*Sonic*BOOM: The 3rd Generation Berkeley Out-of-Order Machine

Jerry Zhao
jzh@berkeley.edu
University of California, Berkeley

Ben Korpan
bkorpan@berkeley.edu
University of California, Berkeley

Abraham Gonzalez
abe.gonzalez@berkeley.edu
University of California, Berkeley

Krste Asanovic
krste@berkeley.edu
University of California, Berkeley

ABSTRACT

We present *Sonic*BOOM, the third generation of the Berkeley Out-of-Order Machine (BOOM). *Sonic*BOOM is an open-source RTL implementation of a RISC-V superscalar out-of-order core and is the fastest open-source core by IPC available at time of publication.

*Sonic*BOOM provides a state-of-the-art platform for research in high-performance core design by providing substantial microarchitectural improvements over BOOM version 2. The most significant performance gains are enabled by optimizations to BOOM's execution path and a redesign of the instruction fetch unit with a new hardware implementation of the state-of-the-art TAGE branch predictor algorithm. Additionally, *Sonic*BOOM provides the first open implementation of a load-store unit that can provide multiple loads per cycle. With these optimizations and new features, *Sonic*BOOM can achieve 2x higher IPC on SPEC CPU benchmarks compared to any prior open-source out-of-order core. Additionally, *Sonic*BOOM achieves 6.2 CoreMark/MHz, making it the fastest currently available open-source core by IPC.

CCS CONCEPTS

• **Computer systems organization → Superscalar architectures**.

KEYWORDS

superscalar, out-of-order, microarchitecture, branch-prediction, open-source, RISC-V

1 INTRODUCTION

As the need for general-purpose computing power increases, the deployment of high-performance superscalar, out-of-order cores has expanded beyond datacenters and workstations, into mobile and edge devices. In addition, the recent disclosure of microarchitectural timing attacks [12, 14] on speculating cores injects a new concern into the design space. Architects must now consider security, in addition to power, performance, and area, when evaluating new designs.

For those studying these problems, an open-source hardware implementation of a superscalar out-of-order core is an invaluable resource. Compared to open-source software models of high-performance cores, like gem5 [5], MARSSx86 [18], Sniper [9], or ZSim [22], an open-source hardware implementation provides numerous advantages. Unlike a software model, a hardware implementation can demonstrate precise microarchitectural behaviors,

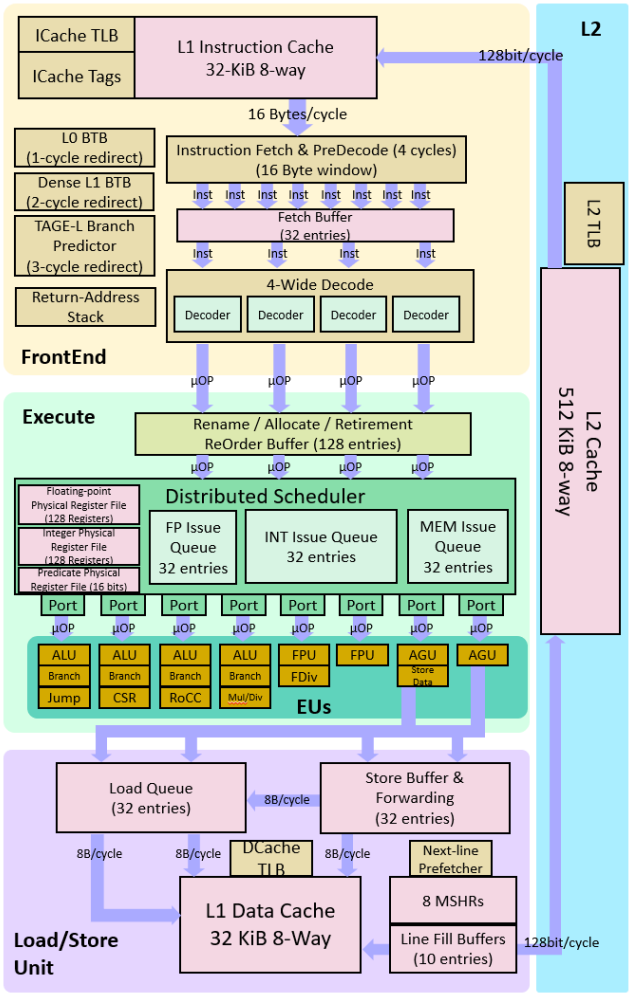

Figure 1: 4-wide fetch, 8-wide issue configuration of *Sonic*BOOM. This configuration is performance-competitive with an AWS Graviton core.

execute real applications for trillions of cycles, and empirically provide power and area measurements. Furthermore, an open hardware implementation provides a baseline platform as a point of comparison for new microarchitectural optimizations.

While the growth in the number of open-source hardware development frameworks in recent years may seem to provide the

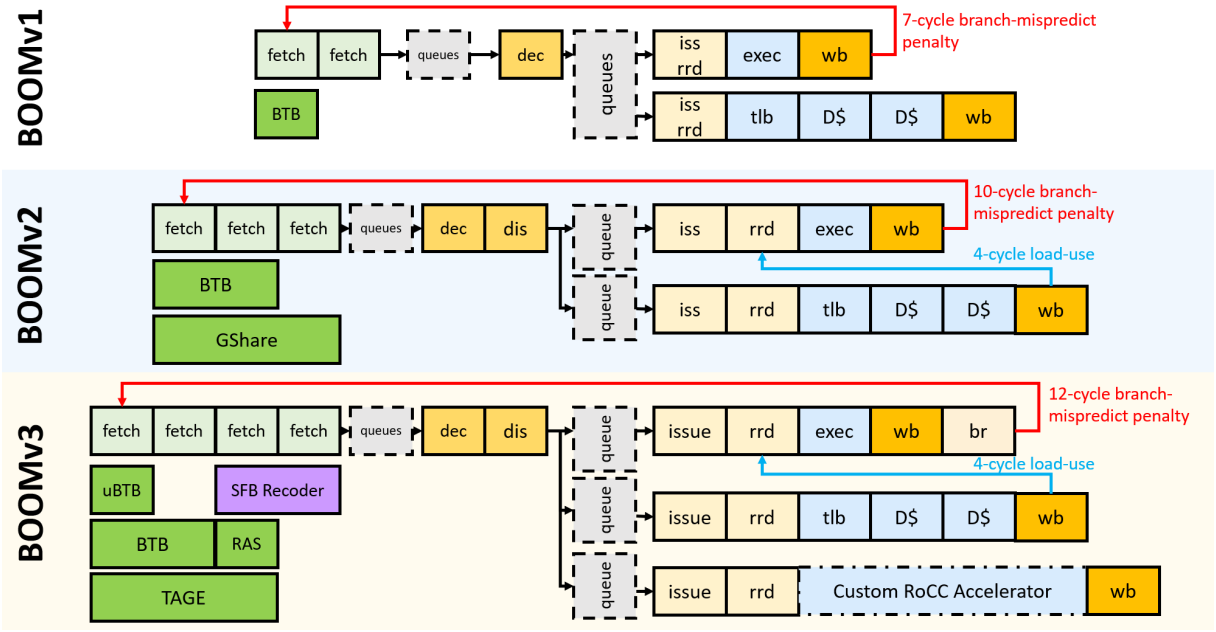

Figure 2: BOOM pipeline across BOOMv1, BOOMv2, and BOOMv3 (*Sonic*BOOM)

solution to this problem [2, 4], most of these frameworks only provide support for simple in-order cores, like Rocket [3], Ariane [31], Black Parrot [21], or PicoRV32 [29]. Without a full-featured, high-performance implementation of a superscalar out-of-order core, these frameworks cannot generate modern mobile or server-class SoCs.

Although there has also been an explosion in the number of open-source out-of-order cores in recent years, these fail to address the demand for an open high-performance implementation. Neither BOOMv2 [11], riscy-OOO [32], nor RSD [17] demonstrate substantial performance advantages over open-source in-order cores. The lack of features like 64-bit support, compressed-instruction support, accurate branch prediction, or superscalar memory accesses, in some of these designs further inhibits their usefulness to architects interested in studying application-class cores.

To address these concerns, we developed the third generation of the BOOM core, named ***Sonic*BOOM**. Table 1 displays how *Sonic*BOOM demonstrates improved performance and extended functionality compared to prior open-source out-of-order cores. Furthermore, the configuration of *Sonic*BOOM depicted in Figure 1 is the **fastest publicly available open-source core** (measured by IPC at time of publication).

Table 1: Comparison of open-source out-of-order cores.

	***Sonic* BOOM**	**BOOMv2**	**riscy-OOO**	**RSD**
ISA	RV64GC	RV64G	RV64G	RV32IM
DMIPS /MHz	3.93	1.91	?	2.04
SPEC06 IPC	0.86	0.42	0.48	N/A
Branch Predictor	TAGE	GShare	Tourney	GShare
Dec Width	1-5	1-4	2	2
Mem Width	16b/cycle	8b/cycle	8b/cycle	4b/cycle
HDL	Chisel3	Chisel2	BSV+CMD	System Verilog

2 BOOM HISTORY

We describe prior iterations of the BOOM core, and how the development history motivated a new version of BOOM. Figure 2 depicts the evolution of the BOOM microarchitecture towards *Sonic*BOOM.

2.1 BOOM Version 1

BOOM version 1 (BOOMv1) was originally developed as an educational tool for UC Berkeley's undergraduate and graduate computer architecture courses. BOOMv1 closely followed the design of the MIPS R10K [30], and featured a short, simple pipeline, with a unified register file and issue queue. BOOMv1 was written in the Chisel hardware description language, and heavily borrowed existing components of the Rocket in-order core in its design, including the frontend, execution units, MMU, L1 caches, and parts of the branch predictor. While BOOMv1 could achieve commercially-competitive performance in simulation, its design had unrealistically few pipeline stages, and was not physically realisable.

2.2 BOOM Version 2

BOOM version 2 (BOOMv2) improved the design of BOOMv1 to be more suitable for fabrication and physical design flows. In order to be physically realizable, BOOMv2 added several pipeline stages to the frontend and execution paths, resolving critical paths in BOOMv1. In addition, the floating point register file and execution units were partitioned into an independent pipeline, and the issue queues were split into separate queues for integer, memory, and floating-point operations. BOOMv2 was fabricated within the BROOM test chip in TSMC 28nm [6].

2.3 BOOM Version 3

BOOM version 3 (*Sonic*BOOM) builds upon the performance and physical design lessons from previous BOOM versions in order to support a broader set of software stacks and address the main performance bottlenecks of the core while maintaining physical realizability. We identify key performance bottlenecks within the instruction fetch unit, execution backend, and load/store unit. We also provide new implementations of many structures which, in BOOMv2, were borrowed from the Rocket in-order core. New implementations in *Sonic*BOOM were designed from the ground-up for integration within a superscalar out-of-order core.

3 INSTRUCTION FETCH

BOOMv2's instruction fetch unit was limited by the lack of support for compressed 2-byte RISC-V (RVC) instructions, as well as a restrictivly tight coupling between the fetch unit and the branch predictors. *Sonic*BOOM addresses both of these issues, with a new frontend capable of decoding RVC instructions, and a new advanced pipelined TAGE-based branch predictor.

3.1 Compressed Instructions

The C-extension to the RISC-V ISA [28] provides additional support for compressed 2-byte forms of common 4-byte instructions. Since this extension substantially reduces code-size, it has become the default RISC-V ISA subset for packaged Linux distributions like Fedora and Debian. These distributions additionally include thousands of pre-compiled software packages, providing a rich library of applications to run without requiring complex cross-compilation flows.

To support the growing community-driven ecosystem of pre-packaged RISC-V software, *Sonic*BOOM includes a new superscalar fetch unit which supports the C-extension. *Sonic*BOOM's new fetch unit decodes a possible 2-byte or 4-byte instruction for every 2-byte parcel. An additional pipeline stage after the frontend-decoders shifts the decoded instructions into a dense fetch-packet to pass into the *Sonic*BOOM backend.

3.2 Branch Prediction

Branch prediction is a critical component contributing to the performance of out-of-order cores. Hence, improving branch prediction accuracy in *Sonic*BOOM was a first-order concern. The tight integration between the fetch unit, branch target buffer (BTB), and branch predictor within BOOMv2 restricted the addition of new features and optimizations within the fetch pipeline. Bug fixes and new features to the fetch unit frequently degraded branch predictor accuracy between BOOMv1 and BOOMv2.

*Sonic*BOOM's fetch pipeline was redesigned with a more general and flexible interface to a pipelined hierarchical branch predictor module. Compared to BOOMv2, *Sonic*BOOM supports a single-cycle next-line predictor, and also provides substantial machinery for managing branch predictor state, local histories, and global histories.

The branch predictor was re-written to integrate cleanly with the superscalar banked fetch unit in BOOM. In BOOMv2, the banked ICache was partnered to an unbanked branch predictor, resulting in frequent aliasing of branches between the even/odd ICache banks in the predictor memories. The final result was that the branch predictor capacity was effectively halved in BOOMv2, as the misconfiguration forced the predictor to learn two entries (one for each bank) for some branches. In *Sonic*BOOM, the branch predictor is banked to match the ICache.

Additionally, *Sonic*BOOM rectifies the minimum 2-cycle redirect penalty in BOOMv2 by adding a small micro BTB (uBTB). This uBTB (sometimes called "next-line-predictor", or "L0 BTB") redirects the PC in a single cycle from a small fully-associative buffer of branch targets, drastically improving fetch throughput on small loops.

The most significant contribution to overall core performance is the inclusion of a high-performance TAGE [24] branch predictor, with a speculatively updated and repaired global-history vector driving predictions. Unlike BOOMv2, *Sonic*BOOM carefully maintains a superscalar global history vector across speculative fetches and misspeculation, enabling more accurate predictions. The *Sonic*BOOM predictors were also redesigned to be superscalar, as we observed aliasing between branches in the same fetch-packet significantly degraded prediction accuracy for those branches in BOOMv2.

*Sonic*BOOM additionally provides new repair mechanisms to restore predictor state after misspeculation. Specifically, the loop predictor and return-address-stack (RAS) are snapshotted and repaired on mispredict [26]. Compared with the original unrepaired RAS from BOOMv2, the *Sonic*BOOM RAS achieves 10x fewer mispredicts, with 98% prediction accuracy on `ret` instructions.

*Sonic*BOOM also changed the branch resolution mechanism. In BOOMv2, only a single branch per cycle could be resolved, as branch units must read the single-ported fetch-PC queue to determine a branch target. We observed that this limitation limited scalability towards wider frontends with high-throughput trace caches, as branch-dense code might include multiple branches in a fetch packet. In *Sonic*BOOM, we add support for superscalar branch resolution, with multiple branch-resolution units. An additional pipeline stage is inserted after writeback to only read the fetch-PC queue to determine the target address for the *oldest mispredicted branch* in a vector of resolved branches. While this increases our branch-to-branch latency to 12 cycles, the additional scheduling flexibility provided by multiple branch units overall improved performance on relevant workloads, as the schedulers could more aggressively schedule branches, instead of waiting for a single branch unit to become available.

4 EXECUTE

We provide two major new features in *Sonic*BOOM's execute pipeline. Support for the RoCC accelerator interface enables integration of custom accelerators into the BOOM pipeline. The short-forwards branch (SFB) optimization improves IPC by recoding difficult-to-predict branches into internal predicated microOps.

4.1 RoCC Instructions

The Rocket Custom Coprocessor interface (RoCC) was originally designed as a tightly-integrated accelerator interface for the Rocket in-order core. When implemented, the Rocket in-order core will send the operands and opcodes of any custom accelerator instruction through the RoCC interface to the accelerator, which can write

data directly into core registers. Additionally, the accelerator can also access ports to the L1, L2, outer memory, and MMU.

The design of the RoCC interface has proven to be very useful for accelerator research. A wide variety of tightly-integrated accelerators have been designed for this interface, accelerating a diverse set of tasks including machine learning [7], vector computation [13], garbage collection [15], page fault handling [20], memory copying [16], and cryptography [23].

Nevertheless, some of the workloads accelerated by these accelerators have been limited by the performance of the host Rocket core, as the host core must be able to fetch and issue enough instructions to saturate the accelerator. We implemented support for the RoCC interface in *Sonic*BOOM to provide a platform for accelerator research on top of high performance out-of-order cores. Unlike Rocket, *Sonic*BOOM speculatively decodes RoCC instructions and holds their operands in a "RoCC queue" until they pass the architectural commit point, at which time they can be issued into the accelerator. As a result, *Sonic*BOOM can more aggressively drive a custom RoCC accelerator, compared to the Rocket core.

4.2 Short-forwards Branch Optimizations

A frequent code pattern is a data-dependent branch over a short basic block. The data-dependent branches in these sequences are often challenging to predict, and naive execution of these code sequences would result in frequent branch mispredictions and pipeline flushes.

While the base RISC-V ISA does not provide conditional-move or predicated instructions, which can effectively replace unpredictable short-forwards branches, we observe that we can dynamically optimize for these cases in the microarchitecture. Specifically, we introduce additional logic to detect short-forwards in fetched instructions, and decode them into internal "set-flag" and "conditional-execute" micro-ops. This is similar to the predication support in the IBM Power8 microarchitecture [25].

The "set-flag" micro-op replaces the original "branch" micro-op, and instead writes the outcome of the branch to a renamed predicate register file. Renaming the predicate register file is necessary to support multiple in-flight short-forwards-branch sequences. The "conditional-execute" micro-ops read the predicate register file to determine whether to execute their original operation, or to perform a copy operation from the stale physical register to the destination physical register. In the example in Figure 3, the short basic block consisting of two `mv` instructions are internally recoded as "conditional-moves" within *Sonic*BOOM, while the unpredictable bge branch can be recoded as a "set-flag" operation.

C	Assembly	Executed MicroOps

`int max = 0;`
`int maxid = -1;`
`for (i = 0; i < n; i++) {`
`  if (x[i] >= max) {`
`  max = x[i];`
`  maxid = i;`
`  }`
`}`
	`loop:`
	`  lw  x2, 0(a0)`
	`  bge  x1, x2, skip`
	`  mv  x1, x2`
	`  mv  a1, t0`
	`skip:`
	`  addi a0, a0, 4`
	`  addi t0, t0, 0x1`
	`  j  loop:`
		`loop:`
		`  lw  x2, 0(a0)`
		`  set.ge  x1, x2`
		`  p.mv  x1, x2`
		`  p.mv  a1, t0`
		`  addi  a0, a0, 4`
		`  addi  t0, t0, 0x1`
		`  j  loop:`

Figure 3: Short forwards branch appearing in a loop to find the max of an array. The C source, assembly, and decoded μOPs are shown.

We observe that this optimization provides up to 1.7x IPC on some code sequences. As an example, *Sonic*BOOM achieves 6.15 CoreMark/MHz with the SFB optimization enabled, compared to 4.9 CoreMark/MHz without.

5 LOAD-STORE UNIT AND DATA CACHE

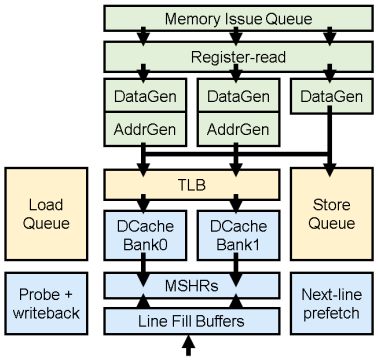

Figure 4: *Sonic*BOOM load-store-unit, rewritten to optimize for superscalar, speculative, and out-of-order memory operations.

In order to maximize RTL re-use, BOOMv1 and BOOMv2 used the L1 data-cache implementation of the Rocket in-order core. However, we observed that this reliance on a L1 data-cache designed for an in-order core incurred substantial performance penalties.

The interface to Rocket's L1 data-cache is only 1-wide, limiting throughput. On a wide superscalar core, this limitation is a significant performance bottleneck, blocking the wide fetch and decode pipeline on a narrow load-store pipeline.

Furthermore, Rocket's L1 data cache cannot perform any operations speculatively, as any cache refill would irrevocably result in a cache eviction and replacement. This results in significant cache pollution from misspeculated accesses, when used in the BOOM core.

Finally, Rocket's L1 data cache blocks load refills on cache eviction. Although the access latency to the L2 is only 14 cycles, the access time as measured from the core was 24 cycles, due to the additional cycles spent evicting the replaced line.

To address these problems, *Sonic*BOOM includes a new load-store unit and L1 data cache, depicted in Figure 4.

5.1 Dual-ported L1 Data Cache

To support dual issue into *Sonic*BOOM's memory unit, the new design stripes the data cache across two banks. The new L1 data cache supports dual accesses into separate banks, as each bank is still implemented as 1R1W SRAMs.

While an alternative implementation using 2R1W SRAMs can achieve similar results, we observe that even-odd banking is sufficient in our core to relieve the data-cache bottleneck and enables physical implementation with simple SRAM memories. Common load-heavy code sequences, such as restoring registers from the stack, or regular array accesses, generate loads that evenly access both even and odd pointers.

The remaining challenge was to redesign the load-store unit to be dual-issue, matching the L1 data cache. The load-address and

store-address CAMs were duplicated, and the TLB was dual-ported. The final *Sonic*BOOM load store unit can issue two loads or one store per cycle, for a total L1 bandwidth of 16 bytes/cycle read, or 8 bytes/cycle write.

5.2 Improving L1 Performance

In *Sonic*BOOM, a load miss in the L1 data-cache immediately launches a refill request to the L2. The refill data is written into a line-fill buffer, instead of directly into the data-cache. Thus, cache evictions can occur in parallel with cache refills, drastically reducing observed L2 hit times. When cache eviction is completed, the line-fill buffer is flushed into the cache arrays.

The final implementation of *Sonic*BOOM's non-blocking L1 data cache contains multiple independent state machines. Separate state machines manage cache refills, permission upgrades, writebacks, prefetches, and probes. These state machines operate in parallel, and only synchronize when necessary to maintain memory consistency.

We also introduce a small next-line prefetcher between the L1 and the L2. The next-line prefetcher speculatively fetches sequential cache lines after a cache miss into the line fill buffers. Subsequent hits on addresses within the line fill buffers will cause the cache to write the prefetched lines into the data arrays.

*Sonic*BOOM's line-fill buffers can also be modified to provide a small amount of resistance to Spectre-like attacks, which leak information through misspeculated cache refills [8]. The line-fill buffers can be modified to write lines into the L1 cache only if the request for that line was determined to be correctly speculated. Misspeculated cache refills can simply be flushed out of the L1, preventing attacker processes from learning speculated behaviors from L1 data-cache state.

6 SYSTEM SUPPORT

6.1 SoC Integration

*Sonic*BOOM plugs-in within the tile interface of the Rocket Chip SoC generator ecosystem and the Chipyard integrated SoC research and development framework. As such, it integrates with a broad set of open-source heterogeneous SoC components and devices, including UARTs, GPIOs, JTAGs, shared-cache memory systems, and various accelerators. These components include the open-source SiFive inclusive L2 cache, the Hwacha vector accelerator [13], and the Gemmini neural-network accelerator [7]. Within the Chipyard framework, *Sonic*BOOM can be integrated with additional RISC-V

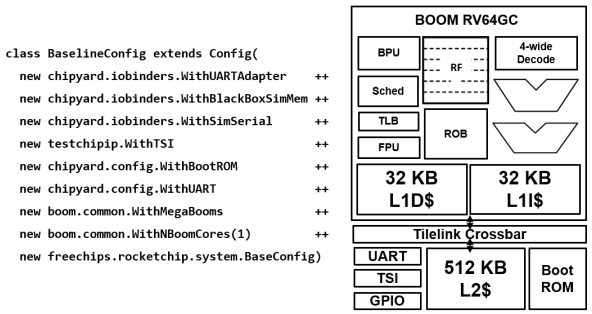

Figure 5: Chipyard couples *Sonic*BOOM with a vast ecosystem of SoC components for simulation and synthesis.

cores such as Rocket [3] or Ariane [31] to generate heterogeneous SoCs similar to modern hybrid processor architectures (ARM® big.LITTLE®, Intel® hybrid architectures). Figure 5 depicts how Chipyard generates a complete BOOM-based SoC from a high-level specification.

The Chipyard framework provides *Sonic*BOOM an integrated emulation and VLSI implementation environment which enables continuous performance and efficiency improvements through short iterations of full-system performance evaluation using FPGA-accelerated simulation on FireSim [10], as well as consistent physical design feedback through the Hammer [27] VLSI flow.

6.2 Operating System Support

*Sonic*BOOM has been tested to support RV64GC Buildroot and Fedora Linux distributions. As the highest-performance open-source implementation of a RISC-V processor, *Sonic*BOOM enabled the identification of critical data-race bugs in the RISC-V Linux kernel.

The RISC-V kernel page-table initialization code requires careful insertion of `FENCE` instructions to synchronize the TLB as the kernel constructs the page-table entries. As high-performance cores might speculatively issue a page table walk before a newly constructed page-table-entry has been written to the cache, these `FENCE` instructions are necessary for maintaining program correctness. *Sonic*BOOM's aggressive speculation found a section of the kernel initialization code where a missing `FENCE` caused a stale page-table-entry to enter the TLB, resulting in an unrecoverable kernel page fault. We are working on upstreaming the fix for this issue into the Linux kernel.

6.3 Validation

Debugging an out-of-order core is immensely challenging and time-consuming, as many bugs manifest only in extremely specific corner cases after trillions of cycles of simulation. We discuss two methodologies we used to productively debug *Sonic*BOOM's new features.

6.3.1 Unit-testing. From our experience, the load-store unit and data-cache are the most bug-prone components of an out-of-order core, as they must carefully maintain a memory-consistency model, while hiding the latency of loads, stores, refills, and writebacks as fast as possible. We integrated *Sonic*BOOM's load/store unit and L1 data-cache with the TraceGen tool in the Rocket Chip generator, which stress-tests the load-store unit with random streams of loads and stores, and validates memory consistency. We additionally developed a new tool `memtrace`, which analyzes the committed sequence of loads and stores in a single-core device, and checks for sequential consistency. These tools helped resolve several data-cache and load/store-unit bugs that manifested only after trillion of cycles of simulation.

6.3.2 Fromajo Co-simulation. *Sonic*BOOM is also integrated with the Dromajo [1] co-simulation tool. Dromajo checks that the committed instruction trace matches the trace generated by a software architectural simulator. Fromajo integrates Dromajo with a FireSim FPGA-accelerated *Sonic*BOOM simulation, enabling co-simulation at over 1 MHz, orders of magnitude faster than a software-only co-simulation system. Fromajo revealed several latent bugs related to interrupt handling and CSR management.

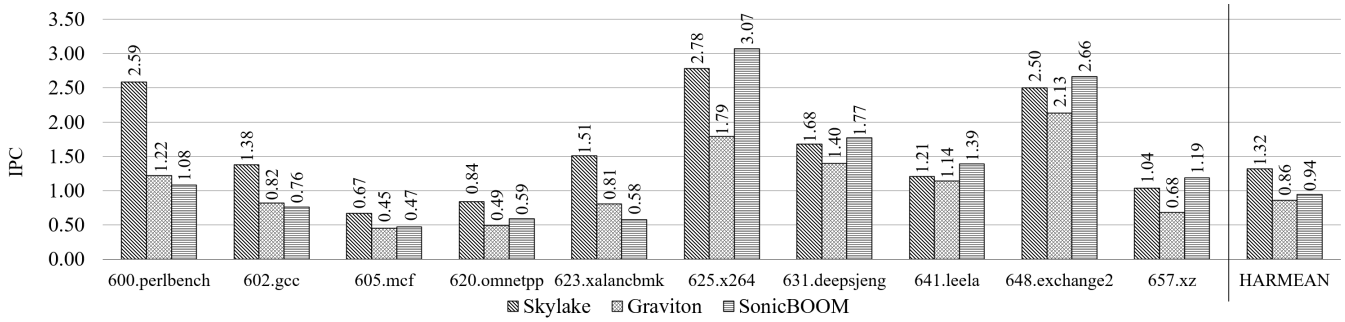

Figure 6: *Sonic*BOOM SPEC17 IPC compared to Intel Skylake and AWS Graviton cores.

7 EVALUATION

*Sonic*BOOM was physically synthesized at 1 GHz on a commercial FinFET process, matching the frequency achieved by BOOMv2. We evaluate *Sonic*BOOM on the CoreMark, SPECint 2006 CPU, and SPECint 2017 CPU benchmarks. For all performance evaluations, *Sonic*BOOM was simulated on FireSim [10] using AWS F1 FPGAs. The FireSim simulations ran at 30 MHz on the FPGAs, and modeled the system running at 3.2 GHz. A single-core system was simulated with 32 KB L1I, 32 KB L1D, 512 KB L2, 4 MB simulated L3, and 16 GB DRAM.

7.1 SPECintspeed

We compare both SPEC06 and SPEC17 intspeed IPC to existing cores for which data is available. All SPEC benchmarks were compiled with `gcc -O3 -falign-X=16`, to enable most default compiler optimizations, and to align instructions into the *Sonic*BOOM ICache.

We compare SPEC17 intspeed IPC against IPC achieved by AWS Graviton and Intel Skylake cores. The Graviton is a 3-wide A72-like ARM-based core, while the Skylake is a 6-wide x86 core. SPEC17 benchmarks were compiled with `gcc`, with `-O3` optimizations.

The results in Figure 6 show that *Sonic*BOOM is competitive with the Graviton core, and can even match the IPC of the Skylake core on some benchmarks. However, we note that the difference in ISAs between these three systems skews the IPC results.

7.2 CoreMark

We compare *Sonic*BOOM performance on CoreMark to prior open-source and closed-source cores. While CoreMark is not a good evaluation of out-of-order core performance [19], published results are available for many existing cores. The results in Figure 7 show that *Sonic*BOOM achieves superior CoreMark/MHz compared to any prior open-source core.

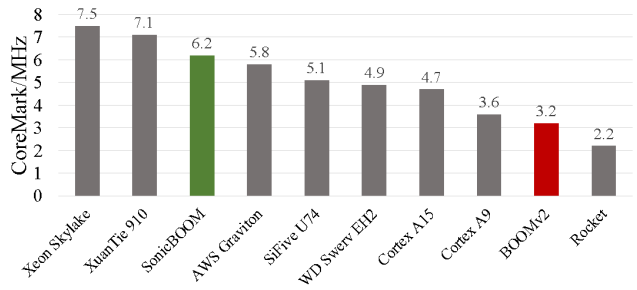

Figure 7: *Sonic*BOOM CoreMark/MHz compared to similar cores. *Sonic*BOOM provides almost a 2x IPC improvement over BOOMv2.

8 WHAT'S NEXT

8.1 Vector Execution

Vector (or SIMD) execution remains as an optimization path, as most high-performance architectures provide some set of vector instructions for accelerating trivially data-parallel code. The maturing RISC-V Vector extensions provides a ISA specification for the implementation of an out-of-order vector-execution engine within the BOOM core. We hope to provide an out-of-order implementation of the Vector ISA in a future version of BOOM.

8.2 Instruction and Data Prefetchers

Both L1 instruction and data cache-misses incur significant performance penalties in an out-of-order core. A L1 ICache miss on the fetch-path inserts at minimum a 10-cycle bubble into the pipeline, equivalent to a branch misprediction. Out-of-order execution may attempt to hide the penalty of a data-cache miss, but the speculation depth in *Sonic*BOOM cannot hide misses beyond L1. As L3 hit-latency is on the order of 50 cycles, BOOM needs to speculate 50-cycles ahead to fully hide the penalty of a L1 miss. On a 4-wide BOOM, 50-cycles ahead is 200 instructions, exhausting the capacity of the reorder buffer.

Thus, both instruction and data prefetchers are vital for maintaining instruction throughput in an out-of-order core. While *Sonic*BOOM provides a small prefetcher to fetch the next cache line after a miss into the L1, a more robust outer-memory prefetcher that can completely hide access time to L3 or DRAM is desired.

9 CONCLUSION

*Sonic*BOOM represents the next step towards high performance open-source cores. Numerous performance bottlenecks introduced by the physical design improvements for BOOMv2 were resolved, and many new microarchitectural optimizations were implemented. The resulting application-class core is performance-competitive with commercially high-performance cores deployed in datacenters. We hope that *Sonic*BOOM will prove to be a valuable open-source asset for computer architecture research.

ACKNOWLEDGMENTS

The information, data, or work presented herein was funded by ADEPT Lab industrial sponsors and affiliates. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

REFERENCES

[1] 2019. *Dromajo*. https://github.com/chipsalliance/dromajo

[2] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* (2020), Accepted.

[3] Krste Asanovic, David A Patterson, and Christopher Celio. 2015. *The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor*. Technical Report. University of California at Berkeley Berkeley United States.

[4] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, et al. 2016. OpenPiton: An open source manycore research framework. *ACM SIGPLAN Notices* 51, 4 (2016), 217–232.

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[6] Christopher Celio, Pi-Feng Chiu, Krste Asanović, Borivoje Nikolić, and David Patterson. 2019. Broom: an open-source out-of-order processor with resilient low-voltage operation in 28-nm cmos. *IEEE Micro* 39, 2 (2019), 52–60.

[7] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert Ou, Max Banister, et al. 2019. Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures. *arXiv preprint arXiv:1911.09925* (2019).

[8] Abraham Gonzalez, Ben Korpan, Jerry Zhao, Ed Younis, and Krste Asanović. 2019. Replicating and Mitigating Spectre Attacks on an Open Source RISC-V Microarchitecture. In *Third Workshop on Computer Architecture Research with RISC-V (CARRV)*.

[9] Wim Heirman, Trevor Carlson, and Lieven Eeckhout. 2012. Sniper: Scalable and accurate parallel multi-core simulation. In *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES-2012)*. High-Performance and Embedded Architecture and Compilation Network of . . . , 91–94.

[10] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. 2018. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 29–42.

[11] Donggyu Kim, Christopher Celio, David Biancolin, Jonathan Bachrach, and Krste Asanovic. 2017. Evaluation of RISC-V RTL with FPGA-accelerated simulation. In *First Workshop on Computer Architecture Research with RISC-V*.

[12] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.

[13] Yunsup Lee, Colin Schmidt, Albert Ou, Andrew Waterman, and K Asanovic. 2015. The Hwacha vector-fetch architecture manual, version 3.8. 1. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-262* (2015).

[14] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

[15] Martin Maas, Krste Asanović, and John Kubiatowicz. 2018. A hardware accelerator for tracing garbage collection. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 138–151.

[16] Howard Mao, Randy H Katz, and Krste Asanović. 2017. Hardware Acceleration for Memory to Memory Copies. (2017).

[17] Susumu Mashimo, Akifumi Fujita, Reoma Matsuo, Seiya Akaki, Akifumi Fukuda, Toru Koizumi, Junichiro Kadomoto, Hidetsugu Irie, Masahiro Goshima, Koji Inoue, et al. 2019. An Open Source FPGA-Optimized Out-of-Order RISC-V Soft Processor. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 63–71.

[18] Avadh Patel, Furat Afram, and Kanad Ghose. 2011. Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors. In *1st International Qemu Users' Forum*. 29–30.

[19] David Patterson. 2019. Embench™: Recruiting for the Long Overdue and Deserved Demise of Dhrystone as a Benchmark for Embedded Computing. https://www.sigarch.org/embench-recruiting-for-the-long-overdue-and-deserved-demise-of-dhrystone-as-a-benchmark-for-embedded-computing/

[20] Nathan Pemberton, John D Kubiatowicz, and Randy H Katz. 2018. *Enabling Efficient and Transparent Remote Memory Access in Disaggregated Datacenters*. Technical Report.

[21] Dan Petrisko. 2020. BlackParrot: An Agile Open Source RISC-V Multicore for Accelerator SoCs. *FOSDEM 2020* (2020).

[22] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer architecture news* 41, 3 (2013), 475–486.

[23] Colin Schmidt and Adam Izraelevitz. 2015. A Fast Parameterized SHA3 Accelerator. (2015).

[24] André Seznec. 2011. A new case for the TAGE branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 117–127.

[25] Balaram Sinharoy, Ron Kalla, William J Starke, Hung Q Le, Robert Cargnoni, James A Van Norstrand, Bruce J Ronchetti, Jeffrey Stuecheli, Jens Leenstra, Guy L Guthrie, et al. 2011. IBM POWER7 multicore server processor. *IBM Journal of Research and Development* 55, 3 (2011), 1–1.

[26] Kevin Skadron, Pritpal S Ahuja, Margaret Martonosi, and Douglas W Clark. 1998. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 259–271.

[27] Edward Wang, Adam Izraelevitz, Colin Schmidt, Borivoje Nikolic, Elad Alon, and Jonathan Bachrach. [n.d.]. Hammer: Enabling Reusable Physical Design. ([n. d.]).

[28] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. 2011. The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62* 116 (2011).

[29] C Wolf. 2019. Picorv32-a size-optimized risc-v cpu.

[30] Kenneth C Yeager. 1996. The MIPS R10000 superscalar microprocessor. *IEEE micro* 16, 2 (1996), 28–41.

[31] F. Zaruba and L. Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 11 (Nov 2019), 2629–2640. https://doi.org/10.1109/TVLSI.2019.2926114

[32] Sizhuo Zhang, Andrew Wright, Thomas Bourgeat, and Arvind Arvind. 2018. Composable building blocks to open up processor design. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 68–81.