Skip It: Take Control of Your Cache!

Shashank Anand
ETH Zurich
Zürich, Switzerland
sanand@student.ethz.ch

Michal Friedman
ETH Zurich
Zürich, Switzerland
michal.friedman@inf.ethz.ch

Michael Giardino*
Huawei Technologies
Zürich, Switzerland
michael.giardino@huawei.com

Gustavo Alonso
ETH Zurich
Zürich, Switzerland
alonso@inf.ethz.ch

Abstract

Mechanisms to explicitly manage the presence of data in caches are fundamental for the correctness and performance of modern systems. These operations, while critical, often incur significant performance penalties even when carefully used. Moreover, these mechanisms are implemented in proprietary and often undocumented hardware, so research into optimizations and novel designs is mostly limited to slow, simplified software simulations. In this paper, we design microarchitectural extensions to support two types of user-controlled cache writebacks to main memory. Furthermore, we propose Skip It, a mechanism built on top of our extensions that substantially reduces redundant writebacks. We implemented these designs on the open-source BOOM out-of-order RISC-V CPU. The performance in hardware is ≈100 cycles which favorably compares to similar operations in commercially available server-class platforms. In addition, Skip It performs as well as or better than state-of-the-art software techniques for avoiding unnecessary writebacks.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; **Superscalar architectures**; • **Hardware** → **Non-volatile memory**.

Keywords: microarchitecture, cache coherence, out-of-order, multicore, cacheline flush, fence, non-volatile memory, FPGA simulation, RISC-V

ACM Reference Format:
Shashank Anand, Michal Friedman, Michael Giardino, and Gustavo Alonso. 2024. Skip It: Take Control of Your Cache!. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620665.3640407

*Work was done while at ETH Zürich

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0385-0/24/04…$15.00
https://doi.org/10.1145/3620665.3640407

1 Introduction

While transparent management of CPU caches is often the most desirable behavior of a caching system, there are several scenarios in which a user or system would prefer fine-grained control of the presence of cached data. Non-volatile main memory (NVMM) is one case in which this control is critical. Persistent byte-addressable memory goes a long way to enable crash consistency [8]. However, due to the lack of fine-grained control of the cache contents, correct persistent algorithms are extremely challenging to implement and costly in terms of performance [17]. In the case of DMA engines, modifications to a locally cached copy must reach memory before subsequent accesses. In both cases, ensuring correctness in the presence of caches requires explicitly writing back data to ensure consistency. With the introduction of coherent interconnects such as CXL [62] that can connect multiple heterogeneous memory domains [49, 42], deliberate control of cached data becomes even more important. Explicit cache control can also assist the hibernation of extremely low power devices [80, 30] and help to mitigate some microarchitectural timing-channel attacks [14] by partitioning on-core resources [44, 75, 12].

Most modern computer architectures have methods for controlling the eviction and writeback of cache data. These instructions, however, do not come without cost. To ensure that writebacks have completed, programmers must integrate costly memory barriers (fences). When multiple threads are involved, the number of writebacks and fences increases, severely degrading performance. Moreover, even when data has already been written back by one thread, other threads are oblivious to the cache line state, and will redundantly writeback and fence the same data. Several researchers have introduced software methods to mitigate these unnecessary overheads by tracking whether a cache line has already been written back [73, 23, 71]. While these techniques are effective, hardware support can hide complexity from the programmer and reduce software overhead. However, due to the lack of open-source hardware, it has not been possible to realistically evaluate such proposals. The advent of performant, out-of-order RISC-V cores [79, 76] allows for the implementation and evaluation of exactly these types of novel hardware designs. Unfortunately, while the RISC-V instruction set architecture (ISA) defines writeback instructions, they have not been implemented in these cores.

This paper presents a complete hardware implementation of two variants of writeback instructions for the RISC-V

BOOM Core: clean and flush. Furthermore, we augment the existing fence implementation to maintain correct writeback ordering. In describing the implemented mechanisms, we provide a detailed examination and methodological description of BOOM core internal structures and related components. We show that the clean and flush latencies are ≈ 100 cycles per cache line, which is favorably comparable to and often lower than commercial x86 and ARMv8 CPU implementations. Furthermore, we demonstrate the advantages of an open architecture by introducing Skip It, a hardware optimization that significantly reduces the number of unnecessary flushes. Skip It performs up to 2.5× better in our experimental setup than FLiT [73] and similarly to other state-of-the-art software solutions, without requiring any software support. The implementation of the writeback instructions and Skip It are publicly available [24].

2 Background

To design and implement support for user-controlled cache writebacks, we built upon existing research and several open-source components. In this section, we provide the necessary background on the RISC-V ecosystem including the memory semantics, coherence, the Rocket and BOOM cores, Chipyard, and the existing cache control mechanisms. Deeper discussion of the BOOM core internals can be found in § 3.

2.1 RISC-V

RISC-V is a modular, open, reduced ISA [72], which eliminates licensing the ISA, removing impediments associated with developing hardware and related software toolchains. The RISC-V ecosystem provides a platform for experimenting with new instructions, microarchitectural features, and functionality. RISC-V has seen a plethora of open-source designs ranging from tiny embedded cores [68] to massively parallel architectures [78], in-order cores [6, 77] and complex out-of-order systems [79, 16, 76]. Several have been taped-out, and some have been commercialized [63, 51] with indications that server class RISC-V SoCs are on the horizon [69].

Chipyard [4] is a framework for agile RISC-V SoCs design. It integrates configurable open-source IP blocks, software register transfer level (RTL) and FPGA-accelerated simulation, automated VLSI flows, and workload generation for bare-metal and Linux-based systems. Rocket Chip [6] is an SoC generator that includes a core, caches, and interconnects. Several parameters are configurable, such as the FPU, cache size, and TLB size. The configurability and composability of these modules allow developers to quickly develop new designs.

2.2 TileLink and Coherence Protocol

TileLink [20, 67] is a chip-scale interconnect that provides low-latency connectivity between CPUs, accelerators, caches, memory, and other SoC peripherals. A link connects a client agent to a manager agent. An agent may be part of multiple links, acting as a client in some and as a manager in others. A legal TileLink network topology always forms a directed acyclic graph (DAG), and TileLink is deadlock-free.

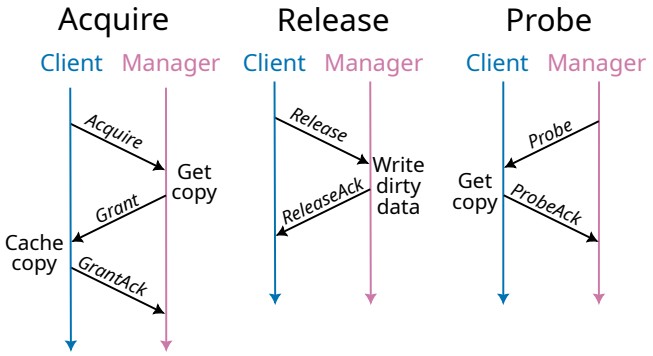

Figure 1. Three TL-C transactions are needed for coherence.

An agent-to-agent link consists of up to five unidirectional channels: $\{A, B, C, D, E\}$. A is used by the client to send read or write messages, D is used by the manager to send response messages. $\{B, C, E\}$ are required for cache coherence.

There are three main coherence messages, the transactions of which are shown in Figure 1. *Acquire* is used by the client to obtain permission from the manager to get a copy of a cache line. *Acquire* messages are sent over A and always trigger a *Grant* message from the manager on D. The client acknowledges the *Grant* with a *GrantAck* response on E. *Acquire* operations may also trigger the manager to generate recursive *Probe* or *Release* requests to other agents. *Release* operations are issued by clients on C to downgrade the permissions of a cache line voluntarily. The manager agent acknowledges it with a *ReleaseAck* message on D. The *Probe* message is issued on B by the manager to modify or revoke a client's permissions to a cache line. The client adjusts the line's permissions and responds with a *ProbeAck* message on C that may contain dirty data. These three messages can be used to implement a cache coherence protocol on RISC-V.

An example is MESI [55], an invalidation-based cache coherence protocol for writeback caches. It contains four primary cacheline states: modified, exclusive, shared, and invalid which can be maintained by the aforementioned TileLink messages. Modified and exclusive states imply a single owner, while a cacheline in shared state may have multiple owners. For example, the transition from shared to modified state and vice versa can be accomplished using the *Acquire* and *Probe* TileLink messages. When an agent requests a cacheline that is in modified state in another cache, a *Probe* message signals the exclusive owner of the cacheline to downgrade its permissions and writeback its dirty data. Similarly, when an agent wants to write data to a local copy of a cacheline in shared state, it must request for exclusive

permissions using *Acquire* which in turn triggers a *Probe* to revoke the copies of the line held by other agents.

2.3 Berkeley Out-of-Order Machine (BOOM)

The Berkeley Out-of-Order Machine (BOOM) [15] is an open-source, out-of-order RISC-V CPU generator developed in Chisel [7]. SonicBOOM [79] is the third-generation instance of BOOM and is a state-of-the-art platform for research in microarchitectural design. The core is parameterizable, synthesizable, and can be implemented on an FPGA. The BOOM core re-uses blocks from the Rocket chip [6] such as TLBs, page table walkers (PTWs), and the control status register (CSR) file, while implementing a new pipeline and data cache. The architecture of BOOM will be discussed in detail in § 3.

2.4 RISC-V Weak Memory Ordering

RISC-V weak memory order (RVWMO) is a weak memory model [72]. Its rules are primarily concerned with accesses to the same or dependent memory locations, but it still preserves *multi-copy atomicity*. Multi-copy atomicity means that a value becomes globally visible to all other processors at the same time. For a single-processor system, the global order corresponds to the program order. However, for a multi-processor system, a processor does not necessarily see that the instructions being executed by another processor correspond to the other processor's program order. Therefore, to guarantee ordering, one needs atomic memory operations, store conditional or load-reserve operations (SC/LR), or fence instructions.

2.5 Cache Management

There are multiple scenarios in which deliberate control of data in caches is essential for correctness. One example is non-volatile main memory (NVMM) where writing back data is fundamental to maintain a consistent state of the data. Without proper management, the order of the writes to a volatile cache is not necessarily preserved when data is written from cache to main memory. If caches are volatile, upon a crash, all cache content will be lost, but the content of the main memory will remain. Thus, later writes may be propagated to the NVMM before earlier writes, leading to data inconsistencies in the main memory.

Consistency is not only important in the case of NVMM, but in many cases in which devices share the same memory space. Disaggregated memory in which multiple nodes share an address space (e.g. CXL-attached memory [62]) correct write ordering must be guaranteed, similar to the NVMM scenario. Peripheral devices often share memory as well. Thus, for consistent DMA reads from main memory, the application needs to ensure that data is properly written-back before the DMA transaction is initiated. Only by calling explicit write-back instructions, can programmers control the ordering in which these writes are written-back to memory.

There are various ways to explicitly writeback data to main memory: *invalidating* and *non-invalidating*, *synchronous* and *asynchronous*. A synchronous writeback blocks following instructions while an asynchronous one can occur at a later point in time subject to memory model constraints. An invalidating writeback, which we will refer to as a *flush*, invalidates the cache line when the data is written back. A non-invalidating writeback, a *clean*, leaves the cache line valid but ensures the modified data reaches main memory.

Modern ISAs have several variants of these instructions, but we limit our discussion to the most relevant. An asynchronous clean is a `clwb` and `dccvac`, while an asynchronous flush is `clflushopt` and `dccivac` in x86 and ARM respectively [33, 5]. To make any of these writebacks synchronous, one can use a *fence*, or in x86, a synchronous flush `clflush`. The RISC-V ISA defines cache management instructions (§ 2.6), however explicit writebacks (clean and flush) are unimplemented.

2.6 RISC-V Cache Control

The `FENCE` instruction in RISC-V is used to force an ordering of the program's execution. It is defined as `FENCE PRE, SUC`, where `PRE` (predecessor) and `SUC` (successor) are flags indicating that the set of operations in `PRE` must complete before the set of operations in `SUC` are allowed to execute.

The RISC-V Cache Management Operations (CMOs) [60], introduce instructions that operate on data in the memory hierarchy. Of interest to us are the `CBO.CLEAN` and `CBO.FLUSH` cache management instructions. They operate on the set of coherent caches accessed by the agent executing the instruction. If a cache line is dirty, `CBO.CLEAN` propagates it to all higher level caches and writes it to memory; however, copies still remain in all caches that possessed one. `CBO.FLUSH` atomically invalidates and cleans the cache line.

Even though these instructions are defined in the RISC-V ISA, the only instruction which has been implemented on all platforms is `FENCE`. This user-level fence ensures that all memory operations before the `FENCE` are completed before any memory operations after the `FENCE` are handled. SiFive released a vendor instruction extension `CFLUSH_D_L1` that could evict from the L1 data cache using TileLink. If an L2 cache is present, the original `CFLUSH_D_L1` instructions will only flush the dirty data to L2. Moreover, `CFLUSH_D_L1`, limited as it is, was only implemented for the in-order Rocket Chip making the instruction unusable on the BOOM core.

3 BOOM Architecture

This section examines relevant internal architecture of the BOOM core and related components. While some of this information is available in [79], much of the architecture and behavior is undocumented outside of the code. We aim

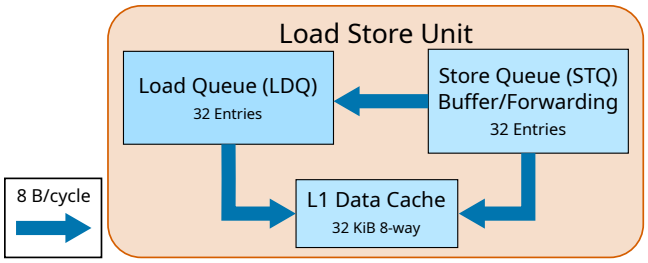

Figure 2. SonicBOOM Load Store Unit (LSU) contains 32 entry load (LDQ) and store (STQ) that transmit data at 8 B/cycle.

to provide necessary background to understand our implementations and a methodical description of the underlying architecture.

3.1 Reorder Buffer

The SonicBOOM's re-order buffer (ROB) is a queue that tracks the status of all in-flight instructions in the pipeline. The ROB consists of a *bsy* flag that tracks whether the instruction is currently executing. Once an instruction completes execution, the *bsy* flag is unset, which increments the head of the ROB, and the instruction is considered committed. The ROB provides the illusion that a program executes in-order, for each processor.

3.2 Load Store Unit

The load-store unit (LSU) of the SonicBOOM (Figure 2) is capable of *firing* two requests per cycle to the memory system. Firing refers to the issuing of the instruction to the data cache from the LSU. The LSU contains two queues: the load queue (LDQ) and the store queue (STQ). LDQ entries contain only the load's address, while STQ entries contain both the store's address and data, thus acting as a store buffer.

The LDQ tracks the status of load instructions, and fires them out-of-order as soon as their data is ready. The STQ only fires a request when the head of the ROB points to the same request, ensuring that stores are fired in order. Incoming loads probe the STQ and may forward data from an STQ entry if a match is found and data is available in the store buffer. Additionally, the LSU is responsible for detecting and marking dependencies between requests in the queues. This ensures that reordering does not violate the guaranteed memory model.

Fence instructions, as described in § 2.6 and § 4, ensure that all previous memory operations, in program order, are completed before subsequent operations are fired into the data cache. Fence instructions are stored in the STQ of the LSU. All LDQ instructions that arrive after the fence are marked as dependent on the fence. This prevents subsequent loads (or any LDQ request) from being issued before the fence is completed. The fence, by virtue of being an STQ request, is guaranteed to not be executed before previous instructions, in program order, are committed.

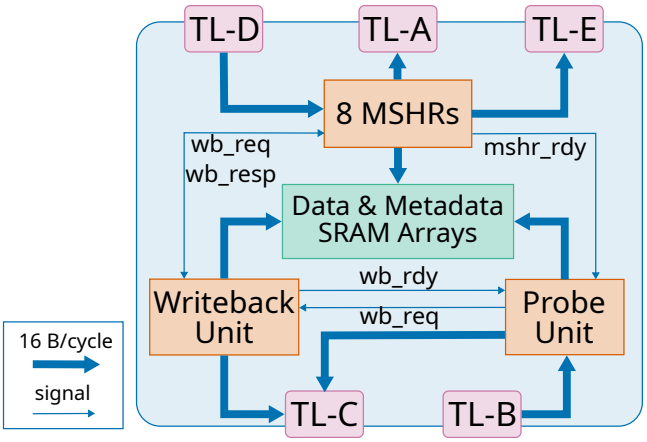

Figure 3. SonicBOOM L1 Data Cache contains 16 B data buses and signals. Its external connections are via TL-X.

3.3 L1 Data Cache

The SonicBOOM contains a non-blocking 32 KiB 8-way writeback L1 data cache. Figure 3 provides a high-level overview of its components.

The data cache contains two static RAM (SRAM) arrays for data and metadata. The metadata array stores information on cache lines such as their tag, coherence state, and the dirty bit. The data array stores the actual data.

When a request arrives in the data cache, the tag and index are compared with the metadata. On a hit, the request is served immediately. Otherwise, a miss status holding register (MSHR) is allocated to serve the request. MSHRs are responsible for retrieving the cache line, updating the data and metadata arrays, and replaying the request.

Multiple misses to the same cache line may be served by a single MSHR. This is accomplished by using the replay queue (RPQ), which exists for every MSHR. Upon a cache miss, it probes the MSHRs to check if one has already been allocated for this line. If so, the instruction may be buffered by the RPQ. Upon successful cache line retrieval, the RPQ is drained and all requests are replayed in the data cache in order of arrival. The request that caused the initial MSHR allocation is referred to as the *primary request* and all subsequent requests that piggy-back are called *secondary requests*.

The data cache of the SonicBOOM does not currently support upgrading permissions of a cache line via the TileLink message *AcquirePerm*. Therefore, the RPQ only accepts a secondary request if the permissions required are less than or equal to the primary request permissions. For example, if the MSHR was allocated as a result of a load, it is unable to accept a store as a secondary request. If the data cache is unable to handle a queued request, it sends a negative acknowledgement (nack) to the LSU. This may occur when there

are no free MSHRs to handle the request or if a matching MSHR exists but is unable to accept the secondary request. On receiving a nack, the LSU retries at a later time.

The writeback unit (WBU) releases dirty data to a higher level cache. The probe unit handles coherence probes from higher level agents, and may request the WBU to release dirty data. It is possible that a cache line being written back is simultaneously probed. As shown in Figure 3, the WBU may temporarily hold probes by signaling via *wb_rdy*. The data cache has a TileLink to a higher level memory agent. This link is used by the MSHRs to acquire data, the probe unit to receive probes, and the WBU to release cache lines.

For stores, or broadly STQ instructions, it is important to observe that they are only fired into the data cache when they are committed, and not before as loads are. Conversely, when these instructions are in the data cache (even if they are waiting in MSHRs), they are considered by the ROB to have completed. Consider two store instructions arriving in the data cache one after the other. If the subsequent store were to miss and the previous were to hit, the subsequent store would be completed immediately while the previous would be allocated an MSHR. However, two consecutive writes cannot be reordered. In this case, a probe to the pending MSHR must wait until the store completes, thus implementing a slightly stronger memory model than the RISC-V weak memory order. This is accomplished using the *mshr_rdy* signal (Figure 3).

3.4 Last Level Cache

The SiFive inclusive cache [64] is a parameterized last-level cache generator included in Chipyard. The SonicBOOM can be configured with this module as its L2 cache. It enforces coherence among a set of caching agents using an invalidation-based coherence policy implemented using a full-map of directory bits stored with each cache line's metadata.

A detailed discussion of the L2 cache, shown in Figure 4 is beyond the scope of this paper; we restrict ourselves to relevant components. *SinkC* is a module that receives incoming TileLink C (TL-C) requests. The *ListBuffer* is a generic hardware queue that stores buffered TL-C requests which are later scheduled to the L2 MSHRs. The *SourceD* module generates TileLink D (TL-D) responses. The *BankedStore* contains the actual cache line data. Finally, the *Directory* stores cache lines metadata (i.e. dirty bit, state of the line, and owners of the line). As seen in Figure 4, the L2 cache contains a pair of TileLinks. The inclusive cache acts as the manager to L1 caches and as the client to main memory.

4 Memory Semantics

Before explaining the implementation details of `CBO.CLEAN` and `CBO.FLUSH` we present the memory semantics of these instructions, and specify their interactions with fences, because the writeback semantics are not defined for RISC-V.

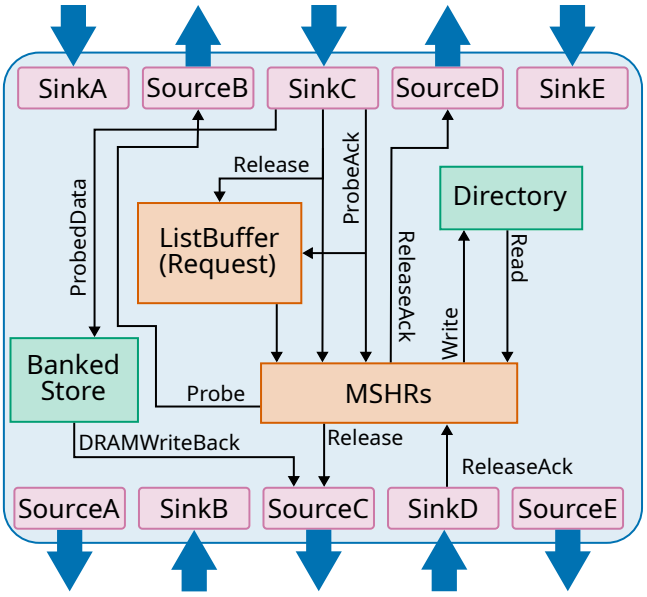

Figure 4. The SiFive inclusive cache contains (meta)data arrays, buffers, and connects to L1 caches and main memory via TileLink.

The full RISC-V memory model for the other instructions is described in [47, 48]. While Pelley *et al.* [56] describe general persistence model semantics, which may be similar or different from the consistency semantics, in this section we refer to the semantics of the lower level instructions. We refer to the instructions `CBO.FLUSH` and `CBO.CLEAN` in § 2.6 collectively as writeback instructions. Writebacks and fences enable the eviction of a specific cache line associated with a particular address at a given point in time.

Consider a memory location c and its associated cache line C such that $c \in C$. *writeback(c)* guarantees that all writes to $c' \in C$, where c' is any memory location associated with C, that happened before that *writeback(c)*, are written back to memory. Thus, a *writeback(c)* is ordered only with respect to all previous writes to C, but not with respect to subsequent writes.

To improve performance and accommodate multiple writebacks in parallel to different memory locations, writeback instructions are handled asynchronously. Now consider two addresses c_1 and c_2, represented by the cache lines C_1 and C_2. *writeback(c_1)* followed by a *writeback(c_2)* does not guarantee that all writes to C_1 that happened before *writeback(c_1)* are written back to memory before C_2 is written back by *writeback(c_2)*. Moreover, to be aligned with BOOM's out-of-order execution, the asynchronous writeback is committed out-of-order as well, and does not necessarily complete its execution before the next instruction is committed.

Figure 5 demonstrates three scenarios. In (a), even though x and y are committed in order, because data is cached, the

(a) No Writeback	(b) Only Writeback	(c) Writeback & Fence
1: x = 1 2: y = x	1: x = 1 2: **writeback(&x)** 3: y = x	1: x = 1 2: **writeback(&x)** 3: **fence()** 4: y = x

Figure 5. Memory semantics via three writeback scenarios.

actual write-back to memory can happen at any point in time. Therefore, y might be written back before x.

In line 2 of scenario (b), writing back the cache line X that holds x is asynchronous and ordered with respect to all earlier writes to X. As mentioned, writing back is done asynchronously to improve performance, and is not handled in order with respect to other writebacks. As a result, in (b), y is not necessarily written back to memory after x, even though *writeback(x)* was issued before the write to y.

Sometimes, however, a cache line needs to be written back at a specific point in execution to maintain certain guarantees. Using a fence ensures that memory instructions are not reordered with respect to this fence. Memory instructions that appear prior to the fence in program order cannot be reordered with respect to instructions that appear after the fence. More specifically, the fence implies that all memory operations (including writebacks) prior to the fence are committed before those that follow the fence are executed. The RISC-V ISA defines 6 practical fence instructions of various strengths [72]. Here, we refer to the strongest fence, i.e. FENCE RW, RW that is the only one that is implemented on the BOOM core and we show how its semantics are extended with respect to writeback instructions. Given a location c, *writeback(c)* followed by *fence()*, guarantees that all writes to cache line C, that happened before *writeback(c)*, are written back to memory before any memory instruction that follows the *fence()* is executed by the same thread. Thus, in scenario (c), by calling *fence()* after *writeback(&x)*, it is guaranteed that by the time that x is assigned to y in line 4, the updated value of x will reside in main memory.

These semantics are closely related to ARMv8's writeback semantics [59], i.e., a writeback of a cacheline is only ordered with respect to previous writes to the *same* cacheline. In x86 [58], on the contrary, due to its TSO nature, a writeback of a cacheline is ordered with respect to all previous writes to *any* cacheline. Furthermore, The only type of a fence/barrier that is implemented on the BOOM core is the strongest fence, and has an equivalent behavior to the strongest fences on x86 and ARMv8.

We note that the RISC-V memory model is weak and does not define the writeback semantics formally, thus we define these writeback semantics accordingly. However, the BOOM core implements only the reordering of loads, providing a stricter model. Because we encode our writeback instructions as a store in § 5.1, writeback instructions are ordered with respect to any previous write to any cacheline on the BOOM core, similar to x86. In addition, it is also ordered with respect to subsequent writes to the same cacheline. As mentioned, flushes are asynchronous and not reordered with respect to one another. To guarantee ordering, a *fence()* must be used.

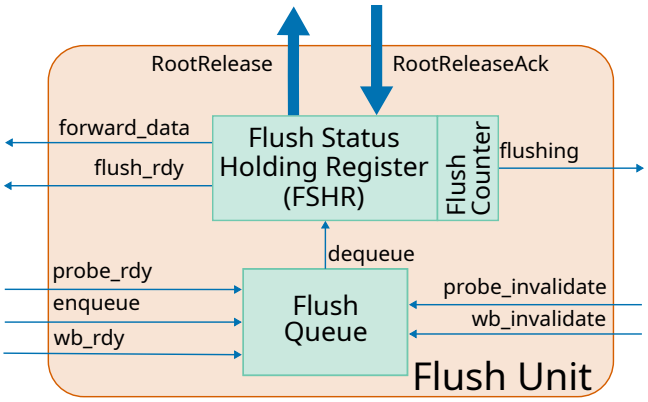

Figure 6. The Flush Unit provides the mechanisms to support `CBO.X` instructions via new signaling, FSHRs, a flush queue and a flush counter.

5 Flush Microarchitecture

This section describes our design and implementation of the *flush unit* and associated architectural support for `CBO.X` (`CBO.FLUSH` and `CBO.CLEAN`) instructions in the BOOM core. We present the flush unit's architecture and discuss the fence instruction modifications to ensure compatibility with `CBO.X`.

5.1 Encoding

`CBO.X` are encoded as STQ requests in the LSU, ensuring that the request is only fired into the data cache when the ROB head points to this instruction. As discussed in § 3.2 this means `CBO.X` instructions are executed in program order.

We introduce two new TL-C messages: *RootReleaseFlush* and *RootReleaseClean* (collectively *RootRelease*). To avoid expanding the op-code bitvector, we encode the messages internally as TL-C *ProbeAck* with parameters *FLUSH* and *CLEAN*. We additionally introduce a TL-D message *RootReleaseAck* which is used to acknowledge *RootRelease* requests. This is encoded as a TL-D *ReleaseAck* message with parameter *ROOT*.

5.2 Flush Unit

The flush unit is a crucial component of implementing microarchitectural support for RISC-V CMOs (§ 2.6) in the BOOM. It is integrated into the data cache and is responsible for handling writeback requests. Figure 6 provides a high-level overview of the flush unit's components. We emphasize that because a writeback unit already exists in BOOM, we denote

this component as the Flush Unit, even though it handles both `CBO.FLUSH` and `CBO.CLEAN`.

Flush Queue. The flush queue buffers all incoming `CBO.X` requests. Some relevant fields of a flush request are: **addr**, the address to be written back; **is_hit**, does the cache line hit?; **is_dirty**, is the cache line dirty? (only relevant if hit = 1); **is_clean**, is this a `CBO.CLEAN`? This bookkeeping data is directly inferred from cacheline metadata which is always fetched with data cache requests. Thus, no overhead is added to the transmission or storage of a cacheline's metadata.

We designed the flush request to contain all pertinent cache line information to reduce metadata array contention. Flush unit performance could be degraded by an arbitrary number of cycles due to metadata contention among flush status holding registers (FSHRs), MSHRs, the probe unit, and the request queued into the data cache. However, we note that the consistency of queue entries must be ensured as an unspecified amount of time may pass between enqueuing and dequeuing a flush request. We explain how we preserve consistency in § 5.3.

When the LSU queues a `CBO.X` request into the data cache, it is accepted by the flush unit if it has free space in the flush queue to buffer the request. If the flush queue is full, the data cache sends a nack to the LSU and the LSU will retry at a later point in time. Because the LSU tracks all dependencies and ordering between requests (§ 3.2), a request that nacks as a result of a full flush queue does not violate the program ordering. Once a `CBO.X` request is buffered, the corresponding instruction is considered to be ready for commit. This means that subsequent STQ and dependent (to the same cache line) LDQ requests are free to proceed.

The flush queue is highly beneficial during periods with many writeback instructions. By buffering and thus allowing the LSU to commit multiple writeback instructions, it frees the LSU to handle newer requests. Furthermore, it also reduces contention in the data cache, as buffered requests are considered completed and are no longer retried by the LSU (similar to stores forwarded to an MSHR).

Flush Status Hit Register. FSHRs are responsible for asynchronously executing `CBO.X` requests. The flush unit contains 8 FSHRs. Every cycle, if the flush queue is non-empty, a free FSHR is allocated by dequeuing the request from the flush queue head. FSHRs are round robin allocated [65]. Each FSHR contains a data buffer used to temporarily store data being written back to main memory.

The data array of the BOOM only supports reading one word per cycle, therefore taking multiple cycles to retrieve all the data of a cache line. We modified the data array by widening the output to one full cache line. We then connected the outputs to the entire cache line indexed, instead of only connecting the output to the word retrieved by the offset into the cache line. Thus, we have optimized the data array to serve an entire cache line in a single cycle.

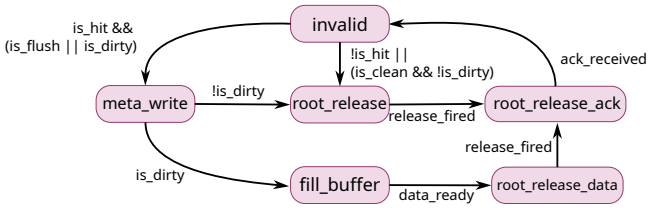

Figure 7. The Flush MSHR operates via the following state machine. It begins in the **invalid** state and ends in **root_release_ack**.

For `CBO.FLUSH`, the metadata must always be invalidated while for `CBO.CLEAN`, only the dirty bit is unset if the requested cache line is dirty. Otherwise, the metadata is unchanged. For both `CBO.CLEAN` and `CBO.FLUSH`, if the requested line is dirty, the data buffer must be filled with the dirty data. In all cases, the FSHR must send a *RootRelease* to L2 and waits for a *RootReleaseAck*. Each FSHR operates based on the state machine depicted in Figure 7 that implements the aforementioned behavior.

The states are as follows:

1. **invalid**: Requests are ready to be accepted. On successful reception of a request, an execution plan is set up based on whether the address hit, the cache line is dirty, and if the request is a clean or a flush.
2. **meta_write**: The metadata of the requested line is modified. In case of a flush, the metadata is invalidated, while in case of a clean, the dirty bit is unset.
3. **fill_buffer**: The data buffer is filled by retrieving the correct line from the data array of the L1 data cache. As mentioned earlier, the data array takes only one cycle to fill the whole buffer.
4. **root_release_data**: A *RootRelease* message is sent with data over TL-C to the higher level cache to write back data to main memory. Because the SonicBOOM's system bus is 16 B (Figure 3), it takes four cycles to send the data to L2.
5. **root_release**: A *RootRelease* message is sent without data over TL-C. As no data is written back, the message is sent in one cycle.
6. **root_release_ack**: The FSHR waits until it receives a *RootReleaseAck* over TL-D. Once the acknowledgement is received, the FSHR transitions back to the *invalid* state.

We summarize the potential paths shown in Figure 7 from the `invalid` state to the `root_release_ack` state. A `CBO.X` to a dirty cacheline must modify (invalidate or downgrade) the cacheline's permissions. However, in case of a hit on a *clean* cacheline, its permissions must be modified (invalidated) only in case of a `CBO.FLUSH`. In case of a cache miss, no action is required. Finally, the appropriate *RootRelease* is sent to the higher level cache. We highlight that in case of a cache miss, *RootRelease* is sent regardless as the cacheline

may still need to be written back from other cores, or from higher levels of the cache hierarchy.

Flush Counter. The flush counter is used to track the number of pending flush requests. It is incremented when a flush request is enqueued and decremented when an FSHR receives a *RootReleaseAck*.

5.3 Handling Data Cache Requests

`CBO.X` instructions are considered to be committed when they are buffered by the flush unit's queue. As subsequent instructions, including those dependent on the buffered `CBO.X`, may be issued to the data cache, we must take special care to ensure correctness. For this reason, the flush unit is designed to be probed to check whether the addresses are either pending in the flush queue or being handled by an allocated FSHR.

Loads. Loads (or broadly LDQ requests) can proceed if the request hits in the L1 cache. Because a load hit does not change the state of any cache line, the metadata of requests in the flush queue remain valid, and the request may be served. However, if the LDQ request misses but there exists an FSHR with a filled data buffer handling the same line, this data is forwarded directly to the load, and the load succeeds. If the load misses and an FSHR is handling the same line without a filled data buffer, then the load must be postponed (nacked) until either the data buffer is filled or that FSHR completes serving its request. This is essential because the metadata of requests in the flush queue must not be modified by the same core.

Stores. If a store is dependent on a request, either pending in the flush queue or being handled by an FSHR, the store will be nacked unless it satisfies all the following three conditions. First, there must be an allocated FSHR for the requested line. Second, the FSHR must be executing a `CBO.CLEAN` request. Finally, the cache line must not be dirty or, if it is dirty, the FSHR must have filled its data buffer. Upon satisfying these conditions, the store instruction is allowed to proceed without waiting for the write back to complete.

As cache line permissions remain unchanged due to a `CBO.CLEAN`, subsequent stores may proceed without an acknowledgment of the `CBO.CLEAN` request. Nevertheless, by ensuring that the data buffer is filled before proceeding with subsequent stores, we guarantee that the data of subsequent stores are not written back by the `CBO.CLEAN` instruction. Dependent `CBO.X` instructions are allowed to coalesce with a corresponding pending flush request if and only if the cache line state remains unchanged between the two `CBO.X` instructions. For example, a `CBO.CLEAN` may coalesce with a pending `CBO.CLEAN` but not with a pending `CBO.FLUSH`. However, `CBO.X` requests of the same kind may not be merged together if they operate on different cache lines. When a subsequent `CBO.X` satisfies the conditions for coalescence, it is marked as committed and is dropped. This improves performance by optimizing away redundant writeback instructions in hardware. Coalescence of different types of `CBO.X` requests, such as a `CBO.CLEAN` with a `CBO.FLUSH` is a potential optimization for future investigation.

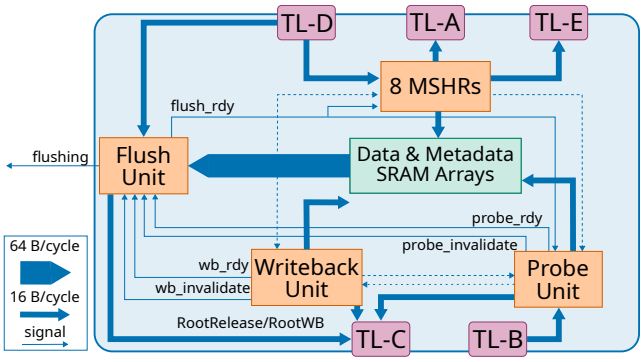

Figure 8. The modified SonicBOOM L1 data cache with the addition of the Flush Unit contains all of the previously described signals (denoted by dashed lines) in addition to new signals and data buses.

All other dependent STQ requests must nack the LSU. This is done to prevent the data of subsequent stores from being written back by previous writeback instructions.

Fences. Fences are adapted to also ensure completion of all pending flush requests before committing. This is achieved using the flush counter. The *flushing* signal in Figure 6 is set when the flush counter is non-zero, implying that if *flushing* is low, there are no pending flush requests in the queue or in FSHRs. We permit fences to commit only if *flushing* is low.

5.4 Writeback Interference

When an FSHR is allocated and a request is dequeued, an arbitrary amount of time has passed since the time of enqueue, thus the request metadata is not guaranteed to be valid. The request metadata may have been modified as a result of the following actions:

1. Out-of-order execution of dependent memory operations.
2. Probes to the address by other cores.
3. Cache line eviction by the MSHRs.

As presented in detail in § 5.3, action 1 is prevented since operations dependent on queued flush requests are blocked. Actions 2 and 3 are discussed in § 5.4.1 and § 5.4.2.

5.4.1 Cross-Core Interference. Cache coherence probes that revoke line permissions may cause entries in the flush queue to become invalid. Consider the case in which a flush request is enqueued into CPU1 with hit and dirty bits set. If CPU2 requests permissions to this line from the L2 cache, CPU1 is probed and must revoke its permissions. If the probe happens before the flush request is dequeued and allocated

to an FSHR, it leads to undefined behavior. We avoid this by augmenting the probe unit and the flush queue to allow invalidation of pending entries in the flush queue. On handling a probe request from a higher level caching agent, the probe unit signals the flush unit via the *probe_invalidate* input (Figure 6) to reset the dirty and/or hit bits based on the permissions downgrade level. As a result, this flush request is handled correctly with updated and valid metadata upon dequeue.

However, this does not address the case in which a line is probed while simultaneously being handled by an allocated FSHR. This situation is resolved via *flush_rdy*. Upon FSHR allocation, *flush_rdy* is unset until the FSHR reaches the state **root_release_ack**. The signal remains unset until the FSHR completes writing any metadata and releasing the line to L2, but before receiving the acknowledgment. The probe unit is blocked from handling the probe if *flush_rdy* is not high, ensuring that probes cannot preempt an allocated FSHR.

Finally, consider the corner case of a probe event arriving and being handled initially without any conflicting FSHRs. Then, a flush request to the same line is dequeued before the probe unit is able to invalidate this request in the queue. This scenario is rendered impossible by the use of *probe_rdy*. As soon as a probe event arrives, and before the probe unit invalidates conflicting entries in the flush queue, the *probe_rdy* signal is held low. The flush unit is only permitted to dequeue a flush request and allocate an FSHR when *probe_rdy* is high. It is possible for *probe_rdy* and *flush_rdy* to be lowered simultaneously if a probe request arrives and a flush queue request is dequeued at the same time. However, the probe unit checks for *flush_rdy* only one cycle after lowering *probe_rdy*. Thus, the flush queue request that was dequeued at the same time as the probe arrived, completes first and raises *flush_rdy*. As *probe_rdy* remains high, no other request can be dequeued from the flush queue in the meantime, and the probe can continue, guaranteeing deadlock freedom in our design.

5.4.2 Cache Line Eviction. Owing to the out-of-order nature of the SonicBOOM, validity of request metadata in the flush queue may be voided by cache lines being evicted to accommodate other cache lines retrieved from higher level caches. Cache line evictions release lines to the higher level cache via the writeback unit (§ 3.3). Similar to the case of probes in § 5.4.1, we augment the writeback unit to invalidate requests corresponding to lines that are evicted. *flush_rdy* is used again to block the MSHRs from choosing a cache line to evict. *wb_rdy* in the writeback unit, used to block probes, is reused to also block flush queue requests from being dequeued. As the synchronization using *flush_rdy* and *wb_rdy* is identical to that of *probe_rdy*, we delegate the discussion of deadlock freedom to § 5.4.1.

5.5 L2 Cache

We modified the SiFive inclusive cache to add support for handling *RootRelease* messages. As described in § 5.1, *RootRelease* is encoded as *ProbeAck*. Figure 4 demonstrates the pathways of such a *ProbeAck* message in the L2.

The *ProbeAck* request is allocated to an MSHR in L2 immediately upon arrival or later if buffered into the *ListBuffer* due to a lack of MSHRs or conflicting MSHRs. If it contains dirty data, it is simultaneously written back to the *BankedStore*.

Once an MSHR is allocated to handle the request, the directory is probed to get the cacheline's dirty bit, and all the lower level agents (L1 caches) that possess permissions to this cacheline. First, the MSHR determines the probing strategy. In case of a *RootReleaseFlush*, the MSHR first recursively probes other owners of the line and revokes their permissions. For a *RootReleaseClean*, probing is performed only if an agent possesses the line with write permissions and the owner (more than one owner may not posses write permissions) is not the agent that requested the *RootReleaseClean*. The MSHR instructs lower level caches to revoke/downgrade permissions on their copies of the cacheline via TL-B messages as discussed in § 2.2. We highlight that probing and cacheline revocation are performed if necessary even in case the core that sent the *RootRelease* did not possess the cacheline. This is important for correctness as the cacheline must be written back to DRAM irrespective of the permissions on the line held by the requesting core.

Upon receiving acknowledgements for the probes, any dirty data obtained is written back to the *BankedStore*. Finally, if the cacheline is dirty (or has been dirtied by the acknowledgements to the probes), the dirty data is released and written back to DRAM via the *SourceC* module using the *Release* TL-C message described in § 2.2. Thus, the last level cache already catches and eliminates unnecessary writebacks by trivially checking its dirty bit.

Upon a successful acknowledgment of the *Release* from main memory, the MSHR acknowledges the completion of the writeback by sending a *RootReleaseAck* (encoded as *ReleaseAck*) to the original requester via the *SourceD* module.

6 Skip It

A cache line that has no dirty data in all levels of the cache hierarchy is said to be *persisted* to main memory. As writing back a persisted cache line does not change the line's state in main memory, this operation can be safely skipped, preventing unnecessary overhead while maintaining correctness. In this section, we outline the architectural details of Skip It, a hardware optimization that reduces costly redundant writeback operations in the same manner as the software mechanisms FliT [73] and link-and-persist [23].

Unfortunately, the dirty bit and valid bit of the L1 alone are not sufficient to identify unnecessary writebacks. This is because a cache line may or may not be dirty in L2, even if

the dirty bit in the L1 cache is unset. Consider the following two scenarios:

- A cache line is not dirty in the L1 cache, but the same cache line is dirty in the L2 cache.
- A cache line is not dirty in the L1 cache, and the same cache line is not dirty in the L2 cache.

For Scenario 1, the write-back instruction must be issued because dirty data exists in some level of the cache hierarchy. However, in Scenario 2, we may safely skip the writeback because the cache line has no dirty data in any level of the cache. But, it is impossible to distinguish between these two scenarios with just the L1 cache's dirty and valid bits, and so we require another bit.

Therefore, to track the persistence status of a cache line, we add the skip bit to the metadata of each line in L1. As persisted cache lines do not need to be written back, we skip writebacks to cache lines whose skip bits are set. Lines with an unset skip bit must be written back. Proof of our assertion that the skip bit, when valid, is equivalent to the negation of L2's dirty bit is presented in § 6.2. We also introduce a new TL-D message called *GrantDataDirty*, sent as a response to *AcquireBlock*. *GrantDataDirty* is functionally identical to *GrantData*, except that reception of a *GrantDataDirty* indicates that the cache line is not persisted, while *GrantData* indicates it is. Therefore, an agent must unset the skip bit on reception of *GrantDataDirty*, while *GrantData* signals the agent to set the skip bit.

6.1 Implementation

Writeback requests in the flush unit now include a *skip_bit* flag. The metadata array in Figure 3 is modified to store the skip bit for each cache line. This bit, combined with the dirty bit, indicates whether data in the L1 cache exists in main memory. If the writeback request hits the cache, is not dirty, and the skip bit is set, the writeback request is dropped and not enqueued into the flush queue. The data cache signals success to the LSU and the instruction is committed.

When the data cache's MSHR receives a response to *Acquire*, it unsets the skip bit of that block if the response is *GrantDataDirty*, and sets it otherwise. The L2 cache, upon responding to *Acquire*, selects between *GrantData* or *GrantDataDirty* depending on whether the block in L2 is dirty.

6.2 Correctness

We say the skip bit in L1 is valid if the cache line is valid and the dirty bit is unset. We claim that when the skip bit of a line in L1 is valid, the skip bit is equivalent to the negation of the dirty bit of that line in L2.

Let us consider all the possibles states of a cache line:

1. **Invalid**: The cache line is invalid in L1, therefore the skip bit is also invalid as it is impossible to know the status of the cache line in L2.
2. **Write Permissions**: The L1 has exclusive write permissions to the cache line and we know that no other L1 cache contains a copy of the line. If the dirty bit of the line is set, the skip bit is treated as invalid because we must writeback dirty data. However, if the dirty bit is unset, then we writeback based on the skip bit. Since only one L1 can have the data exclusively with write permissions, if the skip bit is unset, and a writeback is executed, it will not be able to skip it as the dirty data does not exist in main memory yet. Nonetheless, if the skip bit is set, no other cache line has modified the line, and therefore, it is safe to skip it.
3. **Read Permissions**: The L1 has read permissions to the cache line; this may be shared with other L1 caches or may be exclusive. Since shared or exclusive read permissions cannot pollute the data, the skip bit state remains unchanged. Therefore, if a skip bit is (in)valid, it remains unchanged since it is impossible for any other agent to update the line in the meantime. However, when the line is shared in multiple caches with unset skip bits, and they all perform non-invalidating writebacks (`CBO.CLEAN`), the skip bit may not be updated in time, allowing some redundant writebacks. While some non essential writebacks go through in this specific case, correctness is not violated.

Therefore, as all possible cases have been verified, we conclude that Skip It does not produce undefined or illegal behavior and is functionally correct.

7 Evaluation

To evaluate the clean and flush capabilities we examine several scenarios. § 7.2 evaluates SonicBOOM `CBO.X` performance across thread count and writeback sizes. § 7.3 compares clean and flush to commercial x86 and ARM CPUs. We demonstrate the effectiveness of Skip It in § 7.4 and compare its performance to state-of-the-art software solutions.

7.1 Experimental Platform

We develop our mechanisms extending the latest version of the SonicBOOM[1] and L2 cache[2] provided in Chipyard.

For experiments in § 7.2 and § 7.3, we synthesize a dual core SonicBOOM running at 30 MHz using FireSim [39][3]. Each core has a 32 KiB L1 cache and share a 512 KiB inclusive L2 cache. FireSim performs an FPGA-accelerated RTL simulation with FASED [10], a realistic FPGA-hosted DRAM model. The SonicBOOM is running Buildroot GNU/Linux (kernel v6.2) generated using FireMarshal [57].

Due to the more intensive nature of the evaluations in § 7.4, we slightly modify the experimental setup. The SoC remains unchanged but we use FPGA synthesis on Enzian [18].

[1]BOOM commit hash: deae9f7
[2]L2 commit hash: 850e121
[3]FireSim commit hash: df095fb

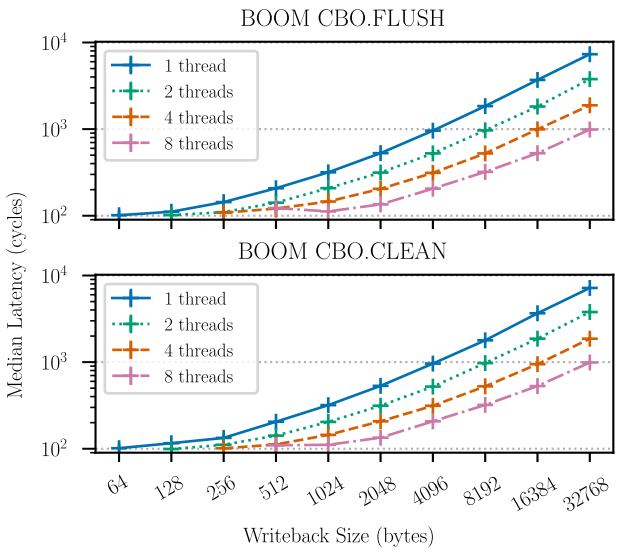

Figure 9. `CBO.X` performance scales as expected with increasing data size. Additional threads provide close to theoretical improvement (e.g. 8 threads writeback 7.2× faster).

Figure 10. We see the value of a cleaning (non-invalidating) writeback. Reading after CBO.CLEAN, we achieve ≈ 2× lower latency due to the cache hit versus refetching from memory due to CBO.FLUSH. Note latency is log scale.

Without the complexity added by FASED, our dual core SonicBOOM runs at 50 MHz. Since the comparisons are exclusively between SonicBOOM with and without Skip It, accurate memory simulations are less critical. Cycle counts are obtained using the *RDCYCLE* [26] control status register [25]. We repeat all microbenchmarks 50 times and report the median latency of the experiments. All other evaluations are performed for 2s and averaged over 5 repetitions.

7.2 SonicBOOM Performance

To characterize the `CBO.X` performance, we writeback various amounts of data, from one cache line (64 B) to the entire L1 cache (32 KiB) using one, two, four, and eight threads. Figure 9 shows the latency for non-contended lines i.e. each thread flushes a different cache region. We dirty the cache, then each thread flushes sequentially and fences once at the end. A single thread can clean or flush one cache line with a median latency of 100 cycles (σ: 13.2) while it takes 7460 cycles (σ : 286.1) to flush all 32 KiB. Additional threads reduce overall latency, especially for larger writebacks: 8 threads show a 7.2× improvement in latency over a single thread.

The `CBO.CLEAN` and `CBO.FLUSH` performance is equivalent in isolation because the mechanism is identical in this case. To compare the difference between them, we perform another microbenchmark that writes to a cache line, issues a clean or flush followed by a fence, and then re-reads the value once the synchronous barrier is passed. Figure 10 shows this benchmark for one and eight threads. The behaviors remain similar across thread count. By simply cleaning and not invalidating data (thus not having to re-fetch) allows ≈ 2× lower latency.

7.3 Comparative Performance

We compare our implementation in the SonicBOOM to commercial x86 and ARM CPUs. These experiments are conducted with two modern x86 CPUs, an AMD EPYC 7763 [1] and Intel Xeon Gold 6238T [34], as well as an AWS Graviton3 [3] ARMv8 processor (C7g.metal [2]). For x86 CPUs we use both standard `clflush` and optimized `clflushopt` (flush) and `clwb` (clean). ARMv8 uses `dccivac` (flush) and `dccvac` (clean). After flushing we include the appropriate memory barrier. Both x86 systems run Ubuntu 20.04 (kernel 5.4.0-153) while the Graviton3 runs Amazon Linux 2. Note that the SonicBOOM has only two levels of cache (L1 and L2), while the commercial CPUs have L3. We do not believe that this significantly changes the relative performance given that memory latency dominates.

Figures 11 and 12 show the comparative performance for one and eight threads. In both scenarios, the Intel `clflush` takes an extremely long time for larger data due to its inherent use of barriers, while Intel's optimized flush is often the best performing x86 implementation. On the other hand, AMD's `clflush` and `clflushopt` perform nearly identically. The SonicBOOM `CBO.FLUSH` outperforms x86 processors for nearly all sizes with both one- and eight-threads, but Graviton's flush latency grows sub-linearly, making flushes greater than 4 KiB faster. In all cases, `CBO.X` are competitive with the commercial implementations.

The gap between Intel's `clflush` and other flushing instructions is not as large using 8 threads. Again, noting the aforementioned caveats about comparing SonicBOOM and

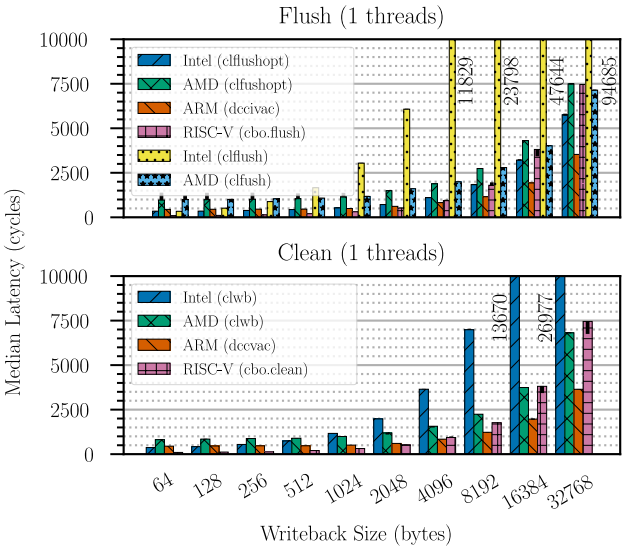

Figure 11. Writeback latencies for a single thread are similar across architectures, with the exception of the Intel `clflush` which is significantly worse at 4 KiB and above.

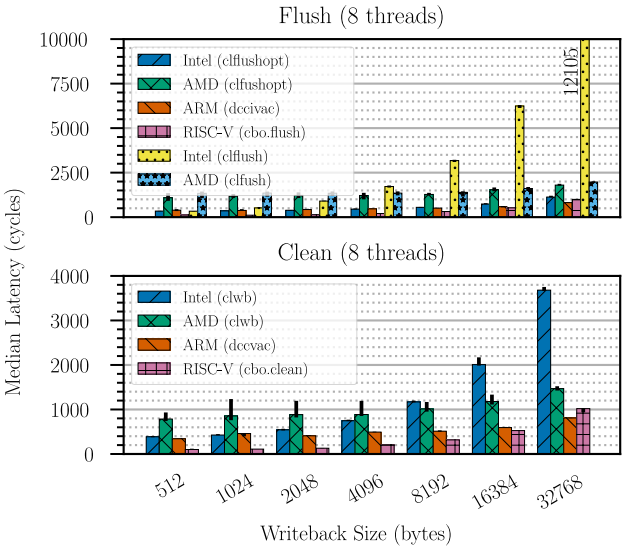

Figure 12. Writeback latencies for eight threads are comparable across architectures, with Intel `clflush` only showing its poor performance above 16 KiB in this case.

commercial CPUs, our implementation outperforms other platforms when using eight threads across nearly all writeback sizes. We recognize the significant additional complexity in commercial CPUs compared to the SonicBOOM core, and we want to be clear that these experiments do not imply that the presented implementation is *per se* better. There are many subtleties across architectures, vendors, and implementations and a thorough study of behavior and performance

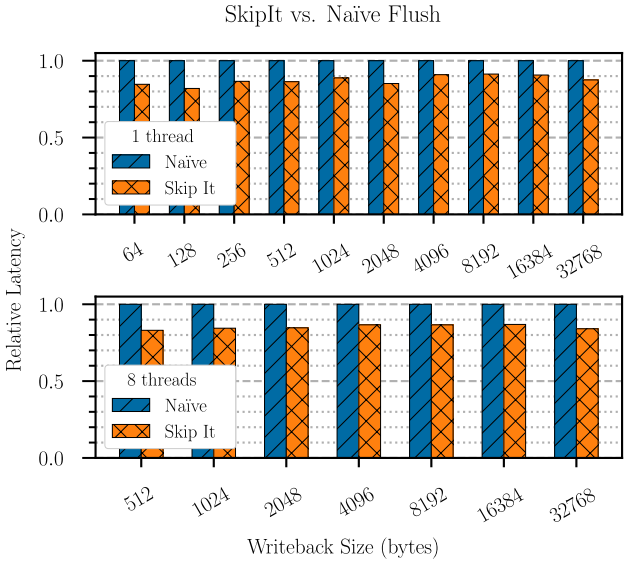

Figure 13. Comparing naïve and Skip It for 10 consecutive flushes, using 1 and 8 threads, Skip It shows 15-20% speedup.

is beyond the scope of this paper. However, we believe the design is a sound implementation and correct. Moreover, having a well performing, comparable set of instructions allows for further experimentation of hardware caching optimizations as we discussed in § 6 and evaluate in § 7.4.

7.4 Skip It

In this section we evaluate the performance of Skip It using a microbenchmark and compare it to state-of-the-art software optimizations for redundant writebacks.

Microbenchmark. To compare Skip It to naïve writebacks, we restrict our comparison to `CBO.FLUSH` as the results are identical for `CBO.CLEAN`. In the benchmark, for each cache line, we execute a store followed by a `CBO.FLUSH` and then 10 redundant `CBO.FLUSH` instructions to the same cache line. As with § 7.3, we evaluated it across 1, 2, 4 and 8 threads.

We observe in Figure 13 that Skip It performs up to 30% better than the naïve version. The L2 cache (as LLC) already supports trivially skipping redundant writebacks by checking its dirty bit. Thus, in the SonicBOOM, Skip It saves the latency incurred by queuing and dequeuing the flush request, allocating an FSHR, communicating with L2 and waiting for a response from L2. It also reduces contention for the L2 cache. A deeper cache hierarchy (i.e. L3 or L4) could show greater improvements due to the increased latencies.

FliT [73]. We compare Skip It to existing software-based optimizations of redundant writebacks. We use `CBO.FLUSH` to writeback cachelines to maximize the penalty of not identifying a redundant writeback. We evaluate these methods on persistent lock-free versions of four data structures: a binary search tree [53], a hash table [23], a linked list [31]

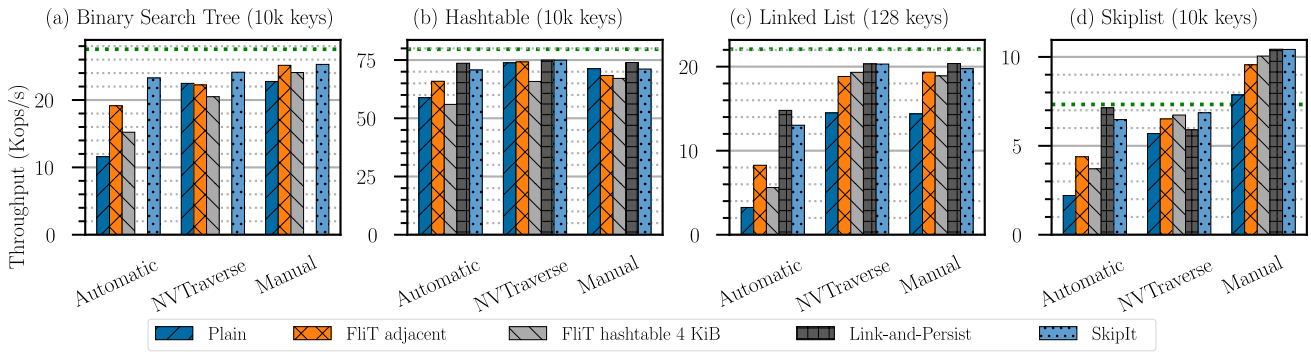

Figure 14. Throughput of varying persistent algorithms across data structures and flush optimizations, with 5% updates.

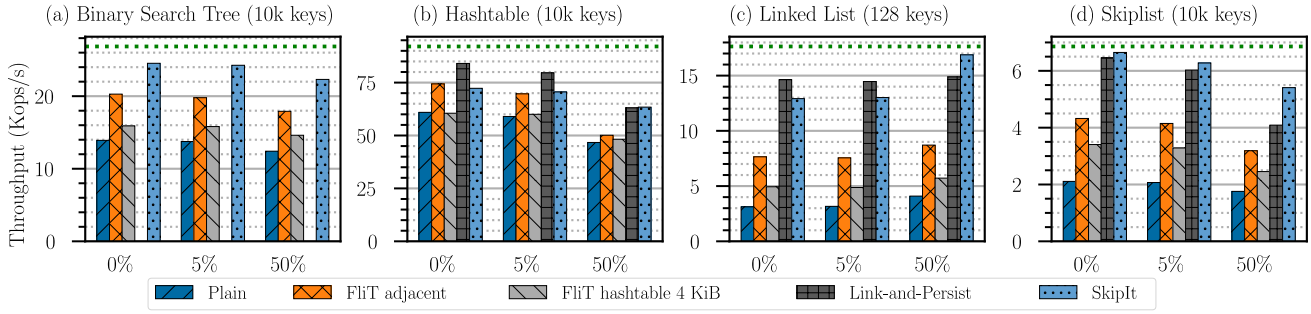

Figure 15. Throughput of varying update percentage across data structures and redundant flush optimizations.

and a skiplist [23]. We evaluate three different algorithms for maintaining persistence: the first being the **automatic** version where a writeback and a fence must be executed after every operation [36, 73], secondly the **NVTraverse** framework [27] and finally a **manual** algorithm [23].

For each scenario, we evaluate four methods to avoid redundant flushes: **FliT adjacent** and **FliT hash table** [73], **Link-and-Persist** [23, 29, 71, 81], and **Skip It**. FliT adjacent places a counter for reducing redundant flushes next to every variable. FliT hash table locates these counters in a separate hash map. Link-and-persist has a bit *within* every cacheline to indicate whether a specific cacheline has been written to peristent memory. Skip It is a combination of these methods, since it has a bit in the cacheline *metadata* to reduce redundant flushes. We additionally compare with the baseline **plain**, which provides no mechanisms to prevent redundant writebacks. All benchmarks were instantiated with 2 threads. Each update is randomly an insertion or a deletion with equal probability.

Figures 14 and 15 summarize the results of the experiments. The horizontal dark green dotted line indicates the throughput of the baseline non-persistent version. We observe that Skip It almost always outperforms FliT adjacent and FliT hash table. This is due to the extra memory that FliT consumes and the fact that Skip It implements a similar mechanism in hardware, which reduces its lookup time. It checks the cacheline's metadata, which is separated from the cacheline itself in a similar way as FliT. As Skip It is a hardware optimization invisible to programmers, it does not require any extra memory in the form of book-keeping data structures, substantially reducing cache contention and memory accesses. Skip It mostly performs comparatively or better than Link-and-Persist, except for the cases of automatic linked list and hash table. Link-and-Persist stores its bit in the 63rd bit of the data word. The data word is read anyway, so testing this bit in software and subsequently cancelling the issue of a writeback instruction may be more efficient to catch and terminate redundant writebacks. While Skip It only requires a single bit for a cacheline in L1, the writeback instruction must travel through the pipeline (including TLB misses and page table walks) to the L1 cache before it is detected as redundant and halted. However, we note that as Link-and-Persist occupies an unused bit of the address, it is not applicable for algorithms that make use of unused bits for their logic (such as the BST), and additionally all accesses to this address must first mask this occupied bit before it performs a memory operation. FliT and Skip It do not suffer from these restrictions. We observe that in Figure 14 (d), the baseline is actually lower than the most performant algorithms. Using a `CBO.FLUSH` to invalidate and writeback a cacheline may provide better performance due

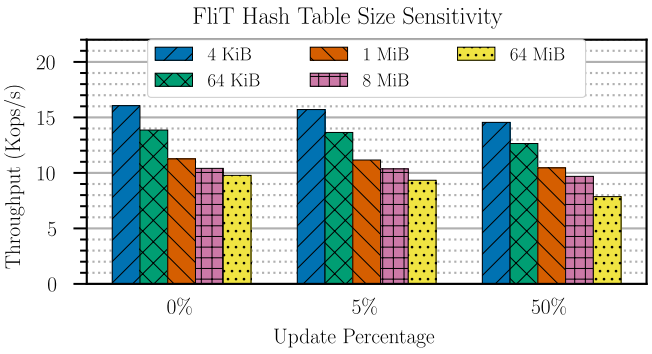

Figure 16. BST (10k keys) throughput is sensitive to FliT hashtable size.

to the memory constraints of our system. As cache line replacement constantly happens in both L1 and L2, manually freeing cache lines via flushes may offer better throughput by preempting cache replacement mechanisms.

The limited cache size of the SonicBOOM is the primary reason for the lower performance of FliT in comparison to others. The combined cache size of the Xeon CPU used in [73] to evaluate FliT is 35.75 MiB, while our SonicBOOM has 544 KiB. As FliT requires auxiliary data structures, a $\sim 67\times$ smaller cache impacts performance significantly due to constant cache contention between the evaluated data structures and FliT's metadata. This is especially apparent in the FliT hash table size, illustrated in Figure 16.

8 Related Work

There has been much research focused on building correct and efficient libraries for non-volatile main memory in which flush and fence are compulsory [70, 32, 45, 61, 22, 50, 35, 41, 27, 28, 21, 81, 74]. To avoid software-initiated writebacks, others proposed new hardware designs for NVMM. From extensive analysis of persistent memory patterns, Hands Off Persistence System (HOPS) [52] aims to improve persistent systems' efficiency by automatically tracking instruction dependencies. Nalli *et al.* propose hardware modifications (e.g. per-thread persist buffers and ISA extensions), simulated in gem5. While HOPS offers whole system persistence, this comes at the cost of significant hardware complexity.

Another line of research focuses on removing writeback instructions from the critical path [38, 19]. Joshi *et al.* [38] tracks inter- and intra-thread conflicts and postpones the persistent instructions to a later point in time, resulting in buffered-epoch or buffered-strict persistency models [56]. They introduce multiple hardware modifications and is simulated on gem5. This work is complementary to ours.

Hardware Logging (HWL) detects transactions and automatically logs all changed values, tasking the memory controller with eviction of modified log entries to NVMM [54] negating explicit writebacks and fences. Bhardwaj *et al.* [9] implemented a way for automatically logging to persistent memory using FPGAs. Braun *et al.* [13] also utilize FPGAs to build a persistence layer, *PLayer*, exploiting the cache-coherence protocol in hardware to guarantee persistence for lock-free data structures with the existence of NVMM or CXL. To support NVMM encryption, a proposed hardware extension to Asynchronous DRAM Refresh (ADR) allows for the atomic persistence of data and associated counters using writebacks efficiently [46].

The vast majority of research done on NVMM has been conducted using Intel Optane due to the provided hardware and toolchains [30]. This limits the ability of researchers to conduct exploratory research into mechanisms and architectures to use novel future memory systems. Little work with NVMM on RISC-V has been done [11] which we argue is due, in part, to the lack of architectural support for the fundamental requirements for correctness (i.e. writeback instructions). Other than support for NVMM, flush instructions are often used for security. The majority of these solutions flush microarchitectural components up to the L1 [75, 12, 44, 43] and most rely on cache coloring [40] to partition L2. Since most are implemented on in-order RISC-V CPUs, our flush instructions can allow for the implementation of these and other techniques on BOOM.

9 Conclusion

In this paper, we describe the design and implementation of two versions of writeback operations for the RISC-V BOOM core. Furthermore, we present Skip It, an optimization that avoids redundant writebacks of clean data. Implementing it on an open-source platform, a modern out-of-order RISC-V CPU, allows for further exploration and research, using and optimizing these fundamental instructions in both hardware systems and applications[4]. Our evaluation shows that it has comparable, and even preferable performance to commercial platforms. Moreover, Skip It demonstrates the value of hardware optimizations by performing better than most state-of-the-art software implementations. We hope the system enables further research into software and hardware optimizations across a broad range of problems.

Acknowledgments

The authors would like to thank the ASPLOS 2024 reviewers and our shepherd Scott Rixner for their guidance in improving the paper. We'd also like to thank the ISCA 2022 reviewers for their helpful feedback on re-approaching this project. En-Yu Jenp's thesis on Rocket Chip [37] inspired this work. Roberto Starc's thesis [66] helped us boot Linux on the cores. The authors would also like to thank Adam Turowski and the Enzian team for their help in the hardware implementation of the system.

[4] https://gitlab.inf.ethz.ch/project-openenzian/applications/risc-v

References

[1] Advanced Micro Devices. 2023. AMD EPYC 7763. https://www.amd.com/en/product/10906.

[2] Amazon Web Services. 2023. Amazon EC2 C7g Instances. https://aws.amazon.com/ec2/instance-types/c7g/.

[3] Amazon Web Services. 2023. Amazon Graviton Processor. https://aws.amazon.com/ec2/graviton/.

[4] 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*, 40, 4, 10–21. DOI: 10.1109/MM.2020.2996616.

[5] ARM. 2018. ARM architecture reference manual ARMv8. https://static.docs.arm.com/ddi0487/da/DDI0487D_a_armv8_arm.pdf.

[6] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. The Rocket Chip Generator. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, (Apr. 2016). http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[7] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, 1212–1221. DOI: 10.1145/2228360.2228584.

[8] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. 2011. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems* (HotOS'13). USENIX Association, Napa, California, 2. https://www.usenix.org/legacy/events/hotos11/tech/final_files/Bailey.pdf.

[9] Ankit Bhardwaj, Todd Thornley, Vinita Pawar, Reto Achermann, Gerd Zellweger, and Ryan Stutsman. 2022. Cache-coherent accelerators for persistent memory crash consistency. en. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*. ACM, Virtual Event, (June 2022), 37–44. ISBN: 978-1-4503-9399-7. DOI: 10.1145/3538643.3539752.

[10] David Biancolin, Sagar Karandikar, Donggyu Kim, Jack Koenig, Andrew Waterman, Jonathan Bachrach, and Krste Asanovic. 2019. FASED: FPGA-accelerated simulation and evaluation of DRAM. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (FPGA '19). Association for Computing Machinery, Seaside, CA, USA, 330–339. ISBN: 9781450361378. DOI: 10.1145/3289602.3293894.

[11] Mehrdad Biglari, Tobias Lieske, and Dietmar Fey. 2019. Reducing hibernation energy and degradation in bipolar ReRAM-based non-volatile processors. *IEEE Transactions on Nanotechnology*, 18, 657–669. DOI: 10.1109/TNANO.2019.2922363.

[12] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. 2019. MI6: secure enclaves in a speculative out-of-order processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO '52), 42–56. https://doi.org/10.1145/3352460.3358310.

[13] Richard Braun, Abishek Ramdas, Michal Friedman, and Gustavo Alonso. 2023. PLayer: expanding coherence protocol stack with a persistence layer. In *Proceedings of the 1st Workshop on Disruptive Memory Systems* (DIMES '23). Association for Computing Machinery, Koblenz, Germany, 8–15. DOI: 10.1145/3609308.3625270.

[14] Samira Briongos, Pedro Malagon, Jose M. Moya, and Thomas Eisenbarth. 2020. RELOAD+REFRESH: abusing cache replacement policies to perform stealthy cache attacks. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, (Aug. 2020), 1967–1984. ISBN: 978-1-939133-17-5. https://www.usenix.org/conference/usenixsecurity20/presentation/briongos.

[15] Christopher Celio, David A. Patterson, and Krste Asanović. 2015. The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor. Tech. rep. UCB/EECS-2015-167. EECS Department, University of California, Berkeley, (June 2015). http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html.

[16] Chen Chen, Xiaoyan Xiang, Chang Liu, Yunhai Shang, Ren Guo, Dongqi Liu, Yimin Lu, Ziyi Hao, Jiahui Luo, Zhijian Chen, Chunqiang Li, Yu Pu, Jianyi Meng, Xiaolang Yan, Yuan Xie, and Xiaoning Qi. 2020. Xuantie-910: a commercial multi-core 12-stage pipeline out-of-order 64-bit high performance RISC-V processor with vector extension : industrial product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 52–64. DOI: 10.1109/ISCA45697.2020.00016.

[17] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8, 7, (Feb. 2015), 786–797. DOI: 10.14778/2752939.2752947.

[18] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. 2022. Enzian: an open, general, CPU/FPGA platform for systems software research. In (ASPLOS '22). Association for Computing Machinery, Lausanne, Switzerland, 434–451. ISBN: 9781450392051. DOI: 10.1145/3503222.3507742.

[19] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (SOSP '09). Association for Computing Machinery, Big Sky, Montana, USA, 133–146. ISBN: 9781605587523. DOI: 10.1145/1629575.1629589.

[20] Henry Cook, Wesley Terpstra, and Yunsup Lee. 2017. Diplomatic design patterns: a TileLink case study. In *1st Workshop on Computer Architecture Research with RISC-V*, 23. https://carrv.github.io/2017/papers/cook-diplomacy-carrv2017.pdf.

[21] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures* (SPAA '18). Association for Computing Machinery, Vienna, Austria, 271–282. ISBN: 9781450357999. DOI: 10.1145/3210377.3210392.

[22] Zheng Dang, Shuibing He, Peiyi Hong, Zhenxin Li, Xuechen Zhang, Xian-He Sun, and Gang Chen. 2022. NVAlloc: rethinking heap metadata management in persistent memory allocators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '22). Association for Computing Machinery, Lausanne, Switzerland, 115–127. ISBN: 9781450392051. DOI: 10.1145/3503222.3507743.

[23] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free concurrent data structures. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, (July 2018), 373–386. ISBN: 978-1-939133-01-4. https://www.usenix.org/conference/atc18/presentation/david.

[24] Enzian Team. 2023. RISC-V on Enzian Repository. https://gitlab.inf.ethz.ch/project-openenzian/applications/risc-v.

[25] Five EmbedDev. 2023. Control and status register (csr) instructions. https://five-embeddev.com/riscv-isa-manual/latest/csr.html.

[26] Five EmbedDev. 2023. Counters. https://five-embeddev.com/riscv-isa-manual/latest/counters.html.

[27] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: in NVRAM data structures, the destination is more important than the journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2020). Association for Computing Machinery, London, UK, 377–392. ISBN: 9781450376136. DOI: 10.1145/3385412.3386031.

[28] Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. Mirror: making lock-free data structures persistent. In (PLDI 2021). Association for Computing Machinery, Virtual, Canada, 1218–1232. ISBN: 9781450383912. DOI: 10.1145/3453483.3454105.

[29] Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, and Igor Zablotchi. 2020. Efficient multi-word compare and swap. *CoRR*, abs/2008.02527. https://arxiv.org/abs/2008.02527 arXiv: 2008.02527.

[30] Jawad Haj-Yahya, Yanos Sazeides, Mohammed Alser, Efraim Rotem, and Onur Mutlu. 2020. Techniques for reducing the connected-standby energy consumption of mobile devices. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 623–636. DOI: 10.1109/HPCA47549.2020.00057.

[31] Timothy L. Harris. 2001. A pragmatic implementation of non-blocking linked-lists. In (DISC '01). Springer-Verlag, Berlin, Heidelberg, 300–314. ISBN: 3540426051.

[32] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems* (EuroSys '17). Association for Computing Machinery, Belgrade, Serbia, 468–482. ISBN: 9781450349383. DOI: 10.1145/3064176.3064204.

[33] Intel Corporation. 2022. Intel 64 and ia-32 architectures software developer manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[34] Intel Corporation. 2023. Intel Xeon Gold 6238T processor ARK. https://ark.intel.com/content/www/us/en/ark/products/192439/intel-xeon-gold-6238t-processor-30-25m-cache-1-90-ghz.html.

[35] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-atomic persistent memory updates via JUSTDO logging. *SIGARCH Comput. Archit. News*, 44, 2, (Mar. 2016), 427–442. DOI: 10.1145/2980024.2872410.

[36] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of persistent memory objects under a full-system-crash failure model. In *Distributed Computing*. Cyril Gavoille and David Ilcinkas, (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, (Sept. 2016), 313–327. ISBN: 978-3-662-53426-7. DOI: 10.1007/978-3-662-53426-7_23.

[37] En-Yu Jenp. 2022. Persistence infrastructure on RISC-V. Master Thesis. (2022). DOI: 10.3929/ethz-b-000641849.

[38] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient persist barriers for multicores. In *Proceedings of the 48th International Symposium on Microarchitecture* (MICRO-48). Association for Computing Machinery, Waikiki, Hawaii, 660–671. ISBN: 9781450340342. DOI: 10.1145/2830772.2830805.

[39] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanovic. 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)*, 29–42. DOI: 10.1109/ISCA.2018.00014.

[40] R. E. Kessler and Mark D. Hill. 1992. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.*, 338–359. https://doi.org/10.1145/138873.138876.

[41] R. Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. 2021. TIPS: making volatile index structures persistent with DRAM-NVMM tiering. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, (July 2021), 773–787. ISBN: 978-1-939133-23-6. https://www.usenix.org/conference/atc21/presentation/krishnan.

[42] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (ASPLOS 2023). Association for Computing Machinery, Vancouver, BC, Canada, 574–587. ISBN: 9781450399166. DOI: 10.1145/3575693.3578835.

[43] Tuo Li, Bradley Hopkins, and Sri Parameswaran. 2020. SIMF: single-instruction multiple-flush mechanism for processor temporal isolation. (2020). DOI: 10.48550/ARXIV.2011.10249.

[44] Tuo Li and Sri Parameswaran. 2022. FaSe: fast selective flushing to mitigate contention-based cache timing attacks. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (DAC '22). Association for Computing Machinery, San Francisco, California, 541–546. ISBN: 9781450391429. DOI: 10.1145/3489517.3530491.

[45] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, and Jinglei Ren. 2017. DudeTM: building durable transactions with decoupling for persistent memory. In *22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, (Apr. 2017), 329–343. https://www.microsoft.com/en-us/research/publication/dudetm-building-durable-transactions-decoupling-persistent-memory/.

[46] Sihang Liu, Aasheesh Kolli, Jinglei Ren, and Samira Khan. 2018. Crash consistency in encrypted non-volatile main memory systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 310–323. DOI: 10.1109/HPCA.2018.00035.

[47] Daniel Lustig. 2022. A formalization of the RISC-V memory consistency model. https://github.com/daniellustig/riscv-memory-model.

[48] Daniel Lustig. 2018. RISC-V memory model. https://www.youtube.com/watch?v=QkbWgCSAEoo.

[49] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: transparent page placement for CXL-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (ASPLOS 2023). Association for Computing Machinery, Vancouver, BC, Canada, 742–755. ISBN: 9781450399180. DOI: 10.1145/3582016.3582063.

[50] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: easy and fast persistence for volatile data structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '20). Association for Computing Machinery, Lausanne, Switzerland, 789–806. ISBN: 9781450371025. DOI: 10.1145/3373376.3378456.

[51] Microsemi. 2020. PolarFire SoC FPGA icicle kit. https://www.microsemi.com/existing-parts/parts/152514. (2020).

[52] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An analysis of persistent memory use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '17). Association for Computing Machinery, Xi'an, China, 135–148. ISBN: 9781450344654. DOI: 10.1145/3037697.3037730.

[53] Aravind Natarajan and Neeraj Mittal. 2014. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN*

Symposium on Principles and Practice of Parallel Programming (PPoPP '14). Association for Computing Machinery, Orlando, Florida, USA, 317–328. ISBN: 9781450326568. DOI: 10.1145/2555243.2555256.

[54] Matheus Almeida Ogleari, Ethan L. Miller, and Jishen Zhao. 2018. Steal but no force: efficient hardware undo+redo logging for persistent memory systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 336–349. DOI: 10.1109/HPCA.2018.00037.

[55] Mark S. Papamarcos and Janak H. Patel. 1984. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture* (ISCA '84). Association for Computing Machinery, New York, NY, USA, 348–354. ISBN: 0818605383. DOI: 10.1145/800015.808204.

[56] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 265–276. DOI: 10.1109/ISCA.2014.6853222.

[57] Nathan Pemberton and Alon Amid. 2021. FireMarshal: making HW/SW co-design reproducible and reliable. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 299–309. DOI: 10.1109/ISPASS51385.2021.00052.

[58] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2019. Persistency semantics of the Intel-x86 architecture. *Proc. ACM Program. Lang.*, 4, POPL, Article 11, (Dec. 2019), 31 pages. DOI: 10.1145/3371079.

[59] Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019. Weak persistency semantics from the ground up: formalising the persistency semantics of ARMv8 and transactional models. *Proc. ACM Program. Lang.*, 3, OOPSLA, Article 135, (Oct. 2019), 27 pages. DOI: 10.1145/3360561.

[60] RISC-V International. 2022. *RISC-V Base Cache Management Operation ISA Extensions*. https://github.com/riscv/riscv-CMOs/blob/master/specifications/cmobase-v1.0.pdf.

[61] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam H. Noh. 2017. Failure-atomic slotted paging for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '17). Association for Computing Machinery, Xi'an, China, 91–104. ISBN: 9781450344654. DOI: 10.1145/3037697.3037737.

[62] Debendra Das Sharma and Ishwar Agarwal. 2022. Compute Express Link 3.0 Standard. Tech. rep. https://www.computeexpresslink.org/_files/ugd/0c1418_a8713008916044ae9604405d10a7773b.pdf.

[63] SiFive. 2020. HiFive unmatched. https://www.sifive.com/boards/hifive-unmatched. (2020).

[64] SiFive. 2019. SiFive inclusive cache. https://github.com/sifive/block-inclusivecache-sifive. (2019).

[65] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. 2012. *Operating System Concepts*. (9th ed.). Wiley Publishing. ISBN: 1118063333.

[66] Roberto Starc. 2023. *Exploring the Microarchitectural Implications of Serverless Workloads Using RISC-V*. Master Thesis. ETH Zurich, Zurich. DOI: 10.3929/ethz-b-000610314.

[67] Wesley W Terpstra. 2017. TileLink: a free and open-source, high-performance scalable cache-coherent fabric designed for RISC-V. In *Proceedings of the 7th RISC-V Workshop*. https://youtu.be/EVITxpSEp4.

[68] Andreas Traber, Florian Zaruba, Sven Stucki, Antonio Pullini, Germain Haugou, Eric Flamand, Frank K Gurkaynak, and Luca Benini. 2016. PULPino: a small single-core RISC-V SoC. In *3rd RISC-V Workshop*. https://riscv.org/wp-content/uploads/2016/01/Wed1315-PULP-riscv3_noanim.pdf.

[69] Ventana Micro Systems Inc. 2022. Veyron V1 product information. https://www.ventanamicro.com/technology/risc-v-cpu-ip/. (2022).

[70] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS XVI). Association for Computing Machinery, Newport Beach, California, USA, 91–104. ISBN: 9781450302661. DOI: 10.1145/1950365.1950379.

[71] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 461–472. DOI: 10.1109/ICDE.2018.00049.

[72] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. 2011. The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA. Tech. rep. UCB/EECS-2011-62. EECS Department, University of California, Berkeley, (May 2011). https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

[73] Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch, and Erez Petrank. 2022. FliT: a library for simple and efficient persistent algorithms. In (PPoPP '22). Association for Computing Machinery, Seoul, Republic of Korea, 309–321. ISBN: 9781450392044. DOI: 10.1145/3503221.3508436.

[74] Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L. Scott. 2021. A fast, general system for buffered persistent data structures. In *Proceedings of the 50th International Conference on Parallel Processing* (ICPP '21) Article 73. Association for Computing Machinery, Lemont, IL, USA, 11 pages. ISBN: 9781450390682. DOI: 10.1145/3472456.3472458.

[75] Nils Wistoff, Moritz Schneider, Frank K. Gürkaynak, Gernot Heiser, and Luca Benini. 2023. Systematic prevention of on-core timing channels by full temporal partitioning. *IEEE Transactions on Computers*, 72, 05, (May 2023), 1420–1430. DOI: 10.1109/TC.2022.3212636.

[76] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, Zuojun Li, Jiawei Lin, Tong Liu, Zhigang Liu, Jiazhan Tan, Huaqiang Wang, Huizhe Wang, Kaifan Wang, Chuanqi Zhang, Fawang Zhang, Linjuan Zhang, Zifei Zhang, Yangyang Zhao, Yaoyang Zhou, Yike Zhou, Jiangrui Zou, Ye Cai, Dandan Huan, Zusong Li, Jiye Zhao, Zihao Chen, Wei He, Qiyuan Quan, Xingwu Liu, Sa Wang, Kan Shi, Ninghui Sun, and Yungang Bao. 2022. Towards developing high performance RISC-V processors using agile methodology. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 1178–1199. DOI: 10.1109/MICRO56248.2022.00080.

[77] 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. DOI: 10.1109/TVLSI.2019.2926114.

[78] Florian Zaruba, Fabian Schuiki, and Luca Benini. 2020. Manticore: a 4096-core RISC-V chiplet architecture for ultraefficient floating-point computing. *IEEE Micro*, 41, 2, 36–42. DOI: 10.1109/MM.2020.3045564.

[79] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: the 3rd generation Berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*. (May 2020). https://carrv.github.io/2020/papers/CARRV2020_paper_15_Zhao.pdf.

[80] Kan Zhong, Duo Liu, Liang, Xiao Zhu, Linbo Long, Yi Wang, and Edwin Hsing-Mean Sha. 2016. Energy-efficient in-memory paging for smartphones. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35, 10, (Oct. 2016), 1577–1590. DOI: 10.1109/TCAD.2015.2512904.

[81] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient lock-free durable sets. *Proc. ACM Program. Lang.*, 3, OOPSLA, Article 128, (Oct. 2019), 26 pages. DOI: 10.1145/3360554.