$(MC)^2$: Lazy MemCopy at the Memory Controller

Aditya K Kamath
Paul G. Allen School of Computer Science & Engineering
University of Washington
Seattle, USA
akkamath@cs.washington.edu

Simon Peter
Paul G. Allen School of Computer Science & Engineering
University of Washington
Seattle, USA
simpeter@cs.washington.edu

***Abstract*—$(MC)^2$ is a lazy memory copy mechanism which can be used within `memcpy`-like functions to significantly reduce the CPU overhead for copies that are sparsely accessed. It can also hide copy latencies by enhancing the CPU's ability to execute them asynchronously. $(MC)^2$'s lazy memcpy avoids copying data at the time of invocation. Instead, $(MC)^2$ tracks prospective copies. If copied data is later accessed by a CPU or the cache, $(MC)^2$ uses the tracking information to lazily execute a copy, when necessary. Placing $(MC)^2$ at the memory controller puts it at the perfect vantage point to eliminate the largest source of memcpy overhead—CPU stalls due to cache misses in the critical path—while imposing minimal overhead itself.**

$(MC)^2$ consists of three main components: memory controller extensions that implement a lazy memcpy operation, a new instruction exposing the lazy memcpy, and a flexible software wrapper with semantics identical to memcpy. We implement and evaluate $(MC)^2$ in the gem5 simulator using a variety of microbenchmarks and workloads, including Google's Protobuf, where $(MC)^2$ provides a 43% speedup and Linux huge page copy-on-write faults, where $(MC)^2$ provides 250× lower latency.

***Index Terms*—lazy copy, memcpy, data transfer, memory controller, memory, DRAM**

I. Introduction

Memory copies significantly impact the execution latency and computational overhead of modern applications. Profiling of Google's datacenters shows that more than 5% of CPU cycles are consumed by memory copy (memcpy / memmove) operations [25, 47], a *substantial* overhead at that scale. Additionally, waiting for copy completion adds "killer microseconds" [5] to application processing which aren't hidden by classical means like out-of-order processing. As we will see in §II-C, most CPU cycles for memcpy are spent stalled, waiting for memory.

CPU memory access latency is not expected to improve in the future due to the classic "memory wall" problem, where technological advances improve clock speeds for CPUs, while memory latencies remain largely stagnant [7]. For example, DDR5 improves memory bandwidth by up to 2× over DDR4 at a slight memory access latency cost [12]. In fact, memory latencies may worsen in the future as cloud providers add higher capacity memories at the expense of latency [17, 21, 32, 35].

Common use cases of memory copies are for temporary buffers. Consider serialization and deserialization mechanisms used to transfer objects across processes and servers [2, 3, 53]. During serialization, a process converts an object into a byte-stream and sends it to another process. The receiving process then deserializes this stream, transforming it back into an object. Both serialization and deserialization can involve many data copy operations as data is moved between the object and the byte-stream. Databases with multi-version concurrency control (MVCC) often use a form of read-copy-update to maintain transactional isolation [58]. Here, the transaction duplicates data it wishes to modify. Modifications happen locally and the duplicate is merged into the database during the commit stage. Copies are also common in operating systems, such as for many IO system calls, for memory defragmentation, and for the `fork` system call. In many of these cases, only a fraction of each copy may be accessed or modified, making the remainder redundant. Further, many accesses occur with a time delay and do not require eager execution of the copy.

A variety of techniques to reduce the overhead of copies have been proposed [43, 44, 55, 61]. For example, Demikernel [61] adds zero-copy APIs to IO stacks, along with programming language based object ownership tracking features. zIO [49] reduces copy overheads in the IO stack without API changes by unmapping copied pages and marking these memory locations as copy-on-access. When a copy is not accessed, the copy latency is saved. Only accessed memory pages incur the copy overhead upon first access. Unfortunately, existing techniques have drawbacks. Zero-copy APIs require significant program redesign to use efficiently. Existing transparent approaches, such as zIO, have high page remapping overheads and thus only provide benefits for large (>16KB) copies of which only a small fraction (≤25%) is accessed, with high performance penalties when this is not the case. Hardware offload techniques, such as DMA engines, have high startup overheads making them similarly impractical for smaller copies [52].

To alleviate the drawbacks of prior approaches, our proposal relies on lazy execution of each memory copy. Laziness is a common technique used across different domains in computer science. When an expensive operation is requested, laziness delays the operation until time of use. For example, functional languages [20] use it to allow declaration of infinitely sized data structures that consume limited memory—only upon data access are the operations resolved and memory is allocated. Operating systems use copy-on-write [48] to avoid performing copies until pages are modified. Laziness provides the advantage that penalties are paid only upon use.

Recent advances in memory and processing logic have allowed significant compute logic to be placed near memory

This work was supported by NSF grants 2212580 and 2212193.

979-8-3503-2658-1/24/$31.00 ©2024 IEEE
DOI 10.1109/ISCA59077.2024.00084

modules, such as within the memory controller [39]. Prior work has taken advantage of this to integrate tasks such as encryption [36], data address remapping [62], or logging [59] within the memory controller. As all memory accesses are marshaled by the memory controller, we believe it is the ideal place to improve data copying mechanisms.

We propose performing Memory Copies lazily at the Memory Controller, i.e., $(MC)^2$. $(MC)^2$ augments the memory controller to allow programmers to issue a lazy memcpy operation on a source and destination buffer. When any CPU or cache writes to the source buffer or reads from the destination buffer, the memory controller performs the copy lazily. At all times, data appears to the program as if it had been copied eagerly. $(MC)^2$ allows applications to avoid the latency of copying in the critical path while only exhibiting overheads for copied data that ends up accessed. $(MC)^2$ also provides the CPU further opportunities to hide copy latencies through prefetching, accelerating copies even in cases where most data is accessed.

$(MC)^2$ enhances the memory controller with a *Copy Tracking Table* (CTT) containing details of prospective copies to be performed lazily. On a memcpy, instead of performing the copy, a new CPU instruction sends a message to the memory controller with the source address, destination address, and copy size which are inserted as an entry in the CTT. When a destination cacheline is read from or a source cacheline is written to, the memory controller consults the CTT, reads the source cacheline from memory, then copies it to the destination, i.e., a *lazy* copy is performed. On a destination cacheline write, the CTT entries are modified to stop tracking the cacheline.

In summary, we make the following contributions:

- We propose a new hardware system, $(MC)^2$ that supports lazy memcpy operations. Programmers may use $(MC)^2$ via a set of new instructions, a `memcpy_lazy` C function, or transparently via a dynamic link library that replaces the standard C library `memcpy` function.
- We implement and evaluate $(MC)^2$ in the gem5 [6] cycle-accurate simulator. $(MC)^2$ has only $\sim$0.2% area overhead and a bank leakage power of 33.8 mW.
- We demonstrate the improvement $(MC)^2$ brings for server-class workloads, with the Protobuf [19] benchmark showing 43% lower runtime, MongoDB's [38] insert operations showing 16% lower latency, and Cicada's [34] transactions having up to 78% higher throughput. $(MC)^2$ also accelerates common OS operations, like huge page copy-on-write faults that have up to 250$\times$ lower latency, and IO buffer copies with up to 99% higher throughput.

We have made our code available at https://github.com/AKKamath/MCSquare-ISCA24 to aid further research.

II. Background

We begin by describing the operation of memory copies down to the hardware architectural level (§II-A). We then look at how and why different types of applications make frequent use of memory copies (§II-B). Finally, we look at the overhead of copying in these applications and determine the hardware architectural source of the overhead (§II-C).

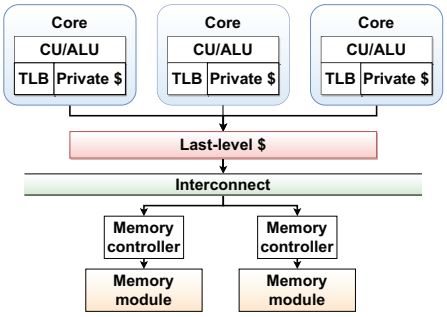


Fig. 1: Base memory system architecture.

A. Memory copy operation

Memory copies (memcpys) are used to transfer a given size of data from a source buffer to a destination buffer. These buffers do not overlap. There are four principal operations involved in a memcpy: load, store, test, and loop [47]. The load fetches data from the source buffer, the store places it into the destination buffer. The test operation checks whether the current copied size matches the provided size. If not, the loop operation restarts the process for the next iteration. Optimizations to memcpy typically involve taking advantage of CPU instructions that support data movement at larger granularity for higher throughput, such as SIMD instructions [9, 26].

While out-of-order and speculative execution allow for some parallelism among iterations of copying, prior work [23] has found that this is largely limited. As the number of copy loop iterations being performed increases, the CPU reorder buffer quickly fills, forcing further iterations to wait. This brings memory access latencies into the critical path of the copy, an effect we will quantify in §II-C.

Memory system effect on memcpy: First, to clearly understand how the memory system affects memcpy overhead, we briefly describe common memory system hardware architecture, shown in Figure 1. It consists of a set of CPU cores, each with their own private cache and a shared last-level cache (LLC). The system also contains a set of memory modules with memory controllers responsible for issuing operations to them. An interconnect connects the memory controllers to the CPU cores via the caches.

For memory read operations, the system first probes the CPU's private cache and the shared LLC to locate the requested data. If unsuccessful, the system transmits the access via the memory interconnect to the appropriate memory controller. This controller then sends a request to the memory module and, upon retrieval, forwards the data to the core via the caches. In contrast, write operations typically involve a direct write to the cache, with the data eventually reaching the memory through subsequent cache evictions.

In this architecture, memory access latencies become progressively worse with distance from the CPU core making the access. The typical dynamic range of memory access latencies can span up to 3 orders of magnitude, from a few nanoseconds for L1 cache access to hundreds of nanoseconds for CXL-attached DRAM and NVM. It is important to note that reads

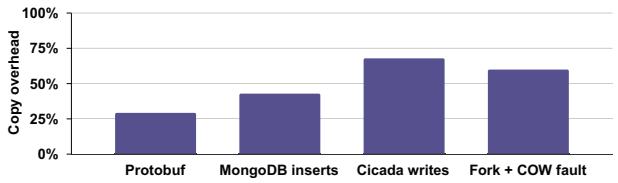


Fig. 2: Many use cases have significant copy overhead.

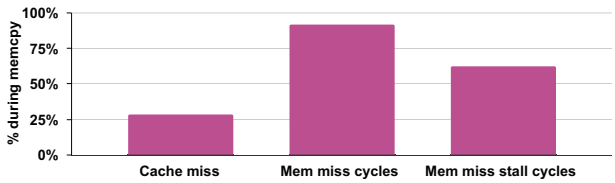


Fig. 3: Source of Protobuf `memcpy` overhead.

incur access latencies on the critical path, while write latencies may be hidden by CPU write buffers and the cache. Thus, reads from the source buffer that miss in the cache have the biggest effect on memcpy overhead by stalling the CPU.

B. Utility of memory copies

We examine a few use cases where memory copies provide substantial utility. Many of these use cases involve copies to create temporary buffers. Temporary buffers are widely used due to the strong isolation and ownership guarantees that they provide. Data that needs to be shared by multiple components of an application can be copied into temporary buffers, giving each component its own local copy. These components can then access and modify data in the buffer without risk of interference by other components. This simplifies program logic, as multiple components of an application do not need to synchronize for local buffer access. In many of these cases, the utility is so great that copies into temporary buffers are used even if not all of the copied data is modified or even accessed.

Serialization: Serialization is a technique used to convert data structures into a format that can be easily transferred between processes. For example, Google's Protobuf [3] is a popular library used for language-agnostic serialization. The process of serialization involves taking a data structure and converting it into a stream of bytes. For this, a buffer is allocated and the data structure is processed and often copied into the buffer. The buffer is then sent to another process, where it is deserialized, i.e., converted back into data structure form.

Many works in the field of ML [24, 57, 67] have noted that copy overheads incurred during serialization have made multiprocessing in Python infeasible. Similarly, prior work [27] developed an accelerator for Protobuf due to the high serialization and deserialization overheads, including copies.

IO buffers: Prior work [49] has noted that IO-intensive applications and the operating system IO stack often make several redundant data copies. Applications like Redis make use of copied buffers to pass data between independent subsystems. These subsystems can modify the data for their specific purpose without having to worry about other subsystems modifying or freeing the buffer, e.g., one subsystem may log data while another inserts it into a hash table. These copies could have been avoided by keeping track of buffer ownership, but this involves complicated, fine-grained memory management and book-keeping to ensure buffers are not freed or modified before subsystems have finished reading.

Multi-version concurrency control (MVCC): Traditional methods of isolation involve the use of locks to ensure that data being modified is not simultaneously read. However, in cases of high contention, this leads to high wait times between readers and writers. To avoid this, in MVCC [58] databases writers instead create copies of tables or tuples being written to and only modify their local copy. At commit, these copies are integrated into the main database. This ensures readers do not end up reading partially modified data. However, this comes at the tradeoff that writers may end up copying data that does not end up being modified.

C. Cost of memory copies

Many use cases across these application domains exhibit high copy overhead. Figure 2 shows the copy overhead for four such use cases which we obtained by running applications on an Intel Skylake server, measuring CPU cycles attributed to memory copy using Linux perf [13]. We see that copy overhead in terms of cycles spent in memcpy can be up to 68%.

Protobuf runs a workload from Google's Fleetbench suite [19] that executes Protobuf operations based on traces from Google's servers. Within this workload, operations such as `MergeFrom*` make heavy use of copying to move data between buffers. MongoDB [38] is a popular NoSQL server that prior work [49] has shown exhibits redundant copy operations to manage IO buffers. Beyond IO, MongoDB also copies inserted fields into an in-memory B-tree for indexing, as well as a log—all of which contribute to the copy overhead shown. Cicada [34] is an MVCC relational database that makes use of copies during write operations for transactional isolation.

`fork` [48] is a common system call used to create a child process. The new process inherits a virtual copy of the memory of the parent process. This is done by creating a copy of the parent page table, then marking all pages as copy-on-write (COW). When a page is modified, a page fault is triggered and the page is copied. As seen in Figure 2, a significant portion of this page fault handling is spent on copying data for 4KB pages. For huge pages, this overhead can reach 99%.

Virtual snapshotting [29, 33] is a technique used by in-memory databases that takes advantage of this feature to take consistent snapshots of the database. This is done by launching a new process whenever a snapshot is needed. The new process then has a virtual copy of the entire in-memory database. While extremely useful, this technique can have high copy overheads in the critical path. For this reason, databases like

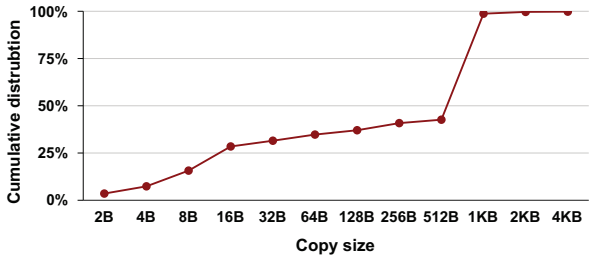


Fig. 4: Distribution of Protobuf `memcpy` sizes.

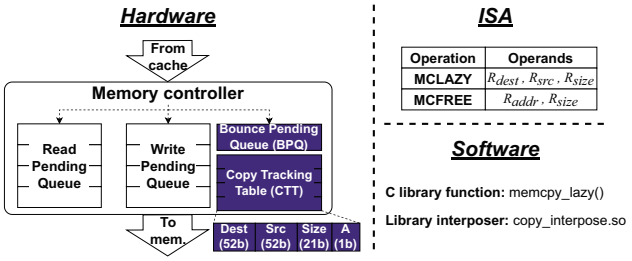


Fig. 5: $(MC)^2$ modifications introduced.

Redis advocate against the use of huge pages [45] due to the copy overhead causing high operation latency spikes.

Memory access stalls are the source of copy overhead: To understand the source of copy overhead in more detail, we perform a deeper analysis of the Protobuf workload. Figure 3 shows memory access statistics obtained from perf during the `memcpy` calls of the Protobuf workload. We see that more than 25% of data accesses end up missing in the cache and have to be serviced from memory. More than 90% of the time, at least one instruction within the CPU is waiting for a memory access to be serviced (Mem miss cycles). These instructions take up a slot in the CPU's reorder buffer (ROB) and can reduce the CPU's effective throughput by blocking further instructions from entering the ROB [23]. Due to this, for more than 60% of the cycles spent in `memcpy`, the CPU is completely stalled (Mem miss stall cycles).

Many memcpys are too small for OS-based avoidance: We also analyzed the sizes of `memcpy` operations executed by the Protobuf workload, shown in Figure 4. We find that the majority of copies ($\sim 56\%$) copy a single kilobyte. An ideal solution to resolve this overhead must thus be able to speed up sub-page sized copies. Existing OS techniques [49] that require page-sized or larger copies cannot provide any benefit.

III. $(MC)^2$ Design

$(MC)^2$'s primary goal is to eliminate copy overhead in the critical path. $(MC)^2$ has to accomplish this while ensuring data consistency. Figure 5 contains all the modifications and new features $(MC)^2$ introduces to accomplish these goals. The left side shows the modifications we make to the memory controller. The right side shows two instructions we introduce, along with the software support that we provide for $(MC)^2$. We now cover the hardware changes made to support $(MC)^2$, then look at its software interface and how memcpy operations can be transparently replaced with their lazy alternative.

A. $(MC)^2$ memory controller design

We start by making changes to the memory controller (MC) to provide support for lazy copies. For clarity, we use the term *prospective copy* to refer to a lazy copy that the processor has requested. This copy is not immediately performed. Instead, we add a *copy tracking table* (CTT) to each MC to track each prospective copy. We use the term *lazy copy* to refer to the act of finally copying data from the source to the destination, typically triggered by an access to either buffer that requires the copy. To manage in-progress writes to source buffers, we extend each MC's existing write pending queue with a *bounce pending queue* (BPQ) to hold the corresponding write, while the lazy copy is performed. The left side of Figure 5 depicts these changes with our additions highlighted in purple. We assume memory accesses reaching the memory controller are at cacheline granularity, typical of most modern systems. $(MC)^2$ supports prospective copies at a byte granularity, but we simplify MC design by restricting tracked destination buffers and lazy copy sizes to be of cacheline granularity. A software wrapper (III-D) converts byte-granularity memcpys to equivalent cacheline-granularity prospective copies.

1) Copy Tracking Table (CTT): We add a copy tracking table (CTT) to each MC that tracks prospective copies. The CTT is an SRAM-based module that performs lookups using the physical address of memory accesses. These lookups are in parallel with the memory access, avoiding overheads on the critical path of access. The CTT ensures that reads to destination buffers are correctly routed (*bounced*) to corresponding source buffers. As the source and destination buffers may be placed across multiple memory modules, we ensure that CTTs across MCs are kept consistent. This is done by snooping the interconnect for broadcast messages informing the MCs of table modifications.

Table entries: Each entry in the CTT occupies 16 bytes consisting of a 52-bit source physical address, 52-bit destination physical address, 21-bit size, an active bit (as shown in Figure 5) and 2 unused bits. Addresses are tracked with 52 bits as this is the upper limit of physical address sizes that most systems support [14, 60]. A 21-bit size allows a single entry in the CTT to track a lazy copy of up to 2MB, the size of a huge page. Many copied buffers do not exceed this size. Further, memory fragmentation causes larger physically contiguous regions to be rare [64], making larger tracking granularities unnecessary.

Table logic: The CTT contains logic to ensure that tracked destination buffers do not overlap. Specifically, if an existing entry contains (part of) a destination buffer that a new operation is inserting, the existing entry is removed (or resized) so that the new and existing entries' destination buffers do not overlap. This corresponds to the case where data is copied to a buffer and then new data is copied to the same buffer. Every destination buffer thus has a unique source buffer.

In addition, when a new entry is inserted into the CTT, we

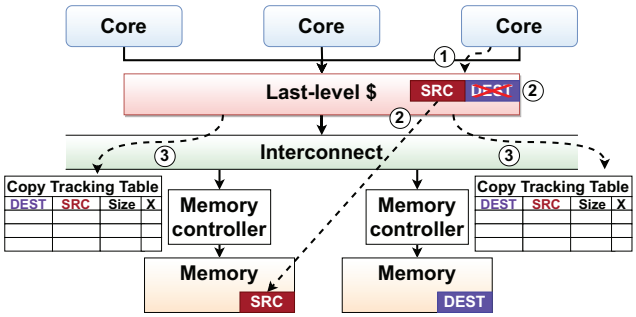


Fig. 6: Prospective copy tracking.

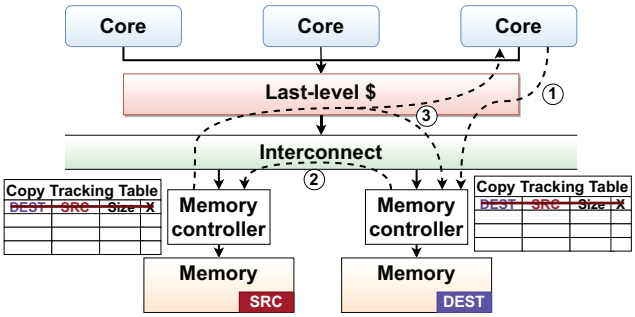


Fig. 7: Lazy copy on destination buffer access.

check whether any part of the source buffer of the new entry was a destination buffer in an existing entry. If so, we split the new entry with the overlapping part using the source buffer of the existing entry. For example, a lazy copy operation is initiated with buffer A being copied to buffer B (copy 1), then buffer B being copied to buffer C (copy 2). The entry in the CTT corresponding to copy 2 will show A being copied to C. This avoids chains of copies, simplifying dependence tracking.

The CTT also merges multiple lazy memcpy operations to a single entry when it finds that the copies are to contiguous source and destination buffers. This can occur when multiple copies are to logically separate entities within the same buffer, e.g., element-by-element copies of an array.

Required table storage: We allocate 2,048 entries in each CTT to allow it to track a large number of active copies. The CTT access latency is negligible compared to the DRAM access latency, avoiding overheads in the critical path (see §V). Altogether, the CTT uses 2,048×16B or 32KB of SRAM.

Avoiding CTT overflow: If the CPU continuously issues lazy copy operations, the CTT can fill up. To avoid this, copies are performed by the MC asynchronously, with entries freed on copy completion. The tradeoff is that asynchronous copies lead to increased memory bandwidth utilization and reduce the potential of avoiding redundant copies. Conversely, waiting until the CTT is full before copying causes stalls to the CPU as it waits for entries to be freed.

To strike a balance, $(MC)^2$ starts lazy copying when the CTT becomes 50% full. For this, the MC identifies entries with the smallest size and creates read requests for their source buffers. Once the read is complete, the data is written to the destination buffer and the entry is then removed from the table. For large servers capable of issuing sizeable bursts of copies, the CTT frees multiple entries in parallel, leveraging the increased bandwidth of these servers (§V-C). This requires only a few bytes for counters to keep track of the number of entries being freed.

2) Bounce Pending Queue (BPQ): If data in a source buffer is being modified, we must ensure that this data is first copied to its destination buffer(s). To handle this, we extend the existing write pending queue (WPQ) in the MC with an additional bounce pending queue (BPQ). The BPQ contains writes to the source buffer that are waiting for data to be copied to the destination buffer(s). Separating the BPQ from the WPQ prevents stalling other writes that could have proceeded. If the number of writes to source buffers exceed the size of the BPQ, further writes are stalled by the MC, creating back-pressure on the caches. We find that a small BPQ supporting 8 cachelines is sufficient to absorb bursts of source buffer writes. We explore the impact of the number of CTT and BPQ entries, and the asynchronous copy threshold in more detail in §V-C.

B. $(MC)^2$ functionality

We examine how $(MC)^2$ interacts with the CPU and memory system and how data consistency is preserved with lazy copies.

1) Lazy memcpy: Requesting a prospective memcpy involves 3 main steps, shown in Figure 6. The source buffer is shown in maroon and the destination buffer is shown in purple.

① The CPU issues `MCLAZY`, creating a packet containing the source buffer address, destination buffer address, and copy size. It sends this packet to the caches.

② Once the packet reaches the caches it triggers writebacks for all the cachelines contained in the source buffer, and invalidates the cachelines contained in the destination buffer. The caches' FIFO write buffer ensures that the writebacks reach the MC before the `MCLAZY` packet. This guarantees that further MC-observed writes were issued after the lazy copy operation, necessary for memory consistency.

③ The packet is then broadcast across the memory interconnect, and all the MCs insert a new entry containing the details of the lazy copy into the CTT.

2) Memory access: The CTT is consulted for every MC-observed memory access. If the memory access is to one of the tracked source or destination buffers, $(MC)^2$ may need to specifically handle it. There are four types of such memory accesses. Namely, reads from destination, reads from source, writes to destination, and writes to source buffers. We will discuss each one in detail.

Read from source: As these accesses do not modify data and the source is up-to-date in memory, they proceed without interference.

Write to destination: If a write to a destination buffer reaches the MC, we no longer need to track it, as the memory will now contain fresh data. The MC removes the entry from the CTT on completion of the write (or splits the entry if it spans multiple

cachelines). It also broadcasts a message on the interconnect for the other MCs to update their CTTs.

Read from destination: As this is a prospective copy, the data in the destination buffer in memory is stale. Retrieving the correct data involves 3 steps, shown in Figure 7:

① The MC fetches the appropriate source buffer address corresponding to the destination address from the CTT.

② It bounces the request to the MC containing the source. This MC then reads the respective source cacheline from memory and stores the data in the destination response packet.

③ This packet is sent back to the core as if it was read from the destination. As we now have the up-to-date version of the destination cacheline, a copy of this packet is also sent as a write to the MC containing the destination buffer, eventually removing the corresponding entry from the CTT. This prevents future accesses to the cacheline from suffering further overheads.

Write to source: If a write to a source buffer cacheline reaches the MC, we need to execute a lazy copy. To do so, the write is first held in the BPQ and a read to the same source cacheline is generated and sent to memory. Once the source cacheline is obtained from memory, the MC generates packets for each destination cacheline that has a prospective copy with the source. The data read from the source cacheline is copied into these packets. The completed destination cachelines are then written to memory and corresponding CTT entries are removed (or resized). Once complete, the corresponding BPQ entries are written to memory.

Unaligned copies: If source and destination copy buffers are not cacheline-aligned with each other, a lazy copy to a destination cacheline may require data from multiple source cachelines. This results in multiple bounces to fetch the entire destination cacheline. For example, if a prospective copy was from physical address 100 (source) to address 512 (destination), we require two bounces to reconstruct the destination; the first access to address [64 - 127] and the second to [128 - 191], which may lie in separate memory modules.

Reducing bandwidth contention: When a destination buffer is read, a copy of the reconstructed cacheline is sent as a write to memory. To avoid contending on memory bandwidth, if the WPQ of the destination MC is more than 75% full, it rejects the write to prioritise the memory bandwidth for accesses from the caches. Otherwise, it writes the destination to memory and removes (or resizes) the corresponding CTT entry. Further reads to this destination cacheline are serviced from memory as normal. §V-A2 evaluates the associated overhead reduction.

C. $(MC)^2$ ISA design

We provide two new instructions for programmers to take advantage of lazy memcpy, shown on the right in Figure 5.

Lazy copy: `MCLAZY` enables the lazy memcpy. It takes three register operands. The first register contains the virtual address of the destination buffer, the second register contains the virtual address of the source buffer, while the last register contains the lazy memcpy size. The destination buffer cachelines encompassed by the operation are invalidated while the source buffer cachelines are written back from the cache (see §III-B1).

```
def memcpy_lazy(dest, src, size):
    # Cacheline align dest
    left_fringe = ALIGN_REM(dest, CL_SIZE)
    memcpy(dest, src, left_fringe)
    dest += left_fringe
    src += left_fringe
    size -= left_fringe
    while size > 0:
        # Calculate remaining size in page
        src_off = ALIGN_REM(src, PAGE_SIZE)
        dest_off = ALIGN_REM(dest, PAGE_SIZE)
        # Pick minimum size left as lazy copy size
        copy_size = min(min(src_off, dest_off), size)
        if copy_size < CL_SIZE:
          memcpy(dest, src, copy_size)
        else:
          # Make copy_size a multiple of CL_SIZE
          copy_size &= ~(CL_SIZE - 1)
          MCLAZY(dest, src, copy_size)
        dest += copy_size
        src += copy_size
        size -= copy_size
    mfence()
```

Fig. 8: Lazy memcpy function pseudocode.

Alignment requirements: The destination address must be cacheline-aligned and the value contained in R_{size} must be a multiple of the cacheline size. This simplifies the MC logic, as destination cachelines are guaranteed to be lazily copied in their entirety, and avoids partial cacheline invalidations for the destination buffer. In §III-D we shall see how a software wrapper removes these requirements.

The source and destination must each be contiguous in physical memory. For user-space applications with buffers spanning multiple pages, the instruction must be called separately for each page. The instruction requires at most two address translations, one for the source and one for the destination.

Instruction parallelism: `MCLAZY` obeys memory consistency similar to `CLFLUSHOPT` and `CLWB` [22]. This means that separate invocations of the instruction proceed in parallel, without serialization required for stores in the x86-TSO memory model [41]. To enforce ordering with future operations, an `MFENCE` or `SFENCE` operation must be called.

Freeing: Finally, we provide an `MCFREE` operation that takes two operands: an address register and a size register. `MCFREE` sends a hint to the MC that the buffer defined by the address and size registers is no longer useful and can be freed. The MC can remove all entries in the CTT where the destination buffer is contained in the freed buffer. Data in the freed buffer needs to be reinitialized before reuse, i.e., a read operation to a freed buffer following an `MCFREE` call is undefined. This instruction can be called within functions like `munmap` where the buffer is guaranteed to no longer be used.

D. Software Design

C/C++ function: To remove constraints on the programmer, we provide a C/C++ library function `memcpy_lazy` that has the same semantics as a standard `memcpy` call. Internally, this function calls `MCLAZY` for each page in the buffers and ensures that the destination is cacheline-aligned.

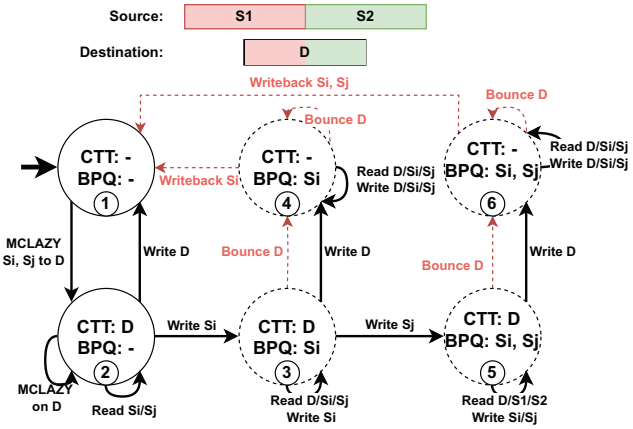


Fig. 9: $(MC)^2$ simplified state transition diagram.

The pseudocode for this function is shown in Figure 8. Here, CL_SIZE refers to the cacheline size, while the macro ALIGN_REM returns the number of bytes required to align a given address to a given alignment size. First, we identify the number of bytes needed to be added to the `dest` address to cacheline-align it (line 3). We then copy these bytes from the source to destination (line 4). The destination is now cacheline aligned. We then identify the number of bytes remaining in the source and destination pages (lines 10 - 11) and pick the smaller one (line 13). This ensures that the copy is being performed on a single page for both the source and destination. If this is smaller than the cacheline size, we manually memcpy it (line 15). Otherwise, we make the copy size a multiple of the cacheline size and call `MCLAZY` for it (lines 18 - 19). We then repeat the process until the entire copy is complete. We finally issue an `mfence` call to order the prospective copies with future memory accesses (line 23). The read and write operations of a standard `memcpy` are replaced by a `MCLAZY` call per copied buffer page, enabling our lazy memcpy goal.

Memcpy interposition: We provide an additional interposer library (`copy_interpose.so`) that converts memcpy calls to lazy memcpy calls for legacy applications. This removes the need to rewrite code to take advantage of $(MC)^2$.

E. Data protection and correctness

We now look at $(MC)^2$'s data protection and correctness guarantees. For concreteness, we assume the x86 CPU architecture and x86-TSO memory model [41], although $(MC)^2$ may be applied to other models with minor modifications.

Protection: As $(MC)^2$ deals with only physical addresses, regular address translation occurs within the CPU where page protection bits are checked. If a process tries to copy data it does not have access to, the MMU raises a page fault. On unmapping pages, the operating systems zeroes out the newly freed pages before providing them to new processes. This eventually reaches the MC as writes, removing entries from the CTT and avoiding data leaks from occurring between processes. This ensures that $(MC)^2$ does not expose data protection risks.

Memory consistency: $(MC)^2$ must ensure that reads and writes to source and destination buffers preserve memory consistency. In $(MC)^2$, each destination cacheline behaves independently. This destination cacheline is a prospective copy of a single source cacheline if they are cacheline-aligned. In cases of misalignment, the destination is split across two source cachelines. To examine the interaction of memory accesses with $(MC)^2$, we make use of a state transition diagram, seen in Figure 9. States ① - ④ cover the transitions for when source and destination are cacheline-aligned. When misaligned, we have additional states ⑤ - ⑥.

Assume that S1, S2 and D are each cachelines, with S1 and S2 being contiguous physical addresses. Part of S1 and S2 are prospective copies to the destination cacheline D, meaning part of D's data lies in S1 and part in S2, as seen in the top of Figure 9. Note that S1, S2, and D may all lie in different memory modules. The different states in the transition diagram show the different possible states of the BPQ and CTT. State transitions are caused by memory accesses by the CPU [in black] or $(MC)^2$ [in red]. To understand how $(MC)^2$ provides memory consistency, we shall now go through each state of the diagram and analyze all the interactions and transitions to see how the system reacts in each case.

① We start with an empty CTT and BPQ. In this state, $(MC)^2$ has no impact. On receiving a prospective copy, we transition to state ②.

② In this state, the CTT contains an entry noting that S1 and S2 have been lazily copied to D. Writing to D removes the entry from the CTT and transitions us back to state ①, while reading S1 or S2 has no impact. Performing another prospective copy with destination D retains us in the same state, with the CTT entry being modified to contain the new source(s). On writing to either S1 or S2 (denoted by Si), we move to state ③, where the write is kept in the BPQ.

③ This is a transitional state, where a bounce packet is generated which reads S1 and S2 from memory (and not the BPQ) and then writes to cacheline D. On completion of the write, we move to state ④, denoted by the "Bounce D" transition. A similar transition occurs when the CPU directly writes to D. Reads and writes to Si issued by the CPU are merged and serviced directly from the BPQ. On a write to Sj (j != i) we move to state ⑤.

④ This is also a transitional state. D is removed from the CTT as its data has been retrieved and written to memory, i.e., the lazy memcpy is complete. Reads and writes to D are serviced normally by the memory as the CTT does not contain an entry for it. The BPQ writes Si to memory to move back to stable state ①. If Si is a common source for multiple prospective copy destinations, e.g., D1, D2, D3..., the BPQ must wait until all entries with Si as source are removed from the CTT before writing Si to memory.

⑤ This is also a transitional state, where both S1 and S2 are kept in the BPQ (denoted by Si and Sj). Similar to state ③, all reads and writes to S1 and S2 from the CPU are serviced by the BPQ, while the bounce packet for D reads directly from memory. Writing to D or completion of the bounce operation

moves us to state ⑥.
⑥ Similar to state ④, D is removed from the CTT. S1 and S2 are written back to memory to move back to state ①. It may occur that S1 and S2 lie in different memory modules operated by different MCs, with both independently generating identical bounce requests for D. One may complete, while the other is still pending. For this, bounce requests for D are dropped on reaching this state.

In all the transitional states where the BPQ contains either S1 or S2 (③ - ⑥), prospective copies involving S1 or S2 are stalled until S1 and S2 have been written back to memory.

F. Performance Tradeoff Discussion

While $(MC)^2$ provides the same semantics as the standard memcpy and aims to accelerate it, there are performance tradeoffs. We discuss them in this section.

Cached source buffers may harm performance: When the source buffer is already present in the cache, the overhead of copying is low as the CPU fetches data from the cache instead of memory. Replacing these copies with $(MC)^2$ could harm performance. Despite this, we shall see in V-A1 that the latency of $(MC)^2$ is not significantly worse than a cached copy.

Reduced cache pollution: Cache pollution is a common problem associated with buffer copies [52, 66], where the destination of the copy is not immediately accessed after the copy. As $(MC)^2$ explicitly invalidates destination buffers from the cache, it avoids this problem.

Copy-and-access may harm performance: If a destination buffer is immediately accessed after a prospective copy, $(MC)^2$ could reduce performance, as the destination buffer's cache lines have been invalidated. We shall see in V-A2 that prefetching eliminates this performance impact in many cases.

Memory footprint: Prior work, such as zIO [49], performed copy elision by unmapping destination pages and marking them copy-on-access, reducing the memory footprint for copies with unaccessed destination buffers. $(MC)^2$ does not reduce memory footprint, as both source and destination buffers must be physically allocated by the OS before lazy copying. This allows $(MC)^2$ to remain transparent to the OS and support lazy copies at sub-page granularity, avoiding hardware interrupts and page faults on access to destination buffers. Nevertheless, for an application like Protobuf, we found that zIO provided no memory footprint reduction due to sub-page sized copies. For MongoDB, we found only a modest 6 – 8% reduction in memory caused by copied data frequently being accessed.

IV. Evaluation methodology

We simulate the performance of $(MC)^2$ by extending GEM5-v22.1 [6]. Table I gives the configuration details of our simulation. Our system resembles a scaled-down server node.

We use CACTI 7.0 [4], a tool that calculates cache specifications based on provided parameters, to obtain the CTT access latency and area. For our configuration, with a 22 nm transistor size, we find that the CTT area is 0.14 mm^2 which is negligible compared to I/O die areas of 100 mm^2 [40]. The bank leakage power is 33.8 mW. The CTT latency is 0.79 ns, significantly lower than typical DRAM access latencies (15 - 90 ns [12]). The CTT latency is incurred when the destination of a prospective copy is read. The entry in the CTT is looked up and the ongoing access is preempted, incurring the CTT latency. The packet is then bounced towards the source.

TABLE I: Simulated configuration.

Hardware			
CPUs	8	**Clock speed**	4 GHz
Private L1 cache	64 KB/CPU, Stride prefetcher	**Shared L2 cache**	2 MB, Stride prefetcher
DRAM size	3 GB	**DRAM channels**	2
DRAM config.	DDR4	**BPQ size**	8 entries
CTT entries	2,048	**CTT latency**	0.79 ns
Software			
OS kernel	Linux 5.7.0	**Distribution**	Ubuntu 20.04

Throughout our evaluation we make use of our C wrapper function `memcpy_lazy`. To accurately model the performance of cacheline writebacks required by `MCLAZY`, this function calls the `CLWB` instruction [22] for each cacheline that needs to be written back. It also cacheline-aligns the provided destination, calling `memcpy` for unaligned fringes (as mentioned in §III-D).

Baselines: We compare our approach to a baseline memcpy operation. We also compare against zIO [49], a state-of-the-art approach for OS-assisted zero-copy IO. zIO elides memcpy operations and tracks the copies in a skiplist. The page table entries are marked as copy-on-access using userfaultfd [1]. On a page fault, zIO allocates physical memory for the destination buffer and performs the copy. We modify zIO to perform elision on all `memcpy` calls instead of just IO-based copies.

V. Evaluation

We evaluate $(MC)^2$ and tease out the performance implications of the lazy memcpy technique through a set of microbenchmarks. We then look at the full-system performance of $(MC)^2$ through benchmarks and applications consisting of Google's Protobuf, MongoDB, and Cicada. We also analyze the impact of $(MC)^2$ on Linux kernel buffer copies and huge page faults. We conclude by examining $(MC)^2$'s sensitivity to different configuration parameters.

Our evaluation answers the following questions:

1) How much lower is $(MC)^2$'s memcpy critical path overhead? What are the main sources of overhead for $(MC)^2$'s memcpy? (§V-A1)
2) What is the impact of *lazily* copying data upon access? (§V-A2)
3) What benefit does $(MC)^2$ provide to applications? (§V-B)
4) How do $(MC)^2$'s parameters impact its performance? (§V-C)

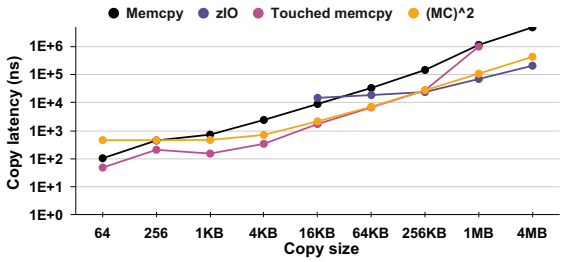


Fig. 10: Copy latency for native `memcpy`, zIO and (MC)^2.

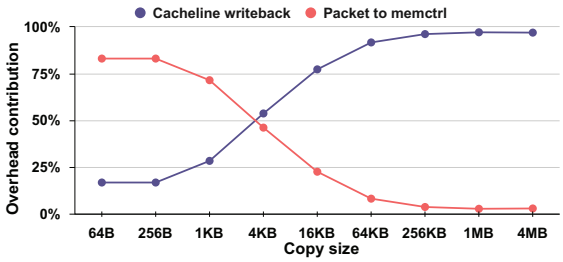


Fig. 11: Overhead breakdown of `memcpy_lazy`.

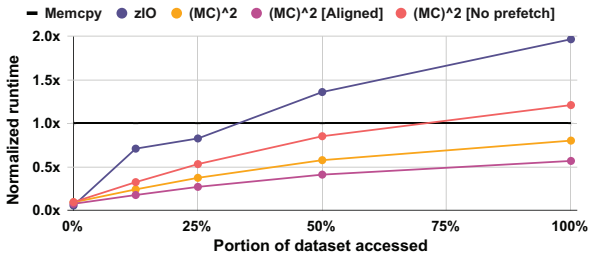


Fig. 12: Sequential destination buffer access overhead.

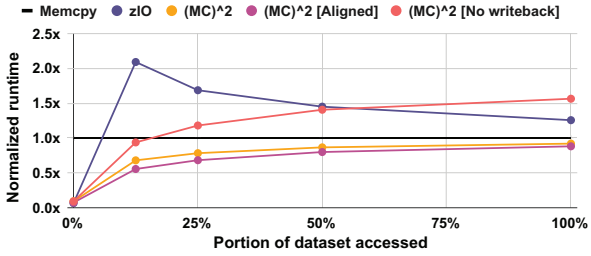


Fig. 13: Random destination buffer access overhead.

A. (MC)^2 performance implications

We first analyze the performance tradeoffs that lazy memcpy provides for different memcpy scenarios.

1) Copy latency and overheads: We examine copy latency of (MC)^2 by performing `memcpy` operations on various sizes of data regions prefaulted into memory. We then take a look at the breakdown of overhead that (MC)^2 incurs.

Uncached source buffer: Figure 10 shows the latency for zIO memcpy elision and (MC)^2 lazy memcpy compared to a baseline of native memcpy (lower is better). (MC)^2 enables cacheline-sized lazy copies, with speedups for copies 1KB and larger. It has a significantly lower overhead than zIO for smaller copies, and is 55% to 11× faster than `memcpy` for copies of a kilobyte and larger. As zIO relies on page table copy-on-access for elision, it requires copy sizes of at least a page to be able to perform elision. The overhead of unmapping pages and issuing TLB shootdowns ends up degrading performance for smaller copy sizes, with zIO performing worse than native memcpy for 16KB copies. zIO's cost becomes justified for copy sizes of 64KB or larger, with a speedup of 23× over `memcpy` at 4MB. Despite this, we shall see that when the destination buffer is accessed, zIO suffers significant mis-speculation penalties, degrading the overall performance.

Cached source buffer: We also analyze the performance of `memcpy` when the source buffer has already been touched before the operation leading to it being cached (Touched memcpy). We see that this outperforms (MC)^2 for smaller sizes, however, for 16KB and above (MC)^2 is able to provide a similar memcpy latency. This shows that (MC)^2 can be used to provide copy latencies similar to cached copies, regardless of whether the data is present in the cache.

Overhead breakdown of `memcpy_lazy`: The primary overheads for (MC)^2 memcpy are writing back cachelines and sending the prospective copy operation to the memory controller. We separately measure these latencies, shown in Figure 11 (lower is better). For copies smaller than 1KB, `CLWB` instructions can proceed in parallel, leading to minimal impacts in performance. Above 1KB, these operations serialize due to the CPU load/store queue and ROB becoming full. The lazy copy packets being sent to the MC proceed in parallel, reducing their impact.

There is scope for improving cacheline writeback latencies. For large copy sizes, a single full-cache writeback operation akin to `INVD` [22] could be introduced, providing a single fixed overhead for cacheline writeback regardless of copy size. For smaller copy sizes, a wider writeback operation could be provided (for example, operating at a page granularity), further reducing its overhead. We therefore view the current overheads of (MC)^2 prospective copies as a conservative estimate.

2) Data access latency: As our approach lazily copies data, we end up with increased data access latencies for buffers of prospective copies. We now look at the impact this latency has and optimizations that allow us to minimize it.

Sequential destination buffer access: For this experiment, we measure the runtime of copying a 4MB source to a destination buffer followed by iterating through the destination buffer reading elements sequentially and accumulating the values into a local variable. This is essentially a streaming access pattern, commonly found in operations like serialization and deserialization. We purposely misalign the source and destination buffer so that (MC)^2 suffers the increased penalty of two bounces during destination access.

Figure 12 shows the runtimes of zIO and (MC)^2 relative

to native memcpy for different access proportions of the destination buffer. While zIO is around 70% faster than $(MC)^2$ when the dataset is not accessed, we see that this swiftly degrades, with zIO performing worse than native memcpy when half or more of the dataset is accessed. This is because zIO performs copy-on-access and has to handle page faults that add additional overhead.

Interestingly, we see that $(MC)^2$ consistently outperforms the native `memcpy` for all access proportions with a worst case runtime of 80% that of `memcpy`. This is because the cache prefetcher predicts the sequential access pattern and prefetches the destination cachelines before the CPU requests them. The prefetches allow some of the extra latency caused by bouncing to be hidden. We can see that, when prefetching is turned off (No prefetch), $(MC)^2$ performs up to 21% worse than native `memcpy`. Conversely, if the source and destination buffers are both cacheline-aligned with each other (Aligned), $(MC)^2$ is able to perform even better at up to 57% the runtime of `memcpy`, as destination accesses bounce only once.

Random access: We repeat this experiment with a random access pattern. We perform a pointer-chasing experiment where each element contains the index of another element in an array contained in the copied buffer. We ensure that every index is unique and randomly distribute the indices among the elements. This brings the memory access latency to the critical path, as every subsequent access is dependent on the value of the previous one, preventing any data access parallelism or prefetching. This type of access pattern is relatively uncommon and degrades the benefit of caches.

Figure 13 shows runtimes of zIO and $(MC)^2$ relative to native `memcpy`. As the access pattern is now random, zIO at 12.5% access suffers from frequent page faults causing its runtime to be 2.1× native `memcpy`. Once more of the dataset is accessed, these accesses occur to already copied pages, causing the runtime to decrease to 1.3× that of `memcpy`.

$(MC)^2$ on the other hand, has a much lower runtime of 92% of memcpy. Our optimization—writing back the destination cacheline after it bounced—is a significant source of this reduced overhead. When the completed read is not written back (No writeback), the latency degrades to at most 1.6× `memcpy`, performing worse than zIO. This is because every memory access bounces twice to reconstruct the destination value, leading to an effectively doubled memory access latency. As before, when the destination and source buffers are cacheline-aligned with each other (Aligned), $(MC)^2$ outperforms `memcpy` with a worst case runtime of 88% of memcpy. Here, the memory copy latency was reduced by $(MC)^2$, and the memory access latency is only slightly higher than normal due to destination accesses bouncing only once.

B. Application Workload Evaluation

Google Protobuf: To evaluate the effectiveness of $(MC)^2$ on a real workload, we ran the Protobuf workload provided in the Google Fleetbench [19] benchmark suite. Fleetbench consists of workloads dedicated to common "hot" library functions using traces obtained from production servers. Google's Protobuf

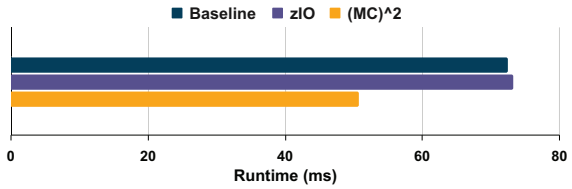


Fig. 14: Runtime of Protobuf.

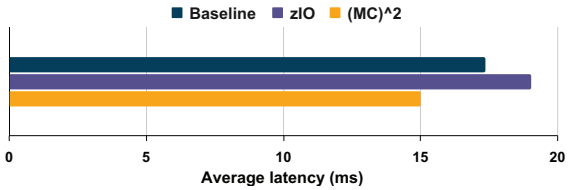


Fig. 15: MongoDB average insertion latency.

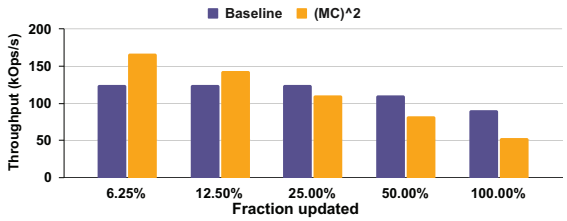
library [3], which provides a language-agnostic framework for data serialization, is a major workload in Fleetbench. The Protobuf workload calls different Protobuf functions with message sizes obtained from server traces. We accelerate this workload with $(MC)^2$ using the library interposer to redirect memcpy calls 1KB and larger to our `lazy_memcpy` function.

In Figure 14 we see the runtime obtained for the baseline Protobuf workload compared to the runtimes of the workload with zIO and $(MC)^2$. We find that all `memcpy` operations were below page size preventing zIO from performing any elision. $(MC)^2$ provides 43% speedup over vanilla `memcpy`.

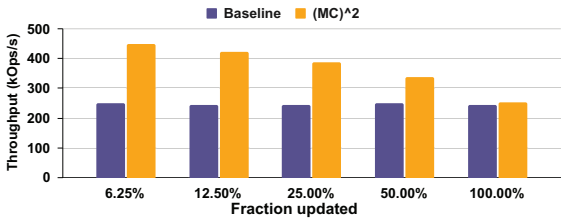
MongoDB: We examine $(MC)^2$'s effectiveness in eliminating redundant copies in IO buffers by running MongoDB [38] with $(MC)^2$ and zIO's MongoDB copy elision interposer [50]. We replicate the experiment performed by Stamler et. al. [49], where a client runs the YCSB [11] load phase with 100KB fields and 10 fields per insertion. The load phase performs 100% inserts in a uniform random distribution. We scale down the number of insertions to 50 to make the simulation time feasible. We run this workload 3 times and report the average insertion latency in Figure 15. $(MC)^2$ speeds up insertions by 15.5%, while zIO slows down insertions by 9.7% due to frequent accesses to copied data.

For large copy sizes, zIO is supposed to provide better copy latencies than $(MC)^2$ (§V-A1). Despite this, $(MC)^2$ provides better performance than zIO as it does not have the page fault penalties that zIO experiences when prospective copies are accessed. MongoDB copies data into an in-memory B-tree used for indexing and into a log during transaction commit. During this process, it accesses copied data. For zIO, this triggers a page fault, forcing it to perform a copy on the accessed page, which $(MC)^2$ avoids.

Multi-version concurrency control: Multi-version concurrency control (MVCC) [58] is a popular database transactional consistency technique. With MVCC, write transactions that modify data first create a local copy of the tuples being modified. Only the local copies are modified, creating a new version of

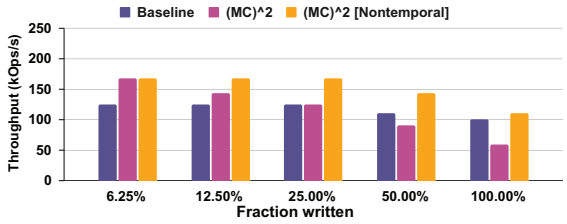


(a) 1 thread.

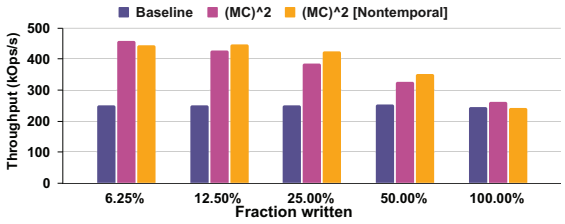


(b) 8 threads.

Fig. 16: Read-modify-write MVCC database throughput.



(a) 1 thread.



(b) 8 threads.

Fig. 17: Write-only MVCC database throughput.

the data upon commit. These new versions are hooked back into the main database, discarding the old data. This allows concurrent read transactions to read consistent data from the main database without requiring locks.

Transactions often only update a small portion of a tuple, incurring unnecessary copy overhead. For example, TPC-C has frequent operations that update a single tuple attribute (e.g., decrementing the quantity of stock of an item [31]) which updates only 1–2% of the tuple. Sub-tuple copies drastically increase the complexity of version management and are thus avoided by databases [58].

$(MC)^2$ allows MVCC databases to utilize tuple-wise copying, while paying the copy penalty only for the portions updated. To demonstrate this, we enhance the Cicada MVCC database [34] with $(MC)^2$. We perform repeated operations on a table with 8KB-sized rows, modifying different fractions of the tuples and measuring throughput. The operations are split in a 50:50 ratio between reads and updates, typical of write-intensive database workloads [11]. We ran this experiment with one and eight threads performing transactions to analyze performance when latency-bound and bandwidth-bound respectively.

Figure 16 shows the transaction throughput of baseline Cicada compared to $(MC)^2$ when the updates are read-modify-write (RMW) operations. The baseline reads data from memory during `memcpy` then performs the RMW locally in the cache. $(MC)^2$ avoids the memory read during `memcpy`, and only reads from memory the fraction of data being updated during the RMW operation. For updates that modify less than 25% of the tuple, $(MC)^2$ provides up to 78% higher throughput. For higher fraction of updates with one thread, the memory read penalty of $(MC)^2$ outpaces the copy speedup. With eight threads the transactions are memory bandwidth-bound, and as $(MC)^2$ reduces memory accesses it consistently provides performance improvement for update fractions less than 100%.

Figure 17 shows the transaction throughput when the update operations are write-only. The throughput mimics that of RMW because cache write misses issue read-for-ownership (RFO) [22] that reads data from memory before writing to the cache, incurring $(MC)^2$'s read penalty. If the store operations are replaced by non-temporal stores [22] that avoid RFO, we see that $(MC)^2$ is able to provide a higher throughput with one thread than the baseline for all write fractions. Similar to the previous case, for 8 threads the application becomes memory bandwidth bound, leading to performance improvements until the entire tuple is modified. We were unable to compare to zIO as Cicada allocates memory using `MAP_SHARED`, which zIO does not support.

Concurrent snapshots with huge pages: In-memory databases make use of virtual memory snapshotting [29] to take concurrent database snapshots. They leverage the `fork` system call to create a virtual memory snapshot in a child process. To minimize overhead, `fork` does not copy memory to create the snapshot. Instead, the parent and child map the same memory as copy-on-write, copying memory pages lazily on writes [48].

With their large datasets, in-memory databases would like to use huge page mappings to minimize TLB misses. However, despite the use of huge pages reducing the direct overhead of `fork` by over an order of magnitude due to smaller copied page tables [63], huge page copy-on-write faults require larger 2MB copies, causing significant latency spikes. For this reason, many in-memory databases advise against huge pages [30, 45].

To demonstrate how $(MC)^2$ can mitigate this issue, we modify the Linux kernel's `copy_user_huge_page` function to use `MCLAZY` instead of copying the huge page immediately.

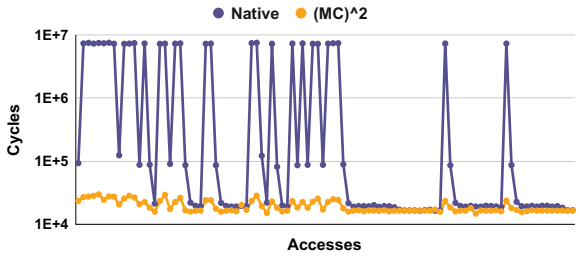


Fig. 18: Write latencies with huge page copy-on-write.

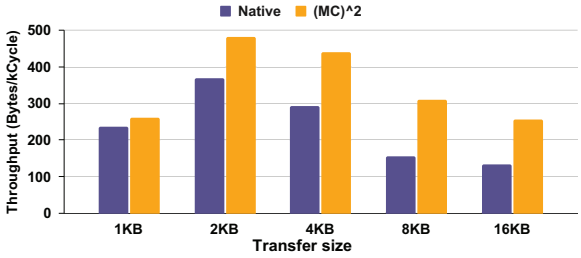


Fig. 19: Linux pipe transfer throughput.

We run a program that initializes a 64MB memory region using huge pages, calls `fork`, and then updates random 8-byte elements in the 64MB region. We measure the latency of each update using the `RDTSC` [22] instruction.

Figure 18 shows the latencies of the first 100 accesses, where Native is the unmodified Linux kernel, and $(MC)^2$ uses our modified kernel. The native kernel experiences latency spikes up to $455\times$ during page faults. $(MC)^2$ experiences spikes at most $2\times$ with worst-case latencies $250\times$ lower than native. $(MC)^2$ reduces latency spikes caused by huge page copy-on-write faults by over two orders of magnitude, while retaining the benefits of huge pages.

User-kernel buffer copies: Copies between user and kernel buffers upon system calls incur overheads across many application domains. For example, cloud platforms heavily use the POSIX socket interface for communication. Socket calls (e.g., `send`/`recv`) involve buffer copies that are exchanged with the NIC via DMA [28]. Similarly, inter-process communication like pipes and Unix sockets involve kernel buffer copies.

We modify the Linux kernel functions `pipe_write` and `pipe_read` to make use of lazy copies instead of copying data to/from a kernel buffer. We measure the latency of transfer when a process sends data to another with these pipes using `RDTSC`, then report the throughput in bytes/kilocycle.

Figure 19 shows the throughput for different transfer sizes. For smaller sizes, the overhead of the system call dominates over the actual data transfer time, leading to $(MC)^2$ having a small improvement in throughput. As these transfer sizes increase, the throughput saturates with $(MC)^2$ providing roughly double the throughput of the native kernel.

	# of CTT entries (access latency)		
Copy threshold	1024 (0.65ns)	2048 (0.75ns)	4096 (0.98ns)
10%	50.8	50.8	51.0
25%	51.0	50.8	51.0
50%	51.2	50.8	51.0
75%	52.2	51.2	51.2
90%	53.4	51.6	51.6

(a) Wall clock execution time in milliseconds.

	# of CTT entries (access latency)		
Copy threshold	1024 (0.65ns)	2048 (0.75ns)	4096 (0.98ns)
10%	1%	0%	0%
25%	4%	0%	0%
50%	20%	0%	0%
75%	58%	5%	0%
90%	100%	30%	12%

(b) Max-min normalized stalls due to full CTT.

Fig. 20: Protobuf performance, varying the number of CTT entries and copy threshold.

C. Sensitivity studies

We now examine the impact of the different parameters of the Copy Tracking Table and Bounce Pending Queue.

Copy Tracking Table (CTT): To identify the impact of the number of CTT entries and the threshold at which we start asynchronously freeing entries, we run Protobuf with a sweep of various CTT sizes and thresholds, shown in Figure 20. The performance difference between worst and best configuration is around 5%, showing that varying the CTT parameters does not drastically impact performance. When the CTT is small (1,024 entries), the CPU frequently suffers from stalls due to the CTT being full (Figure 20b), negatively impacting performance. We see a similar behavior with a high (i.e., 90%) copy threshold.

Interestingly, reducing the copy threshold does not negatively impact performance. This is because $(MC)^2$ limits the outstanding asynchronous copies per memory controller, restricting the memory bandwidth interference with the CPU. However, a too small copy threshold leads to unnecessary copying and underutilizes the CTT. We find that 2,048 entries with a 50% copy threshold provides a small CTT without stalls.

Bounce Pending Queue (BPQ): To identify the performance impact of differing BPQ sizes, we make use of a microbenchmark that lazily copies a source buffer to a destination buffer, overwrites the source buffer, then flushes the writes from the cache. The microbenchmark then executes a fence operation, bringing the overhead of writing to the source buffer into the critical path. We repeat this experiment for varying buffer sizes.

Figure 21 shows the normalized runtime with different BPQ sizes. A small BPQ fills up quickly, leading to stalled writes. As the BPQ size increases, more writes are able to proceed in parallel, reducing the runtime. Enlarging the BPQ gradually receives diminishing returns, with 16 BPQ entries

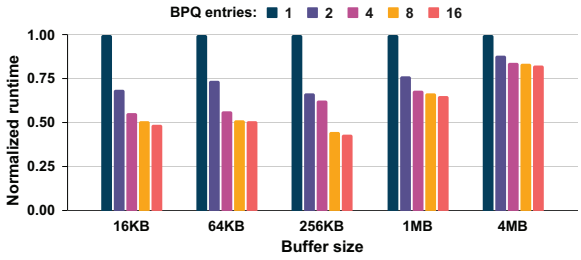


Fig. 21: Normalized runtime when source buffers are written to with varying number of BPQ entries.

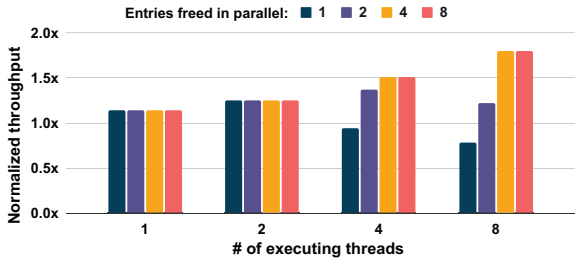


Fig. 22: MVCC database speedup with $(MC)^2$, varying the number of per-MC parallel CTT entry frees.

only providing on average 2% speedup over 8 entries, compared to 2 entries providing 35% speedup over 1 entry.

Scalability: Modern servers provide a considerable number of cores, supporting a large number of parallel threads. As the number of threads executing lazy memcpy operations increases, the CTT can quickly fill, causing stalls while waiting for entries to be freed (§III-A1).

To resolve this, the CTT frees multiple entries in parallel. To demonstrate this, we run the MVCC database with increasing numbers of threads, while varying the number of CTT entries being freed in parallel per memory controller. We examine the throughput obtained, normalized to standard memcpy using the same number of threads (Figure 22). For small thread counts, the performance is stable as the rate of lazy copies is not enough to fill the CTT. For larger thread counts, non-parallel freeing ends up suffering due to stalls. Increasing parallelism improves performance and prevents these stalls. Parallel CTT freeing increases the memory bandwidth utilization. However, servers provision memory bandwidth proportional to cores [42], allowing $(MC)^2$ to scale to larger servers.

VI. Related Work

We are not the first to consider the impact of memcpy in application performance. We review related work here.

Cache-based lazy memcpy: The closest line of work [15, 16, 54, 56] proposes a memcpy accelerator within the cache. This accelerator adds a mapping table to the cache that remaps destination buffer requests to the source buffer. However, the high performance of this accelerator is contingent on the source data being present in the cache. Otherwise, it must still be fetched from memory.

Copy engines: Many proposals [10, 23, 37, 51, 52, 65] present copy engines to improve bandwidth and reduce CPU data movement overhead. Asynchrony is often used to provide high performance, where the CPU initiates the copy engine and performs other compute while waiting for copy completion. Many of these proposals have high initialization overhead, making them impractical for kilobyte-sized copies.

We view these proposals as addressing an orthogonal problem to ours. $(MC)^2$ eliminates unnecessary data movement and removes copies from the critical path, delaying them to access time where copy latencies can be hidden. $(MC)^2$ could make use of copy engines to start asynchronously moving data on lazy memcpy calls, while access to uncopied data follows the usual $(MC)^2$ procedure, providing fully asynchronous copies transparent to the CPU.

In-DRAM copies: A tangential line of work [8, 18, 46] explores in-DRAM copy techniques. These take advantage of high-bandwidth internal links present in DRAM to perform copy operations fully within the DRAM module, without needing to move data across the memory interconnect. These proposals require the source and destination buffers to be present within the same DRAM module.

Application-specific memcpy elision: Many proposals target copy overheads in specific application domains. S. Karandikar et. al. [27] propose a hardware accelerator for Protobuf operations. Several works [43, 44, 55] target reducing copy overheads in serialization. zIO [49] eliminates redundant memory copies present in IO-based application and OS operations. Contrary to these, we seek to provide a general-purpose solution across various application domains.

VII. Conclusion

Data movement is a significant CPU overhead in modern applications. We propose $(MC)^2$, a hardware mechanism that enables lazy memcpy operations. $(MC)^2$ reduces copy overhead in the critical path. We evaluate $(MC)^2$ using gem5 and show that it provides 43% speedup for Google's Protobuf workload and 250× lower latency for huge page copy-on-write faults.

Appendix

A. Abstract

We provide the source code and setup necessary for $(MC)^2$: Lazy MemCopy at the Memory Controller. $(MC)^2$ is a hardware extension that provides support for a lazy memcpy operation.

This operation avoids copying data at the time of function call. Instead, if copied destinations are later accessed, $(MC)^2$ uses tracking information to seamlessly reroute the request to the appropriate source, while lazily executing copies only when necessary. $(MC)^2$ modifies the memory controller and has been implemented using gem5, a CPU simulator.

This artifact consists of the source code of the simulator, benchmarks used for evaluation and all scripts needed to replicate the figures in the paper.

B. Artifact check-list (meta-information)

- **Compilation:** GCC 9 or higher.
- **Binary:** All required binaries are included.
- **Data set:** Scripts are provided to generate necessary datasets.
- **Run-time environment:** Simulator can be run on an x86 machine. We evaluated on a machine with Ubuntu 22.04.
- **Hardware:** An x86 machine with at least 100 GB of disk space that supports KVM.
- **Output:** Text files containing the summarized results are generated as well as PNG files of the graphs in the Evaluation section (§V). Raw performance numbers can be found in the results folder as well.
- **How much disk space required (approximately)?:** 100GB..
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes
- **How much time is needed to complete experiments (approximately)?:** 42 hours
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** 10.5281/zenodo.10884322

C. Description

The artifact contains the source code of $(MC)^2$ along with all evaluated benchmarks and datasets. This allows for reproducing figures 10 - 21 contained in §V.

1) How to access: The artifact can be downloaded from https://github.com/AKKamath/MCSquare-ISCA24 or https://zenodo.org/doi/10.5281/zenodo.10884322.

2) Hardware dependencies: The artifact requires an x86 machine with around 100 GB of free disk space that supports KVM. To see if your CPU supports KVM run:

```
egrep -c '(vmx|svm)' /proc/cpuinfo
```

If it returns 0, your processor does not support KVM. If the command returns 1 or more, your processor supports KVM.

3) Software dependencies: The gem5 simulator requires either Ubuntu 20.04 or 22.04. Root privilege is required to run the experiments. Detailed instructions on how to build can be found here: https://www.gem5.org/documentation/general_docs/building. This page also contains Docker Images with all dependencies already installed.

For Ubuntu 22.04, the following installs all dependencies:

```
sudo apt install build-essential git m4 scons \
    zlib1g zlib1g-dev libprotobuf-dev python3-dev \
    protobuf-compiler libprotoc-dev qemu-kvm \
    libvirt-daemon-system libgoogle-perftools-dev \
    libboost-all-dev pkg-config python3-tk \
    libvirt-clients bridge-utils unzip wget \
    python3-matplotlib python3-numpy
```

D. Installation

The artifact can be built using the following Linux commands:

```
sudo adduser `id -un` libvirt # FOR KVM
sudo adduser `id -un` kvm     # FOR KVM
unzip MCSquare-AE.zip -d mcsquare_ae
cd mcsquare_ae
scons build/X86/gem5.opt -j ${CPUS}
```

${CPUS} is the number of threads to use to build the simulator. A single-threaded build takes around 2 hours.

E. Experiment workflow

Most of the folders contained in the repository are for the gem5 simulator. The relevant files and folders specific to $(MC)^2$ are contained in a folder called "mcsquare/". In this folder, a Makefile is provided which contains all the commands necessary to run the different experiments. The scripts/ directory contains all the scripts used with gem5 to run specific experiments, which are called by the Makefile. These have been organized into folders based on their benchmark. The os/ directory contains the disk and kernel images used by the simulator. On running experiments, a results/ folder will be created within the mcsquare/ directory which will contain all the raw results from the experiments. A figures/ folder will be created on completion of experiments, which shall contain the final plotted figures generated from the results.

The experiments can be launched in parallel and run in the background, to reduce overall time for simulation. If experiments take much longer than the listed time, it's likely the simulator hung during launch and the experiment should be relaunched.

The following commands can be executed within the "mcsquare/" folder to generate the different results:

```
make launch_micro_latency   #Figure 10:      10 min
make launch_micro_breakdown #Figure 11:      10 min
make launch_micro_seq       #Figure 12:      30 min
make launch_micro_rand      #Figure 13:       1 hr
make launch_protobuf        #Figure 14,20:    2 hr
make launch_mongo           #Figure 15:      15 hr
make launch_mvcc            #Figure 16a,17a: 10 hr
make launch_mvcc_8T         #Figure 16b,17b: 10 hr
make launch_hugepage_access #Figure 18       20 min
make launch_pipe            #Figure 19:      15 min
make launch_src_write       #Figure 21:      10 min
make launch_ctt_free        #Figure 22:       2 hr
```

The commands require sudo priviledges, and the account password will be asked when the command is run.

F. Evaluation and expected results

For each key result, a tab-separated result .TXT file and a .PNG graph are generated. The results/ folder contains all generated tab-separated text files with filenames figureX.txt. The figures/ folder contains the PNG graphs with filenames figureX.png, where X is the figure number. The exception is Figure 20 where only a .TXT file containing the result table is outputted. These outputs can be matched against figures reported in the paper. Minor variances in performance numbers occur from run to run, but general trends should remain stable.

REFERENCES

[1] userfaultfd(2). http://man7.org/linux/man-pages/man2/userfaultfd.2.html, February 2020.

[2] pickle — python object serialization. https://docs.python.org/3/library/pickle.html, October 2023.

[3] Protocol buffers. https://protobuf.dev/, October 2023.

[4] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. Cacti 7: New tools for interconnect exploration in innovative

off-chip memories. *ACM Trans. Archit. Code Optim.*, 14(2), jun 2017.

[5] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, mar 2017.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011.

[7] Kevin K. Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Understanding latency variation in modern dram chips: Experimental characterization, analysis, and optimization. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS '16, page 323–336, New York, NY, USA, 2016. Association for Computing Machinery.

[8] Kevin K. Chang, Prashant J. Nair, Donghyuk Lee, Saugata Ghose, Moinuddin K. Qureshi, and Onur Mutlu. Low-cost inter-linked subarrays (lisa): Enabling fast inter-subarray data movement in dram. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 568–580, 2016.

[9] Guillaume Chatelet, Chris Kennelly, Sam (Likun) Xi, Ondrej Sykora, Clément Courbet, Xinliang David Li, and Bruno De Backer. Automemcpy: A framework for automatic generation of fundamental memory operations. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2021, page 39–51, New York, NY, USA, 2021. Association for Computing Machinery.

[10] Zhenke Chen, Dingding Li, Zhiwen Wang, Hai Liu, and Yong Tang. Ramci: a novel asynchronous memory copying mechanism based on i/oat. *CCF Transactions on High Performance Computing*, 3:129–143, 2021.

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *2010 ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[12] Crucial. Ddr5 memory: Everything you need to know. https://www.crucial.in/articles/about-memory/everything-about-ddr5-ram, 2023.

[13] Arnaldo Carvalho De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, pages 1–42, 2010.

[14] A Micro Devices. Amd64 architecture programmer's manual volume 2: System programming. *AMD64 Architecture Programmer's Manual*, 2024.

[15] Filipa Duarte and Stephan Wong. A memcpy hardware accelerator solution for non cache-line aligned copies. In *2007 IEEE International Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, pages 397–402, 2007.

[16] Filipa Duarte and Stephan Wong. Cache-based memory copy hardware accelerator for multicore systems. *IEEE Transactions on Computers*, 59(11):1494–1507, 2010.

[17] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 727–741, New York, NY, USA, 2023. Association for Computing Machinery.

[18] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. Computedram: In-memory compute using off-the-shelf drams. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 100–113, New York, NY, USA, 2019. Association for Computing Machinery.

[19] Google. Fleetbench. https://github.com/google/fleetbench, 2023.

[20] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, sep 1989.

[21] Intel. Intel® optane™ persistent memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[22] Intel. Intel® 64 and ia-32 architectures software developer's manual, 2011.

[23] Xiaowei Jiang, Yan Solihin, Li Zhao, and Ravishankar Iyer. Architecture support for improving bulk memory copying and initialization performance. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 169–180, 2009.

[24] Tim Kaler, Nickolas Stathas, Anne Ouyang, Alexandros-Stavros Iliopoulos, Tao Schardl, Charles E. Leiserson, and Jie Chen. Accelerating training and inference of graph neural networks with fast sampling and pipelining. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 172–189, 2022.

[25] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 158–169, New York, NY, USA, 2015. Association for Computing Machinery.

[26] Giorgos Kappes and Stergios V. Anastasiadis. Asterope: A cross-platform optimization method for fast memory copy. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*, PLOS '21, page 9–16,

New York, NY, USA, 2021. Association for Computing Machinery.

[27] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. A hardware accelerator for protocol buffers. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 462–478, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 67–81, New York, NY, USA, 2016. Association for Computing Machinery.

[29] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206, 2011.

[30] KeyDB. Troubleshooting latency issues. https://docs.keydb.dev/docs/latency/.

[31] Scott T. Leutenegger and Daniel Dias. A modeling study of the tpc-c benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 22–31, New York, NY, USA, 1993. Association for Computing Machinery.

[32] 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. 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, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[33] Liang Li, Guoren Wang, Gang Wu, Ye Yuan, Lei Chen, and Xiang Lian. A comparative study of consistent snapshot algorithms for main-memory database systems. *IEEE Transactions on Knowledge and Data Engineering*, 33(2):316–330, 2021.

[34] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 21–35, New York, NY, USA, 2017. Association for Computing Machinery.

[35] Compute Express Link. Compute express link™: The breakthrough cpu-to-device interconnect cxl. https://www.computeexpresslink.org/.

[36] Sihang Liu, Korakit Seemakhupt, Gennady Pekhimenko, Aasheesh Kolli, and Samira Khan. Janus: Optimizing memory and storage support for non-volatile memory systems. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 143–156, New York, NY, USA, 2019. Association for Computing Machinery.

[37] Howard Mao, Randy H Katz, and Krste Asanović. Hardware acceleration for memory to memory copies. *Master's thesis*, 2017.

[38] MongoDB. Mongodb. github.com/mongodb/mongo/blob/v4.4/.

[39] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. Enabling practical processing in and near memory for data-intensive computing. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[40] Samuel Naffziger, Kevin Lepak, Milam Paraschou, and Mahesh Subramony. 2.2 amd chiplet architecture for high-performance server and desktop products. In *2020 IEEE International Solid-State Circuits Conference - (ISSCC)*, pages 44–45, 2020.

[41] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: X86-tso. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, page 391–407, Berlin, Heidelberg, 2009. Springer-Verlag.

[42] Vinicius Petrucci, Eishan Mirakhur, Nikesh Agarwal, Su Wei Lim, Vishal Tanna, Rita Gupta, and Mahesh Wagh. Cxl memory expansion: A closer look on actual platform. https://www.micron.com/content/dam/micron/global/public/products/white-paper/cxl-memory-expansion-a-close-look-on-actual-platform.pdf, 2023.

[43] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. Breakfast of champions: Towards zero-copy serialization with nic scatter-gather. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 199–205, New York, NY, USA, 2021. Association for Computing Machinery.

[44] Deepti Raghavan, Shreya Ravi, Gina Yuan, Pratiksha Thaker, Sanjari Srivastava, Micah Murray, Pedro Henrique Penna, Amy Ousterhout, Philip Levis, Matei Zaharia, and Irene Zhang. Cornflakes: Zero-copy serialization for microsecond-scale networking. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 200–215, New York, NY, USA, 2023. Association for Computing Machinery.

[45] Redis. Optimizing redis: Diagnosing latency issues. https://redis.io/docs/management/optimization/latency/.

[46] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, page 185–197, New York, NY, USA, 2013. Association for Computing Machinery.

[47] Richard L Sites. Fast memcpy, a system design. https://www.sigarch.org/fast-memcpy-a-system-design/, Dec

2022.

[48] Jonathan M. Smith and Gerald Q Maguire Jr. Effects of copy-on-write memory management on the response time of unix fork operations. *Computing Systems*, 1(3):255–278, 1988.

[49] Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. zIO: Accelerating IO-Intensive applications with transparent Zero-Copy IO. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 431–445, Carlsbad, CA, July 2022. USENIX Association.

[50] Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. zio's copy interposer for mongodb. https://github.com/tstamler/zIO/blob/mongo/tas/lib/sockets/copy_interpose.c, 2022.

[51] Wen Su, Ling Wang, Menghao Su, and Su Liu. A processor-dma-based memory copy hardware accelerator. In *2011 IEEE Sixth International Conference on Networking, Architecture, and Storage*, pages 225–229, 2011.

[52] Karthikeyan Vaidyanathan, Wei Huang, Lei Chai, and Dhabaleswar K Panda. Designing efficient asynchronous memory operations using hardware copy engine: A case study with i/oat. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8. IEEE, 2007.

[53] Kenton Varda. Protocol buffers: Google's data interchange format. *Google Open Source Blog, Available at least as early as Jul*, 72:23, 2008.

[54] Stamatis Vassiliadis, Filipa Duarte, and Stephan Wong. A load/store unit for a memcpy hardware accelerator. In *2007 International Conference on Field Programmable Logic and Applications*, pages 537–541, 2007.

[55] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. Zerializer: Towards zero-copy serialization. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 206–212, New York, NY, USA, 2021. Association for Computing Machinery.

[56] Stephan Wong, Filipa Duarte, and Stamatis Vassiliadis. A hardware cache memcpy accelerator. In *2006 IEEE International Conference on Field Programmable Technology*, pages 141–148, 2006.

[57] Wenchao Wu, Xuanhua Shi, Ligang He, and Hai Jin. Turbognn: Improving the end-to-end performance for sampling-based gnn training on gpus. *IEEE Transactions on Computers*, 72(9):2571–2584, 2023.

[58] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10(7):781–792, mar 2017.

[59] Sujay Yadalam, Nisarg Shah, Xiangyao Yu, and Michael Swift. Asap: A speculative approach to persistence. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 892–907, 2022.

[60] Huaisheng Ye. Introduction to 5-level paging in 3rd gen intel xeon scalable processors with linux. *Lenovo Press*, 2021.

[61] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 195–211, New York, NY, USA, 2021. Association for Computing Machinery.

[62] Jialiang Zhang, Michael Swift, and Jing (Jane) Li. Software-defined address mapping: A case on 3d memory. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 70–83, New York, NY, USA, 2022. Association for Computing Machinery.

[63] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. On-demand-fork: a microsecond fork for memory-intensive and latency-sensitive applications. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 540–555, New York, NY, USA, 2021. Association for Computing Machinery.

[64] Kaiyang Zhao, Kaiwen Xue, Ziqi Wang, Dan Schatzberg, Leon Yang, Antonis Manousis, Johannes Weiner, Rik Van Riel, Bikash Sharma, Chunqiang Tang, and Dimitrios Skarlatos. Contiguitas: The pursuit of physical memory contiguity in datacenters. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[65] Li Zhao, Laxmi N. Bhuyan, Ravi Iyer, Srihari Makineni, and Donald Newell. Hardware support for accelerating data movement in server platform. *IEEE Transactions on Computers*, 56(6):740–753, 2007.

[66] Li Zhao, R. Iyer, S. Makineni, L. Bhuyan, and D. Newell. Hardware support for bulk data movement in server platforms. In *2005 International Conference on Computer Design*, pages 53–60, 2005.

[67] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. Distributed hybrid cpu and gpu training for graph neural networks on billion-scale heterogeneous graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '22, page 4582–4591, New York, NY, USA, 2022. Association for Computing Machinery.