# Imprecise Store Exceptions

Siddharth Gupta*
siddharth.gupta@epfl.ch
EcoCloud, EPFL

Yuanlong Li*
yuanlong.li@epfl.ch
EcoCloud, EPFL

Qingxuan Kang
qingxuan.kang@epfl.ch
EcoCloud, EPFL

Abhishek Bhattacharjee
abhishek@cs.yale.edu
Yale University

Babak Falsafi
babak.falsafi@epfl.ch
EcoCloud, EPFL

Yunho Oh
yunho_oh@korea.ac.kr
Korea University

Mathias Payer
mathias.payer@nebelwelt.net
EcoCloud, EPFL

## ABSTRACT

Precise exceptions are a cornerstone of modern computing as they provide the abstraction of sequential instruction execution to programmers while accommodating microarchitectural optimizations. However, increasing compute capabilities in deep memory hierarchies (e.g., software event handlers, programmable accelerators) expose long exception detection latencies that forgo precise exception semantics for retired stores awaiting completion. Unfortunately, well-known post-retirement speculation mechanisms to tolerate these latencies require excessively large microarchitectural structures per core. This paper rethinks the role of architecture and OS in supporting precise exceptions. We show that instead of forcing the architecture to support precise exceptions transparently in all cases, it is preferable to employ hardware-software co-design to handle imprecise store exceptions efficiently. We develop formalism to prove that this approach complies with underlying memory consistency models and design a RISC-V prototype that passes all litmus tests, demonstrating its efficacy.

## CCS CONCEPTS

• **Computer systems organization → Architectures**.

## KEYWORDS

Memory Hierarchies, Exception Handling, Memory Consistency

**ACM Reference Format:**
Siddharth Gupta, Yuanlong Li, Qingxuan Kang, Abhishek Bhattacharjee, Babak Falsafi, Yunho Oh, and Mathias Payer. 2023. Imprecise Store Exceptions. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3579371.3589087

*Equal contribution.


*ISCA '23, June 17–21, 2023, Orlando, FL, USA*

ACM ISBN 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589087

## 1 INTRODUCTION

Precise exceptions [53] are vital to modern programming as they provide the abstraction of sequential instruction execution to programmers. Precise exceptions simplify software development while permitting various microarchitectural optimizations. A precise exception is one that is triggered exactly at the corresponding instruction after instructions earlier in program order have completed. While long-latency arithmetic and floating-point operations on co-processors [28, 30, 56] generated imprecise exceptions in the past, all exceptions in modern CPUs are precise, to the best of our knowledge. Only fatal ECC errors are handled imprecisely because they are generated in the cache/memory hierarchy.

The advent of paradigms like near/in-memory compute [19, 48], however, pose unique challenges to the traditional notion of precise exceptions. Specifically, modern CPUs assume that stores have their exceptions detected and handled before they retire (even if they have not completed). But, as this paper will show, recent proposals that push accelerator logic near/in-memory break this assumption. In effect, store exceptions may now be detected post retirement, which is problematic because retired store instructions can have pending exceptions while younger instructions can retire.

Post-retirement store exceptions present a wider problem than one may realize, and affect several recent research proposals, including: 1) accelerators that, for example, compress/decompress data requested by a load/store [50]; 2) software coherence handlers [12, 16, 26, 34, 44, 49, 60]; 3) informing memory operations [27] and software handlers for cache misses [8, 37]; and 4) virtual cache hierarchies [9, 10, 22, 33], and intermediate address space [23, 25, 61, 65] designs where virtual memory exceptions are generated in the cache/memory hierarchy.

A first approach to resolving the problems with post-retirement store exceptions may be to disable the store buffer and ensure that all retired stores have had their (even long-latency) exceptions detected and handled. But, store buffers are essential for implementation of relaxed memory consistency models [14, 38]. Store buffers boost performance by allowing long-latency stores to retire early before completion, enabling faster de-allocation of reorder buffer entries and faster retirement of younger instructions. Therefore, though simple to implement, eliding store buffers degrades performance severely.

A second approach, proposed by Qiu and Dubois [41], uses out-of-order techniques to tolerate/reduce long exception detection latencies. Their approach uses memory prefetching to detect memory-level exceptions sooner. But, as memory hierarchies grow deeper, data prefetching does not adequately hide long-latency exceptions in important classes of workloads effectively [15]. While outperforming store buffer elision at the cost of moderate complexity, prefetching does not match traditional CPU performance.

A third approach relies on post-retirement speculation techniques [7, 11, 20, 21, 42, 57]. These techniques were proposed to hide long memory access latencies and can therefore be naturally extended to handle long store exception detection latencies. But, in order to match the performance of traditional CPUs, such techniques require prohibitive amounts of buffering (tens of KBs of SRAM) per core to maintain speculation state. With the end of Moore's law, the silicon required for speculation can instead be provisioned for logic [2, 50, 63, 64] to provide higher performance and energy efficiency. Moreover, microarchitectural structures for speculation, such as register files and map tables, reside in the critical path of an out-of-order pipeline and cannot be scaled easily without impacting the cycle time and increasing power consumption.

We take a different approach and ask whether it still makes sense to provide precise exception semantics to the OS/user in all cases. Does it instead make sense to accept *imprecise store exceptions?*

We show that it is possible to co-design the hardware and OS to efficiently support imprecise exceptions for retired stores by obviating the need for speculative retirement while hiding exception-detection latency. When stores generate imprecise exceptions due to abstractions like virtual memory, then they are handled exclusively in the OS and are not visible to the user. In contrast, if the exceptions originate from other general-purpose compute units and accelerators, then they might have to be exposed to the user for correct imprecise exception handling. As imprecise store exceptions indicate that the store can only be applied to memory after successful exception handling, it is vital to understand the interaction of such stores with other memory accesses in the system in the context of the underlying memory consistency model. Fortunately, exceptions are infrequent and therefore the key challenge in addressing imprecise exceptions is guaranteeing correctness rather than optimizing design performance.

Formalism is key to enforcing that memory consistency models are correctly implemented as such hardware designs are often prone to concurrency bugs. We develop formalism to show that appropriate mechanisms in the hardware and OS can implement imprecise exceptions while preserving the underlying memory consistency model. Our formalism shows that handling normal stores together or separately from stores with imprecise exceptions gives rise to different systems. While both systems are correct, handling the stores together leads to a simpler design. Based on the formalism, we introduce the hardware-OS co-design of a new architectural interface to supply the stores with imprecise exceptions from the microarchitecture to the OS while retaining the required order so that the OS can perform the stores after successful exception handling. The interface also enables various batching optimizations for imprecise exception handling. Finally, we build a RISC-V-based prototype with Linux and evaluate standard litmus tests for memory consistency models to showcase the correctness of our design.

Overall, we make the following contributions:

- We identify that integrating compute capabilities in the deep cache/memory hierarchy can forgo precise exception semantics for retired stores.
- We demonstrate that while post-retirement speculation techniques can be used to maintain precise exceptions, they require up to 25 KBs of SRAM per core for tracking additional speculation state.
- We relax the precise exception semantics for retired stores and develop formalism to demonstrate that imprecise store exceptions correctly adhere to the underlying memory consistency models.
- We introduce hardware-software co-design to implement imprecise store exceptions efficiently while enabling batching optimizations in the OS.
- We build a RISC-V prototype using Linux to showcase the correctness of our design.

## 2 EXCEPTION HANDLING

In this section, we describe the challenges posed by post-retirement store exception detection on the notion of precise exceptions. We then discuss various "obvious" ways to address these challenges and their shortcomings, particularly in the context of relaxed memory consistency models.

### 2.1 Precise exceptions

Precise exceptions [53] are a de-facto abstraction assumed in modern CPUs as they simplify software at the cost of reasonable hardware complexity. Precise exceptions allow programmers to assume a simple sequential execution model where only one instruction executes at a time, and any exceptions are detected and handled before the corresponding instruction executes.

CPUs implementing precise exceptions require exceptions to be triggered precisely at the corresponding instruction only after older instructions completed and successfully modified process state, and before younger instructions modified process state. All exceptions must be detected and handled before instruction retirement. Microarchitecturally, exceptions are implemented by 1) executing instructions such that they modify the process state sequentially, but disabling out-of-order execution and its performance benefits, or 2) employing speculative execution to roll back the effects of any younger instructions along with the exception-generating instruction, benefiting the performance in the common case without exceptions. Importantly, this approach extends to interrupts, with the caveat that interrupts are generated asynchronously by external devices and can be triggered on any instruction.

Virtual memory is an example of a common source of exceptions as it requires every load/store instruction to perform address translation before it can be applied to the cache/memory hierarchy. If an exception (e.g., page fault) is detected during address translation, then the corresponding load/store and all the younger instructions are flushed, and the exception handler is triggered precisely. After the OS handles the exception and reschedules the process, the load/store instruction is re-executed and completes successfully.

| | | |
|---|---|---|
| Fault | Fetch | Control protection exception, Code page fault, Code-segment limit violation |
| | Decode | Invalid opcode, Device not available, Debug |
| | Execute | Divide by zero, Bound range exceeded, FP error, Alignment check, SIMD FP exception, Invalid TSS |
| | Memory | Segment not present, Stack-segment fault, Page fault, General protection fault, Virtualization exception |
| Trap | | Debug, Breakpoint, Overflow |
| Abort | | Double fault, Triple fault, Machine Check |

**Table 1: Classification of x86 exceptions [29].**

While precise exceptions are dominant today, there are cases where imprecise exceptions have been adopted. In the past, imprecise exceptions were generated by long-latency arithmetic or floating-point operations implemented on co-processors [28, 30, 56]). Today, machine checks [54] (e.g., ECC errors in the cache/memory hierarchy) are the only example of imprecise exception in modern CPUs because they are non-restartable and cause the OS to terminate the process or even crash. Finally, emerging accelerators such as GPUs [55] adopt imprecise exceptions because precise exceptions pose a dramatic performance overhead or require significantly more silicon area and power. As computing systems evolve, the cost of supporting precise exceptions and their programming implications are being gradually revisited with new constraints. This paper studies the cost of supporting precise exceptions coming from compute units embedded in the cache/memory hierarchy.

## 2.2 Long-latency exceptions can be imprecise

Today, cores are the only component in a CPU that can generate exceptions, while all other components perform simpler operations and cannot generate exceptions. Table 1 represents the x86 exceptions and their point of origin. Except for machine checks, all exceptions are generated in the fetch, decode, execute, or memory stages and caught synchronously in the reorder buffer (ROB). But with the integration of compute capabilities in the deep cache/memory hierarchies, the integrated compute units can also generate exceptions when responding to load/store instructions executed by the cores. As exceptions can originate from the cache/memory hierarchy, exception detection can even include address translation and memory latency (100s of cycles and growing).

The above problem generally appears in proposals that require software intervention when servicing memory requests, such as 1) software handling of cooperative/distributed shared memory [12, 16, 26, 34, 44, 49, 60]; 2) informing memory operations [27] and software techniques for cache-miss handling [8, 24, 37]; 3) virtual cache hierarchies [9, 10, 22, 33], or intermediate address space [23, 25, 61, 65] designs where virtual memory exceptions are generated in the cache/memory hierarchy; and 4) accelerators that can generate exceptions when executing additional functionality for memory requests performed by the cores [50]. Fortunately, this problem does not apply to accelerators invoked using an explicit request-response programming model [2, 35, 63, 64] where any generated exceptions are treated as interrupts by the cores. We further illustrate the problem using the following two detailed examples:

*Example 1* - täkō [50] is a semi-general-purpose accelerator connected to the L2 and LLC slice of each core to perform user-defined data transformations, such as compression and encryption. Users can configure täkō to compress data when evicting it from the cache and write the compressed version to memory. Upon reading the compressed data from memory on a cache miss, täkō will install the decompressed version in the cache. Such a design allows exceptions (e.g., page fault, divide-by-zero) to be generated by the accelerator when processing memory requests received from a core because täkō relies on the virtual memory abstraction to allow users to define data transformation logic using software-defined callbacks. For example, when a core executes a store instruction resulting in a cache miss, täkō will fetch and decompress data from memory and can potentially encounter a page fault. The page fault will result in a delayed notification of the exception being triggered on the corresponding store instruction.

*Example 2* - Midgard [23] is a novel virtual memory design in which the address translation is broken into two parts using an intermediate address space called Midgard. The Midgard address space maps the virtual memory areas (VMAs) from all processes and is used to index the cache hierarchy instead of physical addresses. The VMA-based (lightweight) virtual-to-Midgard address translation is required for all cache-hierarchy accesses, while the page-based (heavyweight) Midgard-to-physical address translation is required only if the cache hierarchy misses. The overall benefit of Midgard comes from the increasing cache-hierarchy capacity that leads to higher hit rates and reduces the frequency of page-based translations. In such a design, even after the VMA-based translation has succeeded for a memory request, the page-based translation can still generate exceptions due to page faults. For example, the core can execute a store instruction that passes virtual-to-Midgard address translation, misses in the cache hierarchy, detects a page fault during the Midgard-to-physical address translation, and needs to trigger an exception on the corresponding store instruction. Thus, the MMU will notify the core of the exception in a delayed manner.

Modern CPUs use aggressive optimizations [14, 38] that remove the store from the ROB (called retirement) once it becomes the oldest instruction but is yet to write its value to the L1 cache (called completion). As stores do not produce a register value for the younger instructions, they are retired to unblock the pipeline and wait for completion in the store buffer. Subsequently, if the store triggers an exception as in the examples above, the resulting exception cannot be precise because it cannot be triggered at the corresponding store instruction as it has already retired, and the pipeline might have further retired younger instructions. Fortunately, such impreciseness is limited to only store instructions as all the load instructions produce register values for the younger instructions and cannot retire before completion.

## 2.3 Forced precise exceptions kill performance

To imitate the sequential strategy [53], we can disable the store buffer optimization described above to force precise exceptions. Disabling the store buffer would ensure that all load and store

| | |
|---|---|
| Core | 16× ARM Cortex-A76 [59]<br>4-way OoO, WC, 128-entry ROB, 32-entry SB |
| TLB | L1(I,D): 48 entries, L2: 1024 entries |
| L1 Caches | 64KB 4-way L1D, 64KB 4-way L1I<br>64-byte blocks, 2 ports, 32 MSHRs<br>2-cycle latency (tag+data) |
| L2 | 1MB/tile, 16-way, 6-cycle access, non-inclusive |
| Coherence | Directory-based MESI |
| Interconnect | 4 × 4 2D mesh, 16B links, 3 cycles/hop |
| Memory | 80 cycle access latency (default) |

**Table 2: System parameters for simulation on QFlex [40].**

instructions await completion in the pipeline before they retire, forcing any exception detection to happen in the ROB itself and enabling triggering the exception handler precisely at the required instruction. However, the store buffer is the cornerstone of relaxed memory consistency models [38] such as Processor Consistency (PC) and Weak Consistency (WC), and even techniques such as end-to-end Sequential Consistency [52].

Relaxed memory consistency models (or memory models) allow memory reorderings to significantly improve single-thread performance. Such memory reorderings require the long-latency stores to be retired and put aside in the store buffer, allowing any younger instructions to retire, leading to a high pipeline throughput. Disabling the store buffer optimization also disables such reorderings, effectively reverting to Sequential Consistency (SC) and exposing the long latency of stores to the pipeline. Doing so voids the typical 20-30% single-thread performance improvement due to relaxed memory models, thus providing precise exceptions at a significant performance cost.

In this paper, we discuss the following two solutions to maintain the performance gains of relaxed memory models. 1) Using the speculation strategy [53], we demonstrate that speculation can cover the long latency of memory operations to provide precise exceptions with the performance of relaxed memory models but at a much greater silicon cost than required for ROB speculation. 2) We demonstrate that hardware-software co-design can efficiently implement imprecise handling of store exceptions where the exception handler is triggered at an unrelated instruction instead of the corresponding store instruction, thereby imitating an interrupt.

# 3 PRECISE EXCEPTIONS WITH SPECULATION

Post-retirement speculation has been shown to match and even exceed the performance of relaxed memory models at the cost of additional silicon. We now detail how these speculation techniques can be used to maintain the abstraction of precise exceptions even for long-latency store operations.

## 3.1 Post-retirement speculation

As described in subsection 2.3, relaxed memory model implementations utilize the store buffer to accommodate retired stores without blocking the pipeline. If such a store triggers an exception after retirement, then the exception handling cannot be precise as the pipeline has potentially retired younger instructions. For enforcing precise exceptions, the store must await completion and detect all exceptions before retiring, forcing the legal execution to obey SC without a store buffer.

Post-retirement speculation proposals [7, 11, 20, 21, 42, 57] applied to SC can match and even exceed the performance of relaxed memory models such as PC and WC. These proposals obtain performance benefits through speculative memory reorderings while ensuring that other cores cannot observe them by using checkpointing mechanisms to roll back the core to a legal SC state if there is interaction with other cores. As there is little interaction among cores, such proposals significantly benefit from speculative reorderings. The same reasoning applies to exceptions as they are infrequent, and speculative reorderings can provide performance benefits in the common case, while checkpoints can be used to restore the core to a legal SC state when triggering exceptions.

## 3.2 Case study: ASO

We adopt ASO [57] for imprecise exceptions. Other designs such as SC++ [21] require more silicon than ASO, while Invisifence [7] reduces silicon usage by limiting checkpoints, but also limits the overall achievable performance. In ASO, when the core is stalled due to an ordering requirement, it creates a checkpoint that allows it to ignore the ordering requirement speculatively. ASO uses a scalable store buffer to record the program order of all the speculative stores while using the L1 cache to store the latest speculatively read and written values which younger loads can then use, where the speculatively-written values in L1 are not globally visible. The speculation is successful when the core obtains the write permissions for all speculatively written blocks and atomically drains all of them to L2, making them globally visible.

The number of supported checkpoints is pre-decided as part of the ASO implementation. Each checkpoint requires a map table to record the physical registers representing the core's legal state before speculatively executing the checkpointed instruction. While in traditional ROB-contained speculation, the physical register denoting the state prior to the instruction execution is freed when the instruction retires, ASO requires keeping the physical registers until the speculation succeeds and the checkpoint is freed. For imprecise exceptions, each store miss requires a new checkpoint as the missing store can potentially trigger an exception. Once the store miss is resolved without exception, the corresponding checkpoint is merged into the previous checkpoint and the relevant physical registers are freed. Hence, the number of checkpoints reflects the number of outstanding store misses. Speculation fails if an exception is detected on a speculated store, causing the core state to be rolled back using checkpoints, followed by a precise invocation of an exception handler.

## 3.3 Quantifying the speculation state

We use cycle-accurate full-system simulation with QFlex [40] to measure the performance benefits of ASO for imprecise exceptions. Table 2 details system simulation parameters. Table 3 lists the evaluated server benchmarks from GAP [6], Tailbench [32], and Cloudsuite [17], along with the instruction mix and the IPC speedup on a WC system. As the PR, CC, and TC in the GAP benchmark

| | | Instruction mix (%) | | | | WC speedup | Speculation state requirement (KB) | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Store | Load | Sync | Others | | Baseline | 2× memory latency | 4× latency skew |
| GAP [6] | BFS | 11 | 22 | <1 | 67 | 1.53 | 14 | 14 | 17 |
| | SSSP | 3 | 22 | 1 | 74 | 1.06 | 21 | 21 | 21 |
| | BC | 25 | 25 | 0 | 50 | 3.24 | 18 | 18 | 18 |
| Tailbench [32] | Silo | 7 | 13 | 2 | 78 | 1.15 | 18 | 18 | 25 |
| | Masstree | 14 | 13 | <1 | 73 | 1.60 | 16 | 16 | 16 |
| Cloudsuite [17] | Data Caching | 11 | 24 | <1 | 65 | 1.12 | 17 | 17 | 22 |
| | Media Streaming | 9 | 13 | <1 | 78 | 1.16 | 14 | 14 | 17 |
| | Data Serving | 9 | 24 | <1 | 67 | 1.10 | 14 | 17 | 23 |

**Table 3: We list the evaluated benchmarks, their instruction mix (%), the WC speedup over SC, and the speculation state requirements (in KB) to achieve the full WC performance benefits in the baseline SC system, a system with 2× memory latency, and a system with 4× store-to-load latency skew.**

suite have <1% stores and no performance benefits from WC, we do not evaluate them further.

Table 3 shows the amount of speculation state required per core to obtain the speedup equivalent to that of WC. The speculation state includes the scalable store buffer, per-word valid and Speculatively Written (SW) bits in L1D, the Speculatively Read (SR) bits in the L1D and L2 cache, the additional physical registers required to store the legal SC state before speculatively-retired instructions, and the map tables to track the physical registers. Each entry in the scalable store buffer is 16B, while each checkpoint can require up to 32 extra physical registers (256B), resulting in a larger physical register file. Finally, each map table contains 32 logical-to-physical register mappings while storing 8-10 bits as the register index in a 256-1024 entry physical register file. Increasing the number of supported checkpoints increases the total required speculation state. Table 3 demonstrates that post-retirement speculation can indeed match the performance of corresponding WC implementations. As most of our workloads have <1% synchronization instructions, we do not achieve better performance from fence speculation. Unfortunately, the required performance gain comes at a high silicon cost of up to ~25 KB per core. Moreover, most of the required speculation state is part of the physical register file, which is a critical structure in the pipeline and cannot be scaled easily.

We also perform additional studies to quantify the impact of hardware scaling trends on the amount of speculation state required. To this end, we quantify the impact of memory latency and the latency skew between stores and loads on the required speculation state. As the memory capacity in the system scales, the average memory latency becomes higher because of frequent remote memory accesses across sockets and due to denser memory technologies such as persistent memory. Moreover, modern systems feature multiple sockets/chiplets per CPU [58], which can lead to remote cache accesses from far-off sockets. In such a system, cache coherence protocols require extra hops for servicing stores than they need for servicing loads because stores require invalidating all copies of the block, which might reside in far sockets/chiplets.

Table 3 depicts two additional systems with 2x the baseline memory latency, and 4x the baseline store-vs-load latency skew respectively. As demonstrated, a system with 2x the memory latency requires about the same amount of speculation state to reach the WC speedup as the baseline system. The reason is that increasing the memory latency affects both the loads and stores, and as loads typically outnumber the stores, they quickly become the performance bottleneck. In contrast, the required speculation state can increase considerably in a system where the stores take 4x longer latency to complete than the loads. The reason is that as the skew between store and load latencies increases, the pending stores in the store buffer take more time and prevent further stores from retiring, thus blocking the pipeline. Increasing the time required to drain the store buffer also has performance side effects for other subsystems, such as the *tlbi* instruction for performing TLB shootdowns. Overall, while speculation can indeed provide the performance of WC, it requires a significant amount of SRAM resources that are not trivially obtained in the post-Moore era.

# 4 IMPRECISE STORE EXCEPTIONS

Maintaining precise exceptions with SC either leads to significant performance degradation or requires post-retirement speculation mechanisms to provide WC performance using tens of KBs of expensive per-core speculation state. Instead, we propose a hardware-software co-design to implement PC/WC systems with the relaxed semantics of imprecise store exceptions. In this section, we provide the formalism for memory models with imprecise store exceptions and demonstrate that with appropriate hardware and OS support, imprecise store exceptions are compatible with both PC and WC.

## 4.1 Brief description

Similar to modern CPUs, we require that the stores are retired and put in the store buffer to await completion. Subsequently, if they trigger an exception, the exception handler must be triggered imprecisely on the oldest instruction in the ROB. However, the faulting stores (i.e., stores that trigger exceptions) present in the store buffer can neither be drained to the cache hierarchy nor rolled back to be re-executed later as they have already retired and younger instructions have further modified the architectural state. With exception handling latencies of several $\mu s$ (e.g., lazy memory allocation) to tens of *ms* (e.g., demand paging), the stores cannot

| Notation | Definition |
|---|---|
| L(A) | Load latest value from address A |
| S(A), S(A, D) | Store data D to address A |
| $S_{OS}$(A), $S_{OS}$(A, D) | OS stores data D at address A |
| F | Fence as a memory ordering primitive |
| X <p Y | Operation X happens before operation Y in program order on the same core |
| X <m Y | Operation X happens before operation Y in the global memory order |
| PUT(S(A)) | Send S(A) to the architectural interface. |
| GET | Retrieve one faulting store from the architectural interface |
| DETECT | Detect an exception |
| RESOLVE | Resolve the exception and resume execution |
| $MAX_{<m}$ ($\{\}_{S(A)}$) | Return the latest value in memory order from the set of stores to address A |

**Table 4: Memory consistency formalism notations [38].**

stay in the store buffer because they will eventually clog the store buffer and prevent the core from executing younger instructions.

We create a new architectural interface to supply faulting stores along with their address, data, byte mask, and the accelerator-specific exception code to the OS so that we can free up the microarchitectural resources occupied by the store. When triggering an imprecise exception handler, the OS first reads the faulting stores using the architectural interface and then resolves the exception. If the exception is recoverable (e.g., page fault), the OS applies the obtained faulting stores to the corresponding addresses, and the program resumes execution. If the exception is irrecoverable (e.g., segmentation fault), the faulting stores are discarded, and the program is terminated.

As the OS may take up to tens of *ms* to resolve the exception and apply the faulting store, the store is effectively reordered after younger operations in the global memory order, which might seem to violate the underlying memory model. While previous research proposals [36, 46] studied the interaction between memory operations executed by the core and MMUs, there are no studies on such OS-induced reorderings. We claim that we can make these reorderings transparent to user programs with a formally-defined memory model incorporating store exceptions and a hardware-software co-design that conforms to required constraints.

## 4.2 Formal definition of memory models

We first describe the standard formalism [38] of PC and WC using notations defined in Table 4. We study PC and WC because they are the prevalent models in ISAs today (x86, AMD, ARM, RISC-V). We use PC to represent Total Store Order (TSO) as they are identical in modern cache-coherent systems [1].

PC relaxes the store-to-load ordering and is formally defined with the following rules:

$$L(A) <_p S(B) \implies L(A) <_m S(B)$$

$$L(A) <_p L(B) \implies L(A) <_m L(B)$$

$$S(A) <_p S(B) \implies S(A) <_m S(B)$$

$$S(A) <_p F <_p L(B) \implies S(A) <_m F <_m L(B)$$

$$L(A) = MAX_{<m} (\{S(A, D) \mid S(A, D) <_m/<_p L(A)\})$$

WC relaxes various other orderings present in PC and is formally defined with the following rules:

$$L(A)/S(A) <_p F \implies L(A)/S(A) <_m F$$

$$F <_p L(A)/S(A) \implies F <_m L(A)/S(A)$$

$$L(A) <_p L'(A)/S(A) \implies L(A) <_m L'(A)/S(A)$$

$$S(A) <_p S'(A) \implies S(A) <_m S'(A)$$

$$L(A) = MAX_{<m} (\{S(A, D) \mid S(A, D) <_m/<_p L(A)\})$$

where L'(A)/S'(A) depicts another load/store to address A. Overall, WC relaxes all orderings except the ones involving fences and memory operations to the same address.

We further require additional operations to handle imprecise store exceptions. Assuming that S(A) triggers an exception, the DETECT operation indicates the detection of the exception. Once the exception is detected, then S(A) should be supplied to the architectural interface using PUT(S(A)) operation, and consequently, the OS will read the faulty store using GET operation and will apply the faulting store to the address A using $S_{OS}$(A) operation. Finally, the OS can finish the exception handling and resume the program execution using the RESOLVE operation. Overall, these new operations strictly happen in the global memory order as:

$$DETECT <_m PUT(S(A)) <_m GET <_m S_{OS}(A) <_m RESOLVE$$

## 4.3 Observing the memory order

Independent of the underlying memory model and the presence of imprecise exceptions, programs can infer the order among memory operations only by detecting value changes at the corresponding addresses. Assume that an application applies two stores, S(A,1) and S(B,1), to two zero-initialized memory locations, A and B. To detect the memory order between S(A) and S(B), the program requires two *observer loads* L(A) and L(B) that should be executed on a different core to detect the change in values at addresses A and B. The only way that the application can infer S(A,1) <m S(B,1) is if L(A) <m L(B), L(A) reads 1, and L(B) reads 0, corresponding to the execution S(A,1) <m L(A) <m L(B) <m S(B,1). Note that these requirements create a total order among the four operations.

Similarly, the only way to infer S(B,1) <m S(A,1) is if L(B) <m L(A), L(A) reads 0, and L(B) reads 1, corresponding to the execution S(B,1) <m L(B) <m L(A) <m S(A,1). If any of the above requirements are not fulfilled, then the order between S(A,1) and S(B,1) cannot be inferred solely based on the value read by observer loads because there is no dependency chain among the stores and loads. In summary, the order among memory operations can only be inferred by the composition of preserved program orders [3] (memory orders enforced from program order based on the memory model, e.g., program order between two stores in PC) and specific change in value that loads can detect. Consequently, programs cannot infer the memory order if the required value changes are not detectable.

Using the above rules, certain combinations of detectable value changes can lead to violations of the memory model. Consider the message-passing litmus test [38] shown in Figure 1, where Core 0

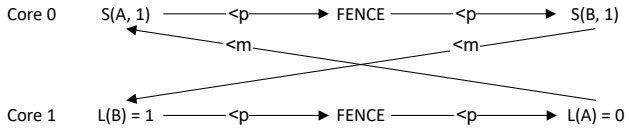



**Figure 1: Violation in the message-passing litmus test.**

communicates B to Core 1 by first setting A to indicate the ready status of B. For simplicity, we explicitly insert two fences between two stores and two loads to make WC identical to PC. Out of four possible results, only the execution with L(B) reading 1 and L(A) reading 0 is prohibited because it indicates that both S(A, 1) <m L(A) and L(A) <m S(A, 1) hold simultaneously, which is impossible. The basis of our formalism is that the applications cannot detect any violations due to faulting stores because the required value changes to infer the memory ordering are either presented in the correct order or are not detectable.

## 4.4 Contract among the cores, interface, and OS

Next, we describe the contract required between the cores, the architectural interface, and the OS to adhere to the underlying memory models, as shown in Table 5. Imprecise exception handling requires an architectural interface to supply the faulting store to the OS, and the interface is required to guarantee that the OS retrieves the faulting stores in the same order as the core sends them. While we do not require a total order among the faulting stores from all cores, we require enforcing a per-core order for PC and do not require any order for WC.

When using the interface to supply faulting stores, the cores must ensure that the stores are supplied in the order the underlying memory model prescribes. In PC, the faulting stores in the store buffer should be supplied to the interface in the FIFO order of the store buffer so that the OS can apply all the stores to memory in the required order after successful exception handling. However, in WC, as there is no order required among stores in the store buffer except for stores to the same address that are already coalesced, the order of supplying the faulting stores is irrelevant.

After successful exception handling, the OS retrieves the faulting stores using the interface and applies them to their target addresses. The order in which the faulting stores are applied is critical for the correctness of the memory model, and requires the OS to obey the following constraints. First, similar to precise exceptions, the program or thread that triggered the imprecise exception can only resume after the exception has been successfully handled. Second, all the retrieved faulting stores must be applied to their target addresses for the imprecise exception handling to be complete. Third and last, the OS must ensure that the faulting stores are applied to memory in the same order as retrieved from the interface. The last rule applies only to PC which requires a strict order among per-core stores. In the case of WC, the OS does not need to enforce any order among stores as the memory model does not mandate it.

The above constraints also imply that if there is a precise exception on a load or if an imprecise exception is pinned on an atomic or a fence instruction because it blocks the ROB waiting for the store buffer to drain where a store in the store buffer generates the imprecise exception, then the load/atomic/fence instruction will be re-executed only after successful exception handling indicated by RESOLVE <m L(A)/Atomic/F . The overall intuition behind these constraints is to ensure that the microarchitecture communicates the order required for obeying the underlying memory model to the OS, which then applies the faulting stores in the required order to complete exception handling successfully before resuming the program execution.

| Component | Requirements for PC |
|---|---|
| Cores | Supply faulting stores to the interface in the serial order dictated by the store buffer |
| Interface | Supply faulting stores to the OS in the same order as received from the core |
| OS | 1) Program resumes only after exception handling<br>2) Apply all faulting stores during handling<br>3) Apply the faulting stores in the interface order |

**Table 5: The contract among the cores, interface, and OS.**

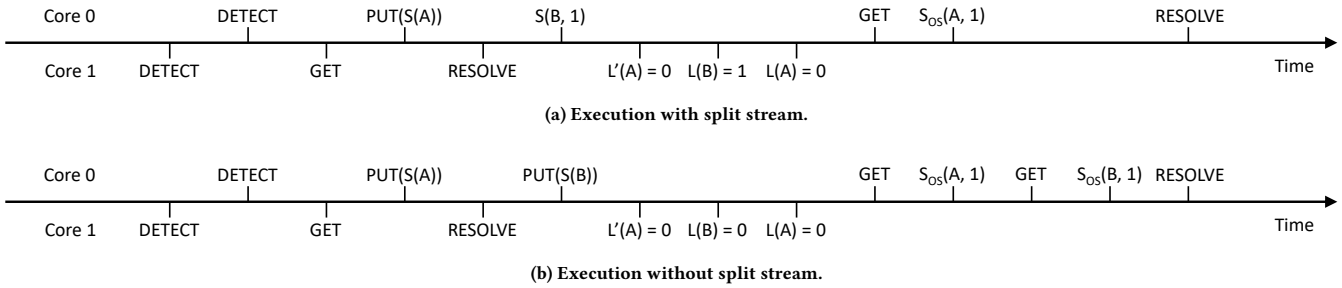
## 4.5 Formalism with split stream

When handling imprecise store exceptions, there are two approaches to treat the non-faulting stores present along with faulting stores in the store buffer. The non-faulting stores can be directly written to memory or supplied to the interface along with faulting stores. The first approach results in a formalism that treats non-faulting stores and faulting stores as two separate streams of operations, while the second approach results in all the stores being treated as the same stream. We provide the formalism for both approaches and describe their differences.

When a faulting store is in the store buffer, the other non-faulting stores can still be drained to memory. We call this case split stream because the stream of faulting stores is treated differently from that of non-faulting stores. In PC, both streams should obey program order, while in WC, the order of stores in each stream is irrelevant. In both models, the non-faulting stores will be drained to memory immediately, while the faulting stores will be applied later by the OS, potentially breaking the required global memory order among all the stores. We show this effect using the following formalism. Assume that S(A) is faulting so that it is supplied to the interface, while another non-faulting S(B) that happens after S(A) in program order is drained to memory as usual. Eventually, an exception handler is triggered, which retrieves S(A) from the interface and applies it to memory, resolving the exception. We can formally represent the scenario as follows:

$$S(A) <_p S(B) \implies \text{DETECT} <_m \text{PUT}(S(A)) <_m S(B) <_m \text{GET} <_m S_{OS}(A) <_m \text{RESOLVE}$$

As shown, S(B) can appear before $S_{OS}(A)$ in memory order and violate the memory ordering requirements for PC. To be compatible with PC, the hardware and software need to ensure that observer loads cannot observe this violation, which requires that $S_{OS}(A) <_m$ RESOLVE $<_m$ L(A) so that L(A) cannot detect the value change on A that leads to the violation. While it might seem like this condition is easy to satisfy when address A is faulting, a subtle race




**(a) Execution with split stream.**

**(b) Execution without split stream.**

**Figure 2: Race condition between the GET operation on Core 1 and the PUT(S(A)) operation on Core 0.**

condition exists in this case. Consider a program where Core 0 executes S(A,1) <p S(B,1), and Core 1 executes L'(A) <p L(B) <p L(A). Assume that both A and B are zero-initialized, and the accesses to address A will result in exceptions at both S(A) and L(A), corresponding to the following two concurrent executions:

Core 0: DETECT <m PUT(S(A)) <m S(B) <m GET <m $S_{OS}$(A) <m RESOLVE

Core 1: DETECT <m GET <m RESOLVE <m L'(A) <m L(B) <m L(A)

There can be a race between PUT(S(A)) on Core 0 and GET on Core 1 such that at the time GET completes, it cannot see PUT(S(A)). As $S_{OS}$(A) might not be applied before Core 1 performs its RESOLVE, it can result in an execution where L(B) reads 1 and L(A) reads 0, as shown in Figure 2a. While such execution is legal in WC, it violates PC because Core 1 can infer that S(B) has taken place, but S(A) has not, even though it comes earlier in program order. For the split stream formalism to work in real designs, the hardware and OS should together ensure that the core executes PUT(S(A)) before the OS performs the final GET to avoid any data races. Such a design would require a barrier or synchronization between the hardware and software so that no further PUT(S(A)) can occur after the GET. Though possible, building such designs is difficult because of the complexity and performance overhead of implementing such barrier or synchronization mechanisms.

## 4.6 Formalism without split stream

We now describe the second approach, where the faulting and non-faulting stores are treated as the same stream. In PC, stores in the store buffer should be applied to memory in program order. In the presence of faulting stores, instead of sending younger non-faulting stores to memory as another stream, they are supplied to the architectural interface along with the faulting stores. The faulting and younger non-faulting stores are still supplied to the interface in the FIFO order of the store buffer, allowing the OS to retrieve and apply them in the correct program order when handling the imprecise store exception. The intuition behind this approach is that if the faulting and any younger non-faulting stores are supplied to the OS and applied by the OS in their original program order, then there are no potential PC violations that the OS needs to hide as in the previous case. Assuming that S(A) is faulting while another younger S(B) is not, the core will supply both stores to the architectural interface in the program order to be applied by the OS. We can formally specify this scenario as follows:

$$\begin{aligned} S(A) <_p S(B) \implies & \text{DETECT} <_m \text{PUT}(S(A)) <_m \\ & \text{PUT}(S(B)) <_m \text{GET} <_m S_{OS}(A) <_m \\ & \text{GET} <_m S_{OS}(B) <_m \text{RESOLVE} \end{aligned}$$

To prove that S(A) <p S(B) $\implies$ S(A) <m S(B), we consider the following four cases:

(1) Both S(A) and S(B) are not faulting,
(2) Only S(B) is faulting,
(3) Both S(A) and S(B) are faulting,
(4) Only S(A) is faulting.

Assume that S(B) is in the store buffer when S(A) is drained to memory or supplied to the architectural interface.

Case 1 is the original PC case. The store buffer drains both S(A) and S(B) to memory in their program order.

In case 2, the store buffer drains S(A) to memory and supplies S(B) to the interface. The OS then retrieves S(B) from the interface and applies it. This case represents the following execution:

$$\begin{aligned} S(A) <_p S(B) &\implies S(A) <_m \text{DETECT} <_m \text{PUT}(S(B)) <_m \\ &\quad \text{GET} <_m S_{OS}(B) <_m \text{RESOLVE} \\ &\implies S(A) <_m S_{OS}(B) \end{aligned}$$

In case 3, the store buffer supplies both S(A) and S(B) to the interface. The OS then retrieves both stores from the interface and applies them in the retrieved order. This case represents the following execution:

$$\begin{aligned} S(A) <_p S(B) &\implies \text{DETECT} <_m \text{PUT}(S(A)) <_m \\ &\quad \text{PUT}(S(B)) <_m \text{GET} <_m S_{OS}(A) <_m \\ &\quad \text{GET} <_m S_{OS}(B) <_m \text{RESOLVE} \\ &\implies S_{OS}(A) <_m S_{OS}(B) \end{aligned}$$

Case 4 is same as case 3 because S(B) is supplied to and applied by the OS as it follows the faulting S(A).

If S(B) is not in the store buffer because it has not retired, then S(A) can either be applied to memory (case 1 and 2), or trigger an exception (case 3 and 4) where the OS applies $S_{OS}$(A) to memory and executes the RESOLVE operation. In all cases, S(B) is guaranteed to retire only after S(A) or $S_{OS}$(A) has been already applied to memory.

**Proof 1: Store-store ordering rule of PC.**

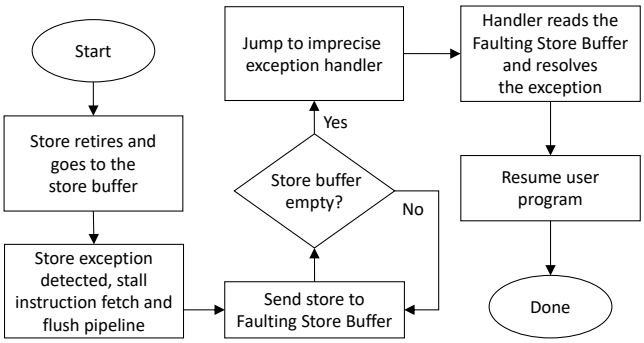



**Figure 3: Imprecise store exception handling flow.**

As shown, the OS always maintains the correct order between $S_{OS}(A)$ and $S_{OS}(B)$ that comes from the program order of S(A) and S(B). Considering the race condition discussed in the previous subsection, even though the race between PUT(S(A)) and GET still exists, as long as L(B) reads 1, which indicates that $S_{OS}(B) <_m L(B)$, L(A) must also read 1 because the OS enforces the order between $S_{OS}(A)$ and $S_{OS}(B)$ such that $S_{OS}(A) <_m S_{OS}(B) <_m L(B) <_m L(A)$, as shown in Figure 2b. On the other hand, if L(B) reads 0, then L(A) can either read 0 or 1, both of which are not PC violations. We can formally prove that the same-stream approach obeys all five rules for PC defined in subsection 4.2. Due to space limitations, we only show the proof for the store-store rule $S(A) <_p S(B) \implies S(A) <_m S(B)$ in Proof 1. Other rules can be proved in a similar manner.

# 5 DESIGN

This section presents a detailed hardware-software co-design of the architectural interface to supply faulting stores to the OS. We assume a generic multicore system as shown in Figure 4. We also assume that the imprecise store exceptions are generated by a generic hardware component situated in the cache hierarchy and away from the cores.

## 5.1 Exception detection

Figure 3 depicts the detection and handling flow for imprecise store exceptions. When the store buffer receives a retired store from the ROB, it sends a memory request for the store to the cache hierarchy. In the case of a faulting store, the memory request cannot find the required cache block, and eventually reaches the required hardware component which generates an exception and sends a response with an embedded error code back to the requesting core. The response message backtracks through the cache hierarchy while freeing the occupied resources, such as MSHRs allocated for the request. The response is received by the L1-D, which then relays it to the corresponding store buffer entry, completing the detection of the imprecise store exception. Then, the store buffer drains the store address, data, byte mask, and the error code into the *Faulting Store Buffer* (FSB), which is the backing storage of the architectural interface that temporarily accommodates faulting stores before supplying them to the OS, as specified in subsection 4.4.

## 5.2 Faulting Store Buffer and Controller

Similar to the x86 virtualization mechanism where the core drains the architectural state of the guest virtual machine to the Virtual Machine Control Structure upon #VMExit [29], the store buffer drains faulting stores to the FSB for imprecise store exceptions. The FSB is a per-core ring buffer located in the main memory with a head and tail pointer, as shown in Figure 4. The FSB is similar to the ring buffers used in OS/virtualization like io_uring [13], XEN [5], and VirtIO [47], or hardware interfaces/protocols like NVMe [39] and RDMA [43] that facilitate uni-directional order-preserving communication. The order among faulting stores is encoded in their relative positions in the FSB. We use a *Faulting Store Buffer Controller* (FSBC) to control the order in which the faulting stores are written into the FSB. Each core has a private FSBC co-located with the store buffer, as shown in Figure 4. After detecting an exception, the store buffer sends the faulting stores to the FSBC in the order mandated by the memory model. The FSBC then writes them to the tail pointer position of the FSB. After each store draining completes, the FSBC increments the tail pointer and sends a completion response back to the store buffer.

The FSBC is exposed to the OS using four per-core system registers in the ISA: *base*, *mask*, *head pointer*, and *tail pointer*. The OS configures the base and mask to specify the address of the FSB in memory, which is allocated by the OS and is not visible to the application. The tail pointer is written by the FSBC and read by the OS, specifying the position to drain the next faulting store. The head pointer is written by the OS and read by the FSBC, specifying the position of the oldest faulting store in the FSB. The OS can retrieve the oldest faulting store by reading the entry at the head pointer. The OS increments the head pointer to mark the faulting store as read and retrieves the next faulting store (if present). Once the head pointer matches the tail pointer, all faulting stores have been handled. The FSB is sized according to the number of store buffer entries, representing the maximum number of already retired stores that might need to be drained. Our proposal does not require any further changes to the existing core microarchitecture. The load/store queues and store buffer can still have their original design and capacity. In the common case when there are no imprecise store exceptions, the core works traditionally with the store buffer providing WC performance benefits over SC. The control and data paths of FSBC are activated only after the store buffer detects an imprecise store exception.

## 5.3 Exception handling

On detecting an imprecise store exception, the core stops the instruction fetch and drains all unfinished stores present in the store buffer to the private per-core FSB without requiring any special synchronization or cache coherence transactions. After draining each store, the FSBC sends a completion response to the store buffer which then discards the corresponding entry. Once all entries are drained, the FSBC triggers an imprecise exception which is attached to the oldest uncommitted instruction in the ROB, resembling an interrupt. Consequently, all the uncommitted instructions, including the one with the attached exception, are flushed, and the core jumps to the corresponding exception handler.

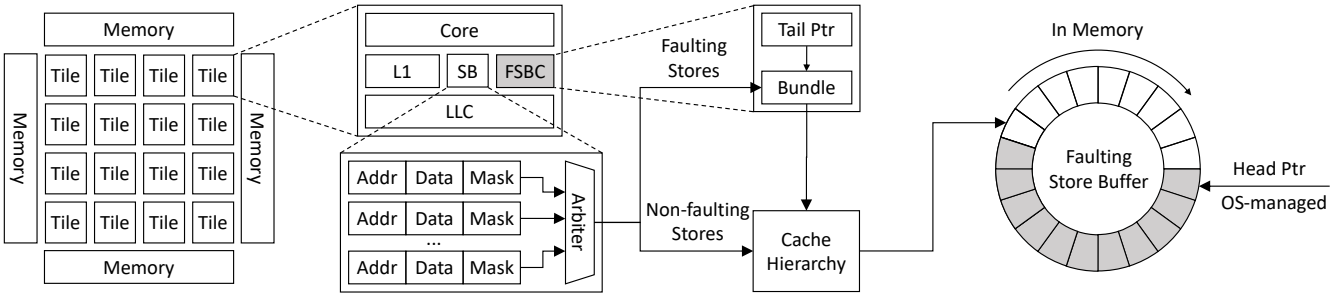



**Figure 4: Modifications to handle imprecise exceptions in a generic multicore system.**

The store buffer dictates the order of stores and associated imprecise exceptions to obey the underlying memory model. In PC, after the store buffer detects an imprecise store exception on a faulting store, all the younger uncompleted stores are drained to the FSB in program order, even if the coherence requests for those stores are still ongoing and potentially result in more imprecise exceptions. Similarly, before handling any precise exception detected in the pipeline, the core drains the store buffer to detect potential imprecise store exceptions. If such an exception is detected on an older store, the core flushes the pipeline, forgoes the precise exception, and handles the imprecise exception instead. Only after the successful handling of imprecise store exceptions, the instruction that triggered the precise exception is re-executed and re-generates the precise exception again. Overall, the design ensures that all exceptions are handled in program order.

The OS can correctly identify an imprecise store exception by the dedicated exception code reserved in the ISA. In the exception handler, the OS first copies all faulting stores from the FSB to an OS-managed data structure and then starts the traditional exception handling. When the handler executes, the effects of all the committed user instructions are present in registers, memory, or the FSB, but the exact architectural state of the faulting store is lost. To the best of our knowledge, typical exception handlers do not examine the architectural state corresponding to past retired instructions but receive the necessary information as part of the exception. Similarly, for imprecise exceptions, the handler retrieves the necessary information from the FSB, such as the faulting store's address and data, and handles the exception. In case of recoverable exceptions, the OS resolves the exception, applies the faulting stores to memory in the same order as they were retrieved from the FSB, and then resumes the responsible application. In case of irrecoverable exceptions, the OS terminates the responsible application and the faulting stores are discarded.

While exception handling is serialized in time, interrupts can be detected concurrently with imprecise store exceptions. In current systems, handling interrupts does not require draining the store buffer, which makes it possible for imprecise store exceptions to be detected while the interrupt handler is executing. We rely on the *Interrupt Enable* (IE) bit defined in the ISA to prevent the exceptions from obstructing the interrupt handler. The IE bit is automatically set when triggering interrupt and imprecise store exception handlers and by the OS when it enters a non-interruptable critical section. The OS clears the IE bit when it exits from the critical section or is ready to handle new interrupts or imprecise store exceptions. By manipulating the IE bit, the core and OS can serialize the handling of interrupts/imprecise store exceptions and the execution of critical sections. Moreover, pending/masked imprecise store exceptions can stop the OS from resuming user applications because the exception cannot be masked in user mode as the IE bit is hard-wired to zero and not effective in user mode.

In contrast to precise exceptions, an imprecise store exception can correspond to multiple faulting stores, allowing the OS to handle them in batches. Consider the case where multiple faulting stores generate major page faults due to demand paging. In the traditional case, each major page fault triggers a precise exception to let the OS schedule an IO request to load the corresponding page back. The next page fault can be triggered only after the last IO request is done and the application is resumed, forcing all IO requests to take place sequentially. In contrast, within a single invocation of the imprecise store exception handler, the OS can schedule multiple IO requests for all the faulting stores covered by the exception, effectively overlapping IO latencies and improving IO throughput. The batching effect also helps reduce the overhead of invoking the handler, where the context switch, exception dispatch, and other miscellaneous costs are only paid once per each handler invocation instead of per faulting store.

## 5.4 OS requirements

As the FSBC controls the draining of the faulting stores into the FSB, the OS should always pin the data pages allocated to FSBs in memory, ensuring no page faults. As the FSB is sized according to the store buffer, the OS only needs to reserve a few 4K pages per core. As a minimal requirement, the OS should ensure that the imprecise store exception handler does not trigger further imprecise exceptions. Otherwise, the handler will need to support recursive exceptions, which significantly complicates the system design. Such recursion is not supported in traditional systems as well.

In cases where the OS must send some data to the accelerator (e.g., when invoking `copy_to_user` where the user buffer is allocated from the accelerator), the kernel can also generate imprecise store exceptions. In such cases, the OS can utilize fence instructions to fully contain imprecise store exceptions and limit them from affecting other parts of the kernel. For example, after invoking `copy_to_user`, the OS can issue a fence instruction to ensure that any potential OS imprecise exceptions are properly reported and handled. The OS can enhance any function that may potentially

generate imprecise exceptions in this way. As the OS does not directly use accelerators, we expect only a few OS functions to require such enhancement. Similarly, the OS should issue a fence before switching to the user mode to avoid OS imprecise exceptions affecting user applications.

# 6 PROTOTYPE AND EVALUATION

In this section, we introduce our full-system prototype for imprecise store exceptions and evaluate our prototype's silicon overhead, functional correctness, and performance to demonstrate the feasibility and benefit of handling store exceptions imprecisely.

## 6.1 Prototype overview

We use XiangShan [62], an open-source, high-performance, Out-of-Order RISC-V CPU written in Chisel [4] to build our prototype. XiangShan implements the RISC-V Weak Memory Ordering (RVWMO) [45] as its memory model. We extend XiangShan's microarchitecture to support detecting and handling imprecise store exceptions as described in subsection 5.2. We also port our prototype to AWS cloud FPGAs for fast simulation using FireSim [31]. Due to the limited capacity of cloud FPGAs, our prototype currently only supports two minimal XiangShan cores. We use Linux 5.15 as the OS and add various device drivers and handlers to support injecting and handling imprecise store exceptions as specified in subsection 5.3.

We synthesize and implement our prototype using Vivado 2020.2. In the routed design, FSBC consumes 354 CLB LUTs and 763 CLB registers per core, corresponding to only 0.12% and 0.48% of the total core consumption. As FSBC is tightly integrated into the core, some core modules (e.g., the *CSR* module that manages system registers and exceptions) are also changed accordingly. The silicon overhead of these extra modifications is also minimal.

## 6.2 Error injection and handling

We create a hardware component *EInject* for error/poison injection to model imprecise store exceptions that accelerators might generate. EInject monitors each non-coherent TileLink-UL [51] transaction between the LLC and memory. For transactions whose addresses lie in the memory region reserved by EInject, it looks up a bitmap to check whether the targeting physical page is marked as faulting. If so, EInject terminates the transaction and generates a response to the LLC with a bus error by setting the *denied* bit.

As an MMIO device, EInject exposes two MMIO registers, *set* and *clr*, to the software to manage the bitmap. Writing an address A to these two registers sets or clears the bit corresponding to the 4KB page of that address in the bitmap. Thus, the software can dynamically inject faults into the system by setting some pages as faulting and handle these faults by setting the pages back to non-faulting. We add a device driver in Linux to allow user-level applications to *mmap* the memory reserved by EInject and control the errors on the mapped pages by *ioctl*.

We implement a minimal OS handler for resolving imprecise store exceptions. Since EInject is the only source of imprecise exceptions in the system, the handler marks the corresponding page as non-faulting through the EInject interface for each faulting store, performs the store using normal store instructions, and then increments the head pointer. The handler continues this action until the head pointer catches the tail pointer, thus indicating that all the faulting stores have been served.

| Ordering relation | Explanation | Cases covered |
|---|---|---|
| Dependencies | Register dependencies for addr, data, and ctrl | 2366 |
| Program order (same location) | Rd-Rd or Wr-Wr to same the address from the same core | 368 |
| Preserved program order | Instruction pairs maintained in program order (Atomic, LR/SC) | 733 |
| External read-from order | Wr-Rd to the same address from different cores | 1544 |
| Internal read-from order | Wr-Rd to the same address from the same cores | 1304 |
| Coherence order | Wr-Wr total order to the same address | 747 |
| From-read order | Rd-Wr to the same address | 976 |
| Barriers | Ordering imposed by barriers | 1581 |

**Table 6: Ordering rules [3] covered in litmus tests.**

## 6.3 Functionality correctness

We use the litmus tests [18] in the RISC-V specification [45] to verify that our prototype does not violate RVWMO even with imprecise store exceptions. The test suite targets various ordering relations and constraints specified in RVWMO, as shown in Table 6. We modify each test to allocate the memory for consistency check from the EInject regions. Before running each test, we also intercept the *main* function to mark the allocated memory as faulting, as described in subsection 6.2, to inject bus errors on all load, store, and atomic instructions, which generate many precise and imprecise exceptions that are silently handled by the minimal handler in Linux. We pick all (1600) 2-core litmus tests that can run successfully on QEMU and run them in a large batch on our prototype system. Our prototype does not produce any RVWMO violation for all the litmus tests. Overall, we empirically prove that our prototype does not break the underlying memory model.

## 6.4 Performance: microbenchmark

We use a microbenchmark with injected imprecise store exceptions to evaluate the performance overhead incurred by both hardware and software of our prototype. The microbenchmark runs multiple iterations of a loop that applies 10 K stores to a 512 MB array. To stress the imprecise store exception handling, at the start of each iteration, the microbenchmark picks a random subset of 4KB pages and marks them as faulting using the EInject interface. The resulting imprecise store exceptions are then transparently handled by the

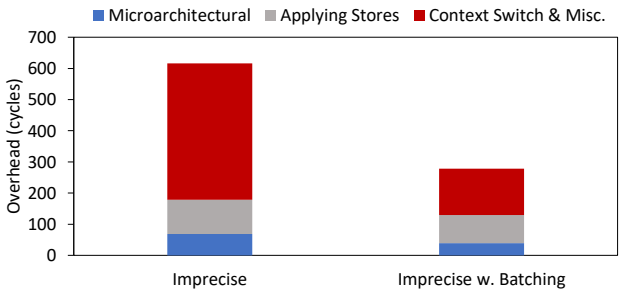



**Figure 5: Overhead breakdown of imprecise exceptions with and without batching.**

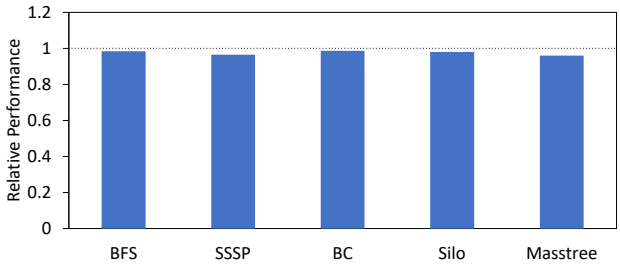



**Figure 6: Relative performance of GAP and Tailbench workloads with imprecise store exceptions.**

minimal handler. The microbenchmark uses RISC-V's performance monitor counters to read clock cycle numbers from the hardware.

Figure 5 shows the breakdown of the overhead of handling a single faulting store. The overhead consists of three parts: 1) microarchitectural overhead, which contains the time spent on draining faulting stores to the FSB and ROB/pipeline flush, 2) OS overhead of applying the faulting store, and 3) other OS overheads such as context switches, exception dispatching, etc. We can see that in the case of our minimal handler, handling each faulting store consumes roughly 600 clock cycles, among which the microarchitectural overhead is only a tiny fraction. In more realistic cases where the handler has more complex OS logic, the overhead of microarchitecture and applying the faulting store can be largely ignored.

As explained in subsection 5.3, one imprecise store exception can correspond to multiple faulting stores if they are simultaneously present in the store buffer. If the exception rate is sufficiently high, then faulting stores are handled in batch, and the OS overhead per faulting store is reduced significantly, as shown in Figure 5. The microarchitectural overhead decreases because the store buffer is drained only once for multiple faulting stores. We anticipate that the overhead reduction will be more prominent in realistic cases if the handler schedules batched IO operations.

## 6.5 Performance: real workloads

We run BFS, SSSP, and BC from GAP as well as Silo and Masstree from Tailbench to evaluate the end-to-end performance of our prototype. To inject synthetic imprecise exceptions, we modify the workloads to allocate memory for the graph (in GAP) or the request packets (in Tailbench) from the EInject region. All the allocated memory regions are marked as faulting before the workload starts. For GAP, we configure each workload to process a graph with ~1M nodes and ~8M edges to stress the handling of imprecise exceptions. We use the total execution time, including both the user and OS parts, of the computing kernel as the performance metric. For Tailbench, we run each workload in the integrated mode for a fixed duration and use the aggregated throughput as the performance metric. In both cases, the workloads run as normal Linux processes, experiencing all normal OS activities, including syscalls, task scheduling, and the handling of timer interrupts, page faults, and injected imprecise exceptions. For comparison, we run the workloads with and without imprecise exceptions, denoted as Imprecise and Baseline, respectively, to obtain the relative performance of each workload with imprecise exceptions.

All the workloads can successfully run from the start to the end. During the execution of each GAP workload, roughly 16K~32K injected imprecise store exceptions are transparently handled by Linux. For each Tailbench workload, thousands of imprecise store exceptions are injected per second, which is much larger than the frequency at which page faults are triggered in modern workloads. Our proposed prototype works and interacts flawlessly with the entire hardware/software system.

Figure 6 shows the relative performance of various workloads in the Imprecise case. Our prototype achieves over 96.5% of the Baseline performance for all GAP workloads. Moreover, the difference in user execution time between the Imprecise and Baseline cases is below 1%. For Silo and Masstree, handling extra exceptions absent in the Baseline case reduces the aggregated throughput by less than 4%, which is minimal. Overall, our approach enables imprecise exceptions while maintaining WC performance without excessive silicon overhead.

## 7 CONCLUSION

Integration of compute capabilities in deep cache/memory hierarchies makes it possible to have exceptions with long-latency detection, which can forgo the precise exception semantics expected in modern programming. As continuing to support precise exceptions in all cases comes at a high performance or silicon cost, we investigated supporting imprecise exceptions for only retired stores using efficient hardware-software co-design. We presented a detailed analysis of imprecise store exceptions using memory model formalism to show that we can ensure correct behavior using appropriate hardware and OS support. Finally, we showcase our system design using a RISC-V prototype which passes all litmus tests and successfully runs real workloads, showcasing the success of our approach.

## ACKNOWLEDGMENTS

We thank Boris Grot, Alexandros Daglis, and the anonymous reviewers for their feedback and support. This work was partially supported by FNS projects "Hardware/Software Co-Design for In-Memory Services" (200020B_188696), a Qualcomm Innovation Fellowship (461760), Intel research donation, the Basic Science Research Program of the National Research Foundation of Korea (NRF)

funded by the Ministry of Education (NRF-2022R1C1C1011021), and the National Science Foundation (award 2118851). Finally, we acknowledge the late Michel Dubois, educator, mentor, and friend to us and the field of computer architecture. His seminal work on late exceptions shaped our thinking through this research. We are grateful for his science and dedicate this work to his memory.

# A ARTIFACT APPENDIX

## A.1 Abstract

The artifact provides pre-built binaries and scripts to help users fully reproduce the evaluation results in the paper. The user should have an AWS EC2 f1.2xlarge instance for the cloud FPGA to run experiments. The results that are used to create figures in the paper are also included for reference.

## A.2 Artifact check-list (meta-information)

- **Program:** RISC-V litmus tests, GAP, Tailbench.
- **Run-time environment:** AWS EC2 f1.x2large instance.
- **Hardware:** AWS cloud FPGAs.
- **Metrics:** Total execution time and/or throughput.
- **Output:** Textual log files with expected results included.
- **Experiments:** Scripts provided.
- **How much disk space required (approximately)?:** 1 GB.
- **How much time is needed to prepare workflow (approximately)?:** 1 hour.
- **How much time is needed to complete experiments (approximately)?:** 10 hours.
- **Publicly available?:** Yes.
- **Workflow framework used?:** No.
- **Archived (provide DOI)?:** No.

## A.3 Description

*A.3.1 How to access.* The artifact, containing the pre-built binaries and helper scripts for the experiments, can be downloaded from https://github.com/parsa-epfl/imprecise_store_exceptions.

*A.3.2 Hardware dependencies.*

(1) An AWS EC2 f1.2xlarge instance.

*A.3.3 Software dependencies.*

(1) The python `pandas` library for data analysis.

## A.4 Experiment workflow

All the experiments are performed on the AWS f1.2xlarge instance. The user can refer to the official AWS EC2 documentation (here) to launch, configure and start such instances.

The execution of the experiments on the cloud FPGA cannot be automatically stopped. The user should press `Ctrl-C` to terminate the execution after a "`DONE!`" shows up in the terminal.

To run the microbenchmark:

```
$ make rst
$ make run BIN=mbench.bin LOG=mbench.log MB=1
```

To run the litmus tests:

```
$ make rst
$ make run IMG=litmus.img LOG=litmus.log
```

To run the GAP workloads:

```
$ make rst
$ make run IMG=gap.img LOG=gap.log
$ make run IMG=gap-ref.img LOG=gap-ref.log
```

To run the Tailbench workloads:

```
$ make rst
$ make run IMG=silo.img LOG=silo.log
$ make run IMG=silo-ref.img LOG=silo-ref.log
$ make run IMG=masstree.img LOG=masstree.log
$ make run IMG=masstree-ref.img LOG=masstree-ref.log
```

## A.5 Evaluation and expected results

The user can analyze the results either on AWS or on a local machine. The following commands all assume that the log files generated from previous steps are located in the `post` directory.

We also provide log files used to generate figures in the paper in the `res` directory. Due to uncontrollable timing differences when running the experiments on the cloud FPGA, we expect the results the user generates to be close to ours but not strictly the same.

To analyze the result of the microbenchmark and generate data for Fig. 5 of the paper:

```
$ post/1-mbench.py post/mbench.log
```

To analyze the results of litmus tests:

```
$ post/2-litmus.py post/litmus.log res/herd.log
```

The "`OK`" output indicates that there is no line in the original log file that starts with "`!!! Warning negative differences in`", indicating that the hardware does not exhibit any behavior that the model does not allow (as described in https://github.com/litmus-tests/litmus-tests-riscv/blob/master/README.md), and thus the hardware complies with the memory model.

To analyze the results of GAP and Tailbench workloads and generate data for Fig. 6 of the paper:

```
$ post/3-gap.py post/gap.log
$ post/3-gap.py post/gap-ref.log
$ post/4-silo.py post/silo.log res/silo-ref.log
$ post/5-masstree.py post/masstree.log res/masstree-
    ref.log
```

# REFERENCES

[1] Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *Computer* 29, 12 (1996), 66–76.
[2] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David T. Blaauw, and Reetuparna Das. 2017. Compute Caches.. In *Proceedings of the 23rd IEEE Symposium on High-Performance Computer Architecture (HPCA)*. 481–492.
[3] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models.. In *22nd International Conference on Computer Aided Verification*. 258–272.
[4] Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. 2012. Chisel: constructing hardware in a Scala embedded language.. In *Design Automation Conference 2012*. 1216–1225.
[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization.. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*. 164–177.
[6] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015).

[7] Colin Blundell, Milo M. K. Martin, and Thomas F. Wenisch. 2009. InvisiFence: performance-transparent memory ordering in conventional multiprocessors.. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*. 233–244.

[8] Bob Boothe and Abhiram G. Ranade. 1992. Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessors.. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*. 214–223.

[9] Michel Cekleov and Michel Dubois. 1997. Virtual-address caches. Part 1: problems and solutions in uniprocessors. *IEEE Micro* 17, 5 (1997), 64–71.

[10] Michel Cekleov and Michel Dubois. 1997. Virtual-address caches.2. Multiprocessor issues. *IEEE Micro* 17, 6 (1997), 69–74.

[11] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. 2007. BulkSC: bulk enforcement of sequential consistency.. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*. 278–289.

[12] David Chaiken and Anant Agarwal. 1994. Software-Extended Coherent Shared Memory: Performance and Cost.. In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*. 314–324.

[13] Jonathan Corbet. 2019. Ringing in a new asynchronous I/O API. https://lwn.net/Articles/776703/

[14] Michel Dubois, Christoph Scheurich, and Faye A. Briggs. 1986. Memory Access Buffering in Multiprocessors.. In *Proceedings of the 13th International Symposium on Computer Architecture (ISCA)*. 434–442.

[15] Babak Falsafi and Thomas F. Wenisch. 2014. *A Primer on Hardware Prefetching*. Morgan & Claypool Publishers.

[16] Babak Falsafi and David A. Wood. 1997. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA.. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*. 229–240.

[17] Michael Ferdman, Almutaz Adileh, Yusuf Onur Koçberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: a study of emerging scale-out workloads on modern hardware.. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVII)*. 37–48.

[18] Shaked Flur and Luc Maranget. 2022. RISC-V architecture concurrency model litmus tests. https://github.com/litmus-tests/litmus-tests-riscv

[19] Daichi Fujiki, Xiaowei Wang, Arun Subramaniyan, and Reetuparna Das. 2021. *In-/Near-Memory Computing*. Morgan & Claypool Publishers.

[20] Chris Gniady and Babak Falsafi. 2002. Speculative Sequential Consistency with Little Custom Storage.. In *IEEE PACT*. 179–188.

[21] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. 1999. Is SC + ILP=RC?. In *Proceedings of the 26th International Symposium on Computer Architecture (ISCA)*. 162–171.

[22] James R. Goodman. 1987. Coherency for Multiprocessor Virtual Address Caches.. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*. 72–81.

[23] Siddharth Gupta, Atri Bhattacharyya, Yunho Oh, Abhishek Bhattacharjee, Babak Falsafi, and Mathias Payer. 2021. Rebooting Virtual Memory with Midgard.. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*. 512–525.

[24] Siddharth Gupta, Yunho Oh, Lei Yan, Mark Sutherland, Abhishek Bhattacharjee, Babak Falsafi, and Peter Hsu. 2023. AstriFlash A Flash-Based System for Online Services.. In *Proceedings of the 29th IEEE Symposium on High-Performance Computer Architecture (HPCA)*. 81–93.

[25] Nastaran Hajinazar, Pratyush Patel, Minesh Patel, Konstantinos Kanellopoulos, Saugata Ghose, Rachata Ausavarungnirun, Geraldo F. Oliveira, Jonathan Appavoo, Vivek Seshadri, and Onur Mutlu. 2020. The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework.. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*. 1050–1063.

[26] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. 1993. Cooperative Shared Memory: Software and Hardware Support for Scalable Multiprocesors. *ACM Trans. Comput. Syst.* 11, 4 (1993), 300–318.

[27] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. 1996. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors.. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*. 260–270.

[28] Sorin Iacobovici. 1988. A pipelined interface for high floating-point performance with precise exceptions. *IEEE Micro* 8, 3 (1988), 77–87.

[29] 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

[30] Simon L. Peyton Jones, Alastair Reid, Fergus Henderson, C. A. R. Hoare, and Simon Marlow. 1999. A Semantics for Imprecise Exceptions.. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI)*. 25–36.

[31] 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 H. Katz, Jonathan Bachrach, and Krste Asanovic. 2018. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud.. In *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*. 29–42.

[32] Harshad Kasture and Daniel Sánchez. 2016. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications.. In *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*. 3–12.

[33] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. 1992. Architectural Support for Single Address Space Operating Systems.. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*. 175–186.

[34] Kai Li and Paul Hudak. 1989. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comput. Syst.* 7, 4 (1989), 321–359.

[35] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sánchez, and Nathan Beckmann. 2020. Livia: Data-Centric Computing Throughout the Memory Hierarchy.. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*. 417–433.

[36] Daniel Lustig, Geet Sethi, Abhishek Bhattacharjee, and Margaret Martonosi. 2017. Transistency Models: Memory Ordering at the Hardware-OS Interface. *IEEE Micro* 37, 3 (2017), 88–97.

[37] Todd C. Mowry and Sherwyn R. Ramkissoon. 2000. Software-Controlled Multithreading Using Informing Memory Operations.. In *Proceedings of the 6th IEEE Symposium on High-Performance Computer Architecture (HPCA)*. 121–132.

[38] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2020. *A Primer on Memory Consistency and Cache Coherence, Second Edition*. Morgan & Claypool Publishers.

[39] NVM Express, Inc. 2022. NVM Express Specifications. https://nvmexpress.org/specifications/

[40] Parallel Systems Architecture Lab (PARSA), EPFL. 2020. QFlex. https://qflex.epfl.ch

[41] Xiaogang Qiu and Michel Dubois. 1999. Tolerating Late Memory Traps in ILP Processors.. In *Proceedings of the 26th International Symposium on Computer Architecture (ISCA)*. 76–87.

[42] Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. 1997. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap Between Memory Consistency Models.. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*. 199–210.

[43] RDMA Consortium. 2009. Architectural Specifications for RDMA over TCP/IP. http://www.rdmaconsortium.org/

[44] Steven K. Reinhardt, James R. Larus, and David A. Wood. 1994. Tempest and Typhoon: User-Level Shared Memory.. In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*. 325–336.

[45] RISC-V International. 2022. Specifications. https://riscv.org/technical/specifications/

[46] Bogdan F. Romanescu, Alvin R. Lebeck, and Daniel J. Sorin. 2010. Specifying and dynamically verifying address translation-aware memory consistency.. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XV)*. 323–334.

[47] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Oper. Syst. Rev.* 42, 5 (2008), 95–103.

[48] Somayeh Sardashti, Angelos Arelakis, Per Stenström, and David A. Wood. 2015. *A Primer on Compression in the Memory Hierarchy*. Morgan & Claypool Publishers.

[49] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. 1994. Fine-grain Access Control for Distributed Shared Memory.. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*. 297–306.

[50] Brian C. Schwedock, Piratach Yoovidhya, Jennifer Seibert, and Nathan Beckmann. 2022. täk¯: a polymorphic cache hierarchy for general-purpose optimization of data movement.. In *Proceedings of the 49th International Symposium on Computer Architecture (ISCA)*. 42–58.

[51] SiFive, Inc. 2017. SiFive TileLink Specification. https://static.dev.sifive.com/docs/tilelink/tilelink-spec-1.7-draft.pdf

[52] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd D. Millstein, and Madanlal Musuvathi. 2012. End-to-end sequential consistency.. In *Proceedings of the 39th International Symposium on Computer Architecture (ISCA)*. 524–535.

[53] James E. Smith and Andrew R. Pleszkun. 1985. Implementation of Precise Interrupts in Pipelined Processors.. In *Proceedings of the 12th International Symposium on Computer Architecture (ISCA)*. 36–44.

[54] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. 2015. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly.. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XX)*. 297–310.

[55] Ivan Tanasic, Isaac Gelado, Marc Jordà, Eduard Ayguadé, and Nacho Navarro. 2017. Efficient exception handling support for GPUs.. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 109–122.

[56] David L. Weaver and Tom Germond. 1994. The SPARC Architecture Manual - Version 9. https://www.cs.utexas.edu/users/novak/sparcv9.pdf

[57] Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2007. Mechanisms for store-wait-free multiprocessors.. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*. 266–277.

[58] Wikichip. 2020. AMD Zen3. https://en.wikichip.org/wiki/amd/microarchitectures/zen_3.

[59] Wikichip. 2020. ARM Cortex A76. https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a76.

[60] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. 1993. Mechanisms for Cooperative Shared Memory.. In *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)*. 156–167.

[61] David A. Wood, Susan J. Eggers, Garth A. Gibson, Mark D. Hill, Joan M. Pendleton, Scott A. Ritchie, George S. Taylor, Randy H. Katz, and David A. Patterson. 1986. An In-Cache Address Translation Mechanism.. In *Proceedings of the 13th International Symposium on Computer Architecture (ISCA)*. 358–365.

[62] 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 *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1178–1199.

[63] Arash Pourhabibi Zarandi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. 2020. Optimus Prime: Accelerating Data Transformation in Servers.. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*. 1203–1216.

[64] Arash Pourhabibi Zarandi, Mark Sutherland, Alexandros Daglis, and Babak Falsafi. 2021. Cerebros: Evading the RPC Tax in Datacenters.. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 407–420.

[65] Lixin Zhang, Evan Speight, Ramakrishnan Rajamony, and Jiang Lin. 2010. Enigma: architectural and operating system support for reducing the impact of address translation.. In *Proceedings of the 2010 ACM/IEEE Conference on Supercomputing (SC)*. 159–168.