires a more restricted execution unit organization. Each column contains an identical ALU, with more expensive or infrequently used units being shared between columns. Arbitration over shared units is performed after issue in parallel with the above mentioned read port arbitration and below mentioned writeback collision logic. Execution unit organization is pictured in figure 5.

## 4.7 Writeback

The presence of shared execution units necessitated a more sophisticated writeback mechanism. This mechanism has to arbitrate between shared units and column ALUs over writeback ports while retaining the bypassability of load-hits from a shared memory unit. The following scheme has been devised, which uses two writeback crossbars. Fully pipelined units with predictable latencies use the fast writeback crossbar. Collisions between fast-writeback instructions are prevented in the issue stage, so no stall logic is required on this crossbar. As these units all have predictable latencies, the output of this crossbar can be bypassed. On the other hand, units which are unpipelined (division) or exhibit unpredictable latencies (loads) use the slow writeback crossbar. The slow writeback crossbar accesses a separate set of write ports on each register file bank: the banks thus have two write ports, as pictured in figure 5. Load-hits are directly bypassed as described in section 4.5.

## 5 RESULTS

### 5.1 Per-cycle Performance

IPC was compared with the latest mainline BOOM core, SonicBOOM [12], in CoreMark [13] and SPEC CPU 2017 [14]. The same branch predictor was used, while execution structure sizes and memory pipelines were configured as similarly as possible. Both the evaluated version of RingBOOM and SonicBOOM feature a 4-wide rename/dispatch pipeline. RingBOOM was configured with 3 read ports per register file bank. Degradation in IPC was expected, as a fully-ported microarchitecture was being compared to a banked microarchitecture with novel optimizations. The goal was to minimize this degradation while improving physical characteristics.

All simulations were performed with FireSim [15] using AWS F1.2x FPGA instances. The clockspeeds of the FPGA simulated RingBOOM variants ranged from 50 MHz to 100 MHz, largely contingent on memory system configuration. These fast simulator clockspeeds permit the use of long-running benchmarks such as SPEC, whose constituents each take over 1 trillion cycles to run.

RingBOOM achieved a CoreMark performance of 5.1 CoreMark/MHz, 82% of SonicBOOM’s 6.2 CoreMark/MHz. This is precisely in-line with the harmonic mean of the SonicBOOM-normalized SPEC results shown in figure 7. In general, benchmarks with higher available ILP (such as 625.x264) tended to transfer worse to RingBOOM, likely due to a higher frequency of shared resource conflicts.

**Table 2: IPC Summary Relative to SonicBOOM**

Benchmark	Normalized Score
CoreMark	0.82
SPEC2017 HARMEAN	0.82

## 5.2 Synthesis Results

Several variants of RingBOOM’s integer execution core were synthesized on a slow, low-voltage corner of a commercial FinFET process. A maximum frequency of 2.1 GHz was achieved after implementing many experimental QoR tweaks. This is compared to an initial frequency of 1.1 GHz for RingBOOM and 0.86 GHz for the default 4-wide version of SonicBOOM’s integer core. Additionally, an area reduction of around 25% was observed.

**Table 3: Execution Core Structure Configuration**

Structure	Size
ROB Entries	128
PRF Registers	128
INT Issue Window Size	32
Branch Recovery Slots	16
Dispatch Width	4
ALU Pipelines	4
MEM Pipelines	2

**Table 4: Summary of Synthesis Results**

Metric	SonicBOOM	RingBOOM
Frequency (GHz)	0.86	2.1
Total Area ( $mm^2$ )	0.27	0.21

**Table 5: Synthesized Frequencies of BOOM Core Variants**

Variant	Frequency (GHz)
SonicBOOM	0.86
Initial RingBOOM	1.1
+Arbitration Stage	1.4
+Dispatch Stage	1.9
+Allocation Stage	2.1

## 6 DISCUSSION

### 6.1 IPC Tweaks

Many microarchitectural tweaks were needed for RingBOOM to achieve the IPC reported in section 5.1. These tweaks are described below.

**6.1.1 Load-hit Bypassing.** In early implementation, bypassing from load-hits was performed with ring-topology bypassing ports, just as with ALU bypassing. However, it was observed that multiple instructions are often waiting on a single load: if load-hit wake-ups and bypasses were fanned out to every column, the waiting instructions could be allocated into different columns and executed simultaneously after being woken up. Load dependency tracking is described in section 4.1 while bypassing is described in section 4.5.

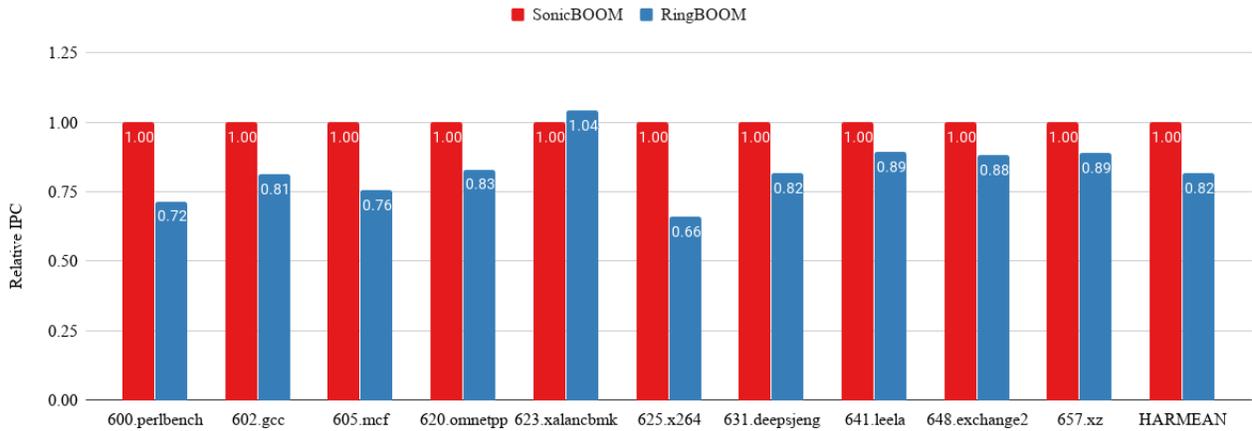


Figure 7: SPEC 2017 IPC Relative to SonicBOOM

6.1.2 *Memory Ordering Failures.* The division of a unified scheduler into columns tends to increase the prevalence of memory ordering failures. This is because ready loads and stores are not always executed in age order: nearby reads/writes of the same address which are woken up on the same cycle may accidentally proceed out-of-order. Using a rotating-priority to arbitrate access to the shared load/store unit was found to reduce the frequency ordering failures.

6.1.3 *Fast Exception Recovery.* To reduce the impact of memory ordering failures and simplify some of the below QoR improvements, the cycle-by-cycle "rollback" exception recovery mechanism was replaced with a commit snapshot. This allows the commit state of rename structures to be "flushed back" in a single cycle, similar to how fetch mispredictions are handled. This is especially helpful as the speculative depth of the core is increased: deep speculation implies more instructions to be rolled back in the case of a memory ordering failure.

6.1.4 *Flexible Banking.* To keep the arbiter fast, PRF banking initially involved a fixed two read ports per bank: the first of these ports could be allocated to requesting rs1 operands, while the second could be allocated to requesting rs2 operands. This inflexible allocation meant that two conflicting rs1 requests would result in one instruction being cancelled, even if no rs2 operands requested that bank. This was replaced with a more flexible mechanism which is parameterized to allow configurations with more than two read ports per bank. The read-sharing optimization described in [6] was also implemented, but increasing the number of read ports per bank to 3 was found to subsume much of the benefit.

6.1.5 *High-Priority Issue Requests.* A problem was observed in the execution of short loops: at least two subsequent instructions depend on loop index bumps or array pointer bumps. These dependencies are mapped to the same column in order to receive a wakeup from the index bump. Only one age-ordered ready instruction can be issued per cycle out of a column, preventing the next

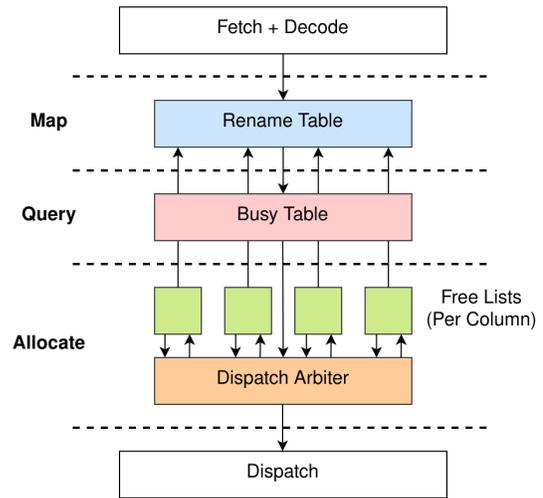


Figure 8: 3-stage Column-Steering Rename Pipeline

loop iteration from being executed on the following cycle. This was addressed with a high-priority request port for index bumps.

## 6.2 QoR Improvements

At the outset, the initial RingBOOM variant achieved a synthesized clockspeed of roughly 1.1 GHz. Following significant effort to resolve critical paths, a maximum clockspeed of 2.1 GHz was registered. The changes underlying this difference are summarized below.

6.2.1 *Allocation Stage.* To hit the highest mentioned clockspeed in section 5.2 a pipeline stage for physical register allocation was added, pictured in figure 8.

6.2.2 *Dispatch Stage.* A dispatch pipeline stage (DIS) was added, described in section 4.2.

**6.2.3 Arbitration Stage.** An arbitration pipeline stage (ARB) was added to prevent resource collisions without slowing down the wakeup-select loop, described in section 4.3.

**6.2.4 Fetch Mispredict Logic.** As instruction fetch mispredictions may be detected by any ALU (branches) or the JMP unit (jumps), the oldest mispredict of many needs to be selected to flush the pipeline and redirect instruction fetch to the correct address. This was previously done in a way which scaled poorly to wide pipelines: it has been replaced with efficient, low-depth one-hot logic.

**6.2.5 Issue Grant Fan-out.** Issue slot compaction logic previously depended on whether or not a potential issue slot resident had been selected for issue that cycle. This required an issue grant signal to fan out to a multiplexer select for every bit within an issue slot, slowing the wakeup-select loop. This dependence was not necessary and has been removed.

**6.2.6 Delayed Rename Structure Writes.** Writes into the map table and busy table were delayed by a cycle. These delayed writes are bypassed to reading instructions.

**6.2.7 2-Stage JMP Unit.** The JMP unit checks that jump-register instructions (such as subroutine returns) jumped to the right address. It was previously found to show up as a critical path, as the fetched bundle address needs to be correctly offset and corrected for RVC compression before comparison with the correct jump address. Thus, a second stage was added and retiming was used to automatically repipeline the unit.

**6.2.8 Divider Readiness.** The integer divider is an iterative unit, which can be busy for up to 64 cycles. Previously, a readiness bit was fanned out from the divider to each issue slot, preventing DIV instructions from requesting issue when the divider was busy: this fan-out unnecessarily slowed the wakeup-select loop. The mechanism was changed to cancel DIV instructions during the ARB stage when the divider is busy.

## 6.3 Scalability

Preliminary evaluation of a 10-wide RingBOOM core suggested that RingBOOM's IPC may scale well beyond a 4-wide pipeline. This will be limited by memory ordering failures and contention over the load/store unit (see section 7). Additionally, it is possible that the various column crossbars will prevent graceful scaling to a very wide pipeline. In this case, it may be sensible to implement the form of clustering used in the Alpha 21264 [4] atop the banked organization or to employ scalable routers in lieu of a crossbars as in Ultrascalar [16].

## 7 FUTURE WORK

The remainder of the BOOM core needs to be "brought up to speed" with the frequency improvements attained by the RingBOOM core. This will pose significant challenges: before doing this, it may make sense to determine an optimal logical depth in modern technologies, similar to the analysis performed in [17]: this could be used as a target for any further logical depth optimizations. Subsequently, work should focus on further scaling the pipeline width and the implementation of a scalable load/store unit. These two goals are intertwined: a scalable load/store unit will allow execution

of more load/store instructions per cycle. Load and store instruction throughput will limit performance of the wide execution cores enabled by the banked ring organization.

## 8 CONCLUSION

Many scalable approaches to out-of-order superscalar microarchitectures have been proposed and attempted over the past two decades. Many of these schemes rely on subdivision of a large core into small, nearly independent cores. Other schemes rely on highly complex control logic. Banked microarchitectures and RingScalar stand out as an efficient means to build a centralized execution core. They approximate the behavior of an ideal, fully-ported core by optimizing for common patterns found in instruction streams. This work has focused on the implementation of this microarchitecture as a fork of the open-source BOOM core. The IPC and physical results suggest that this microarchitectural style deserves further investigation and refinement.

## REFERENCES

- [1] Jessica Tseng and Krste Asanovic. Ringscalar: A complexity-effective out-of-order superscalar microarchitecture. Technical Report MIT-CSAIL-TR-2006-066, Massachusetts Institute of Technology, September 2006.
- [2] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, Jan 1967.
- [3] K. C. Yeager. The mips r10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–41, April 1996.
- [4] R. E. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March 1999.
- [5] E. Borch, S. Manne, J. Emer, and E. Tune. Loose loops sink chips. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, page 0299, Los Alamitos, CA, USA, feb 2002. IEEE Computer Society.
- [6] J. H. Tseng and K. Asanovic. Banked multiported register files for high-frequency superscalar microprocessors. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pages 62–71, June 2003.
- [7] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The multicenter architecture: reducing cycle time through partitioning. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 149–159, Dec 1997.
- [8] J. H. Tseng and K. Asanovic. A speculative control scheme for an energy-efficient banked register file. *IEEE Transactions on Computers*, 54(6):741–751, June 2005.
- [9] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimantas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.
- [10] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A. Patterson, and Krste Asanovic. Boom v2: an open-source out-of-order risc-v core. Technical Report UCB/EECS-2017-157, EECS Department, University of California, Berkeley, Sep 2017.
- [11] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. The risc-v instruction set manual, volume i: User-level isa, version 2.1. Technical Report UCB/EECS-2016-118, EECS Department, University of California, Berkeley, May 2016.
- [12] Jerry Zhao, Abraham Gonzales, Ben Korpan, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. Technical report, EECS Department, University of California, Berkeley, May 2020.
- [13] Embedded Microprocessor Benchmark Consortium. Coremark: An eembc benchmark, 2018.
- [14] Standard Performance Evaluation Corporation (SPEC). Spec cpu 2017, 2017.
- [15] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 29–42. IEEE Press, 2018.
- [16] D. S. Henry, B. C. Kuzmaul, and V. Viswanath. The ultrascalar processor-an asymptotically scalable superscalar microarchitecture. In *Proceedings 20th Anniversary Conference on Advanced Research in VLSI*, pages 256–273, 1999.
- [17] Victor Zyuban, David Brooks, Viji Srinivasan, Michael Gschwind, Pradip Bose, Philip Strenski, and Philip Emma. Integrated analysis of power and performance for pipelined microprocessors. *Computers, IEEE Transactions on*, 53:1004 – 1016, 09 2004.