How to Copy Memory? Coordinated Asynchronous Copy as a First-Class OS Service

Jingkai He[1,2], Yunpeng Dong[1], Dong Du[1,2], Mo Zou[3], Zhitai Yu[3], Yuxin Ren[3], Ning Jia[3], Yubin Xia[1,2], Haibo Chen[1,2]

[1]Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

[2]Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

[3]Huawei Technologies Co., Ltd.

Abstract

In modern systems, memory copy remains a critical performance bottleneck across various scenarios, playing a pervasive role in system-wide execution such as syscalls, IPC, and user-mode applications. Numerous efforts have aimed at optimizing copy performance, including zero-copy with page remapping and hardware-accelerated copy. However, they typically target specific use cases, such as Linux zero-copy send() for messages of ≥10KB. This paper argues for *copy as a first-class OS service*, offering three key benefits: (1) with the asynchronous copy abstraction provided by the service, applications can overlap their execution with copy; (2) the service can effectively utilize hardware capabilities to enhance copy performance; (3) the service's global view of copies further enables holistic optimization. To this end, we introduce Copier, a new OS service of *coordinated asynchronous copy*, to serve both user-mode applications and OS services. We build Copier-Linux to demonstrate Copier's ability to improve performance for diverse use cases, including Redis, Protobuf, network stack, proxy, etc. Evaluations show that Copier achieves up to a 1.8× speedup for real-world applications like Redis and a 1.6× improvement over zIO, the state-of-the-art in optimizing copy efficiency. To further facilitate adoption, we develop a toolchain to ease the use of Copier. We also integrate Copier into a commercial smartphone OS (HarmonyOS 5.0), achieving promising results.

CCS Concepts: • **Software and its engineering → Operating systems**; *Memory management*; • **Computer systems organization → Multicore architectures**.

SOSP '25, October 13–16, 2025, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1870-0/2025/10…$15.00

https://doi.org/10.1145/3731569.3764800

Keywords: Memory Copy, Asynchronous Copy, OS Service

ACM Reference Format:

Jingkai He, Yunpeng Dong, Dong Du, Mo Zou, Zhitai Yu, Yuxin Ren, Ning Jia, Yubin Xia, and Haibo Chen. 2025. How to Copy Memory? Coordinated Asynchronous Copy as a First-Class OS Service. In *ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3731569.3764800

1 Introduction

Memory copy remains pervasive across modern OSes [1–6] and apps [7]. It can introduce significant performance costs for memory-intensive and I/O-intensive workloads like KV store [7, 8], proxies [9, 10], and storage services [11, 12], and is a common performance tax for various apps and OS services [13]. One example is inter-process communication (IPC), where message-passing often becomes the primary bottleneck [14], typically requiring one or more copies. Syscalls constitute another example where cross-privilege switching entails copy for security reasons and semantic gaps. Despite providing an easy-to-use abstraction, copy-based data movement is known for its inefficiency [15–19], blocking computation as shown in Fig. 1-a.

Numerous efforts have been made to mitigate costs caused by copy. One approach leverages hardware features, e.g., SIMD instructions [20, 21] and on-chip DMA [1, 22], to boost copy performance. However, a significant limitation is that the hardware capabilities usually *cannot be fully utilized*. For example, while libc memcpy() adopts SIMD (e.g., x86 AVX [21, 23] and ARM SVE/NEON [20]) for better performance, the Linux kernel does not utilize SIMD because of the high costs of saving and restoring register states (up to several KB). Conversely, while the kernel can utilize DMA to improve copy performance and reduce CPU costs [1], such privileged features can hardly be utilized by user apps.

Another common approach, zero-copy, reduces copy costs through memory sharing or remapping, as shown in Fig. 1-b. However, state-of-the-art zero-copy methods face several limitations for common cases, including page alignment requirements, lack of support for scenarios requiring multiple replicas (e.g., CoW handler), and the susceptibility to Time-of-check to Time-of-use (TOCTTOU) attacks [14]. Meanwhile,

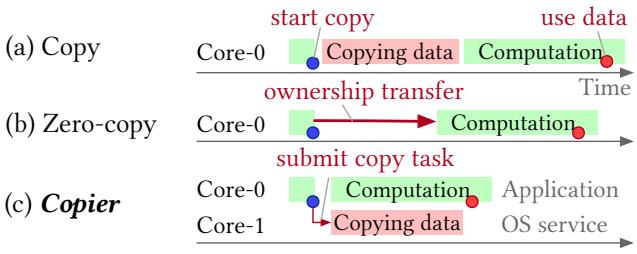


Figure 1. Comparison among different copy methods. *Copier achieves both low latency and an easy-to-use abstraction by reorganizing tasks among processing units.*

since the operations of remapping or sharing incur extra costs, the performance benefits are conditional. For example, the Linux kernel provides zero-copy send() and clarifies that it can only benefit messages ≥10KB [24]. Similarly, zIO provides an on-demand way to achieve zero-copy, but requires copy sizes ≥16KB to bring benefits [7].

This paper argues the necessity of *making copy a first-class OS service* (like file systems and network), for three reasons. First, **asynchronous copy with the OS service**. Our study reveals that there is often a sufficient time window between data being copied and used (§3), a period we refer to as Time-of-Copy to Time-of-Use (Copy-Use) window. As shown in Fig.1-c, the OS service can leverage the window to hide copy latency while not blocking application's execution. Second, **effective utilization of under-utilized hardware**. As an OS service, it can fully utilize hardware features like SIMD and DMA, and provide the "best" copy performance to apps and OS services. Third, **holistic optimization of copy tasks**. The service has a global view of the copy requests, and can bring holistic optimizations like absorbing unnecessary intermediate copies and scheduling high-priority ones first.

We design Copier embodying the idea of *copy as a first-class OS service*. Copier provides interfaces for both user apps and kernel services, addressing the following challenges.

Challenge-1 (from sync to async): How to fully exploit the performance benefits of asynchronous copy while maintaining the semantics of synchronous copy? Async copy can utilize the Copy-Use window to overlap data copy and use. However, the performance gains are limited by (1) the costs of submitting copy tasks for small copies, and (2) the blocking latency to wait for the completion of large copies. Besides, async copy complicates the programming model and may impose a burden on developing or porting apps.

Challenge-2 (from function to OS service): How to fully utilize hardware features and leverage the global view for optimization with minimal extra cost? As an OS service, Copier can exploit both user-space and kernel-space hardware to accelerate copy. With the global view of copies, Copier can eliminate unnecessary intermediate copies. However, they are not free lunches: (1) Heterogeneous copy units have asymmetric performance, e.g., although DMA copy does not consume CPU cycles, it is slower than AVX and wastes cycles to submit DMA tasks and wait for completion. (2) Eliminating copy based on global view is non-trivial, as forming the global view requires correctly tracking dependency among copies across privilege levels.

Challenge-3 (from single to multiple clients): How to ensure resource isolation, fairness, and correctness? Traditional sync copy (via functions) uses clients' own time slices, simplifying the fairness and isolation among clients. As an OS service, however, clients may contend for Copier's service for copy. A further complication arises from the coordination with other OS subsystems, e.g., memory management supports features like CoW and on-demand paging, which may trigger page faults during copy that Copier must properly handle in its own address space and context.

To address the challenges, we propose the following techniques. First, we design queue-based **Copier abstractions**. Different from prior kernel abstractions [25–27], which update a task's status only after completion, and execute tasks in a determined order, Copier abstractions highlight fine-grained update of task status, and out-of-order execution. Fine-grained update enables apps to use data without waiting for the entire copy to complete, forming copy-use pipelines. Out-of-order execution allows users to adjust the execution order of tasks as needed, solving head-of-line blocking. We introduce *csync* primitive to enable apps to check and control the progress of copy, and formally verify the semantic equivalence between async copy with *csync* and sync copy.

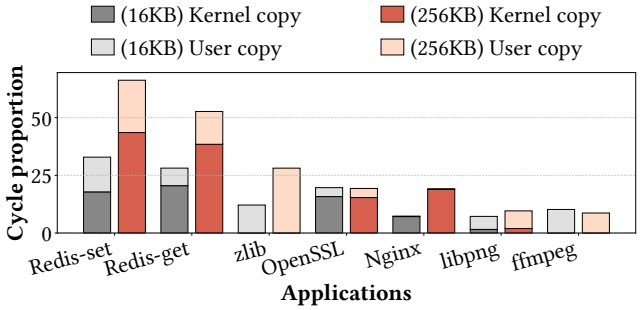
Second, we design the **piggyback-based dispatcher** to harmonize heterogeneous copy units. It piggybacks DMA tasks onto AVX tasks and executes them in parallel, overlapping DMA copy with AVX copy without wasting cycles on waiting for DMA to finish. Different from traditional dispatchers [1, 28] partitioning a single copy to run on different units, which can only optimize large copies, Copier leverages the queuing nature of async tasks to fuse several copies and execute them in the same round of piggybacking, benefiting relatively small copies. Copier supports efficient tracking of copy dependency by utilizing system events (e.g., syscall trap and return) as indicators, thereby realizing **copy absorption**, which merges redundant copies.

Last, recognizing copy can be treated as a resource, we design **Copier scheduler** and a cgroup extension taking *copy length* as the resource unit to ensure fairness and isolation. We propose *proactive fault handling* to proactively identify and handle faults, instead of relying on hardware faults.

We design and implement Copier-Linux utilizing Copier to improve the performance of apps and key kernel services. We also integrate Copier into a commercial smartphone OS (HarmonyOS 5.0 [4, 35]) experimentally. Evaluation shows promising results. Specifically, for Redis [8], Copier achieves a 1.8× speedup in latency compared with baseline, and a 1.6× speedup over zIO [7], the SOTA work in optimizing

Table 1. Overview of systems with copy optimizations. *W/o alignment*: whether the system requires buffers to be page-aligned. *Cross privilege/address space*: whether the system can optimize cross-privileged/cross-address-space copies. *No blocking*: whether copies will not block execution. *Absorb copy*: whether the system can eliminate unnecessary copies.

Systems			Usability				Performance		
Type	**Name**	**Insight**	**Target scenario**	**W/o alignment**	**Cross privilege**	**Cross addr. space**	**Hardware features**	**No blocking**	**Absorb copy**
Baseline	U-mode memcpy	New hardware features	Apps	✓	✗	✗	SIMD [23]	✗	✗
	K-mode memcpy	Hardware w/o state costs	Kernel	✓	✓	✓	ERMS [29]	✗	✗
Copy optimization	L4 [30]	Temporary mapping	IPC	✓	✗	✓	Page table	✗	Partial
	Zero-copy socket [24]	Shared pages	≥10KB/OS socket	✗	✓	✗	Page table	✓	✗
	Linux sendfile [31]	Address transfer in kernel	Copying files	✓	✓	✗	CPU	✗	Partial
	Splice/vmsplice [32]	Page moving (no copy)	Pipe	✗	✓	✓	Page table	✓	✗
	Arrakis [33]	Hardware demultiplexing	I/O	✓	✓	✗	SR-IOV	✓	✗
	XRP [34]	Offloading logic to kernel	Storage	✓	✓	✗	CPU	✗	✓
	zIO [7]	No unnecessary copy	Copy ≥16KB	Partial	✗	✗	CPU	✓	✓
	Fastmove [1]	DMA to save costs	NVM Storage (OS)	✓	✓	✓	DMA	✗	✗
	Copier	Async copy w/ OS service	**Kernel/Apps ≥0.5KB**	✓	✓	✓	SIMD+DMA	✓	✓



(a) Cycle proportion of copy in Linux Apps

Apps	Camera recording	Music playing	Home swiping	Keyboard input	Online meeting	Browser I/O
Prop.	6%–16%	4%–15%	12%–19%	3%–15%	4%–8%	49%

(b) Cycle proportion of copy on HarmonyOS

Figure 2. Analysis of copy across apps and OSes.

copy efficiency. On phones, Copier reduces video decoding latency by up to 10%. Copier also performs well in boosting the performance of send() and recv() syscalls, Protobuf [36], Binder IPC [37], CoW fault handling, etc.

2 Systematic Analysis of Copy in the Wild

2.1 Copy Performance is Still Critical

Copy has been studied for decades; however, its performance remains critical for today's apps and OSes.

Quantitative Analysis. We measure the cycle proportion of copy for widely used apps and libraries (on Linux), as shown in Fig. 2-a[1]. Although they are carefully optimized, copy remains their main performance bottleneck, consuming up to 66.2% of cycles. The copy includes kernel-mode copy and user-mode copy. Copy is also the major bottleneck of key scenarios on HarmonyOS 5.0 (smartphone), as shown in Fig. 2-b. *Copy can be a significant challenge for apps, especially for memory-intensive or I/O-intensive ones* (**Finding-1**).

Prevalence of copy. Today's copy acts as a common "performance tax" — although it may not be the top-ranking bottleneck for specific apps, optimizing copy is valuable from the whole system perspective, e.g., 4–5% of (whole) datacenter cycles are consumed by copy in Google datacenters [16, 17].

However, copy is not easy to be eliminated. To illustrate the challenges, we classify copy from a system perspective into two types: *intra-boundary copy* and *inter-boundary copy*. Boundary means the same address space and privilege level.

Although developers make significant efforts to mitigate unnecessary intra-boundary copies, some still remain for the following reasons: (1) *Organizing memory*, e.g., Redis copies the key and value to new buffers after parsing a SET request to avoid fragmentations caused by protocol and separators; (2) *Bridging semantic gaps*, e.g., userspace network stacks [43, 44] use copy to concatenate packets into a continuous message in recv(); (3) *Multi-replica requirement*, e.g., CoW handler creates multiple replicas of the page.

Inter-boundary copy can be further grouped into *cross-privilege copy* and *cross-address-space copy*, e.g., copies during recv() and IPC. Optimizations for these cases are harder compared with intra-boundary copy since OS support or app-OS co-design is usually necessary. *Optimizing copy is highly valuable for the whole system, but challenging* (**Finding-2**).

2.2 Existing Optimizations on Copy

Prior optimizations on copy [1, 7, 24, 30–34] still face challenging trade-offs. We present the comparison in Table. 1.

Copy with hardware features. Although hardware provides support to optimize copy performance, we observe that apps and the kernel usually cannot fully utilize these hardware features. SIMD extensions [45] like AVX [23] can move a large amount of data, which helps to accelerate copy.

[1] **Detailed experimental settings.** Fig. 2-a is conducted on Linux (5.15.131) servers and Fig. 2-b is conducted on smartphones, with settings detailed in §6. Redis SET/GET: Redis SET/GET commands; zlib: zlib [38] compressing strings; OpenSSL: OpenSSL [39] receiving encrypted messages (SSL_read(), AES-GCM encrypted); Nginx: Nginx [10] proxying between echo servers/-clients to forward messages; libpng: libpng [40] decoding PNG images stored in an ext4 file system; ffmpeg: ffmpeg [41] encoding videos with libx265 encoder. The cycle proportions of copies are measured with perf [42]. The 16/256KB specifically corresponds to value sizes, string lengths, encrypted message lengths, forwarded message lengths, image sizes, and video bitrates.

These features have been adopted by user-mode apps (e.g., glibc [21]). However, the Linux kernel still cannot utilize these features [46] because of the significant overhead of saving and restoring SIMD-related registers (the total size amounts to several KB). Another case is DMA, which can be utilized to copy data without CPU costs, e.g., Fastmove [1] utilizes Intel I/OAT [47] DMA to boost DRAM-NVM copy. However, user apps usually cannot easily benefit from DMA due to the requirement of privileged operations (e.g., MMIO).

Zero-copy. Prior efforts [48] exploit zero-copy methods, which usually utilize page remapping or shared memory. Linux vmsplice [32] and zero-copy socket [24] utilize page tables to share buffers between apps and OS. Zero-copy has its limitations. First, zero-copy-based kernel services have security issues as kernel operations on the buffer are visible to apps (potential TOCTTOU attacks [14, 49]). Second, shared buffers complicate memory ownership management, e.g., the app using zero-copy send() has to use additional syscalls to check the buffer's status, and ensure not to reuse it before the background transmission finishes. Third, zero-copy relying on remapping requires the data to be page-aligned, which is not feasible in many cases. Last, a common limitation for zero-copy is that it only supports one instance of data, and cannot support scenarios requiring multiple replicas.

zIO [7] utilizes a new observation that most copied data will not be accessed, thus the copy can be avoided and designs a page-fault-based on-demand copy method. It supports multiple data replicas; however, it still requires page table remapping, leading to non-trivial overheads. As a result, zIO requires ≥16KB [7] size to bring benefits. This is common for other zero-copy methods [24, 50], e.g., Linux zero-copy send() is suggested only for payload size ≥10KB [24]. Zero-copy is effective in scenarios with large copy (e.g., photo services [51]). However, in many scenarios the medium or small copies are the majority, e.g., our analysis of traces shows that 95.1% of Twitter memcached requests are ≤10KB [52], and 69.8% of AliCloud block service requests are 4KB–10KB [53].

Some systems leverage zero-copy (or remapping) to reduce copy times but still support multiple replicas, e.g., L4 [30] and Linux sendfile() [31]. They are designed only for specific scenarios and still require blocking to move data.

3 Copy as an OS Service

3.1 Observations and Insights

Copier is inspired by two insights. First, observing that traditional approaches supporting copy as library functions cannot fully utilize hardware capabilities, and that global management of copies brings holistic optimization opportunities. We claim that **copy should be treated as a first-class OS service** (**Insight-1**). The OS service can effectively resolve the challenges in existing works: (1) it can fully utilize hardware features. Apps and other OS services can benefit from the enhanced copy performance. (2) it has a global view of copies, which can be utilized to intelligently schedule copies and absorb unnecessary copies.

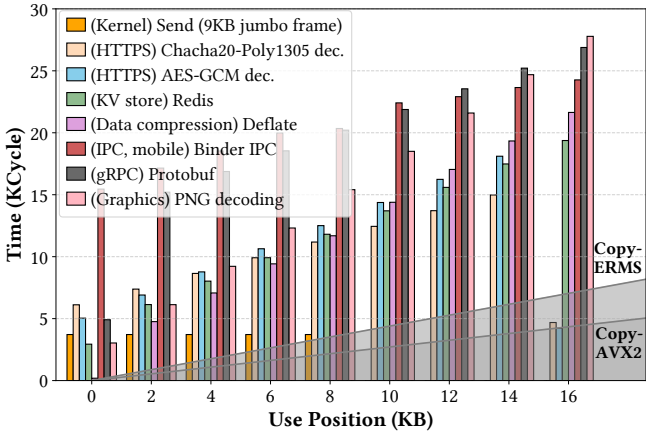


Figure 3. Copy-Use window and copy time. *ERMS (Enhanced REP MOVSB/STOSB) [29] is the kernel's copy method. The bars show the Copy-Use window of data at x-axis position. The shaded areas show the time needed to copy x-axis size.* ***The root cause of Copy-Use windows is that programs typically copy data in bulk but use it piece by piece in fixed patterns.*** *E.g., in Protobuf, after copying data to userspace in recv(), the CPU has to update socket status, return to userspace, initialize deserializing context and deserialize the previous 8KB before it accesses the byte at 8K. We obtain the data in the figure by modifying the apps and recording the timestamps.*

Second, we observe that a **time-of-copy to time-of-use (Copy-Use) window** is common in copy-intensive apps and kernel services, which can be utilized to hide copy latency without compromising correctness and security (**Insight-2**). The Copy-Use window refers to the time interval between the completion of the copy and the first use of the data. We evaluate the Copy-Use windows at different positions during data usage for representative apps and kernel services. Results (Fig. 3[2]) show that the Copy-Use window is mostly sufficient to cover the time needed for data copy, and is usually as high as 2–10x the time required for copy. The substantial time window provides us with the opportunity to overlap copy and computation, removing copy from the critical path. Combined with the first insight, apps and kernel services can non-blockingly (or asynchronously) submit a

[2]**Detailed experimental settings.** Send: sending 16KB messages using send() syscall (§6.1.2); Chacha20/AES dec.: receiving 16KB encrypted messages with OpenSSL (§6.2.3); Redis: Redis SETs with 16KB values (§6.2.1); Deflate: compressing 16KB strings with zlib (§6.2.3); Binder IPC: passing 16KB messages with Android Binder (§6.1.2); Protobuf: receiving 16KB serialized messages with protobuf (§6.2.3); PNG decoding: decoding 16KB images stored in an ext4 file system with libpng. For Send, we study the copy from userspace to kernel; for Chacha20/AES dec., Redis, and Protobuf, we study the copy in recv(); for PNG decoding, we study the copy in read(); for Deflate, we study its internal userspace copy; for Binder, we study the kernel copy in Binder driver. The environment settings are detailed in §6.

copy task to the global service (called Copier), and only synchronize the status before use.

3.2 Copier Overview

Based on these insights, we present Copier, an OS service for async copy, with the following designs.

Abstractions. Copier provides queue-based abstractions. To make the most of Copy-Use windows, Copier achieves fine-grained update of copy status with segment-based copy. To mitigate head-of-line blocking, Copier supports out-of-order execution with task promotion, which enables apps to raise the priority of some async copies as needed (§4.1). To ensure the correctness of async copies, and form the global view, Copier tracks the order dependency of copies with cross-queue barriers, which synchronize the queues separated by privilege levels, and utilizes system events as barrier indicators. Copier also tracks the data dependency of copies to maintain correctness of the reordered tasks (§4.2).

Optimizations. To fully utilize hardware capabilities, we design a dispatcher which piggybacks DMA tasks onto AVX tasks and executes them in parallel to overlap DMA copy with AVX copy (§4.3). Leveraging its global view, Copier employs layered copy absorption to eliminate redundant intermediate copies in a fine-grained manner. Apps can further exploit copy absorption with lazy copy semantics (§4.4).

Management. Copier maintains fairness and isolation among *clients* (user processes or OS services with standalone contexts [54]) with Copier scheduler and cgroup extension (§4.5).

Usages. Copier introduces two programming primitives: amemcpy() and csync(). Apps perform async copy with amemcpy(), and sync the data before use with csync(). Copier provides a toolchain to ease the use, including libraries, a debug tool, and an experimental compiler. We present Copier's use cases for whole system copy optimization (§5) on two OSes, where Copier is used to optimize copies in OS services, frameworks, and apps.

4 Copier Design

4.1 Pipelined Copy-Use with Copier Abstraction

As shown in Fig. 4 (left), Copier's library provides two major high-level interfaces: amemcpy (async_memcpy) and csync (copy_sync). Clients can submit an async copy task with amemcpy and ensure the data is ready before use with csync. The detailed APIs and usage are described in §5.1.

From the OS API perspective, clients interact with Copier through three types of (per-client) queues (CSH Queues): **Copy Queue**, **Sync Queue**, and **Handler Queue**, which are mapped to the client's address space. These queues are the underlying abstractions supporting high-level APIs, as shown in Fig. 4 (right). Copier has its own context (e.g., kthread in Linux) and polls the queues to handle the requests

```
▶Existing copy and use
func copyUse(src, dst, n):
  memcpy(dst, src, n);
  free(src);
  /* some work ... */
  val = dst[0];
  return val;

▶Copier programming model
func copyUse(src, dst, n):
  amemcpy(dst, src, n,
    handler={free, src});
  /* some work ... */
  csync(addr=dst, len=8);
  val = dst[0];
  return val;
```

```
▶Internal imp with CSH Queues
func amemcpy(dst, src, n, handler):
  /* alloc desc, map copy to it */
  desc = alloc_desc(dst, src, n);
  enqueue(CopyQueue, {dst, src, n,
          desc, handler, ..});

func csync(addr, n):
  desc = lookup(addr);
  if (!desc_ready(addr, n)):
    enqueue(SyncQueue, {addr, n});
    while(!desc_ready(addr, n)){};

/* Periodically invoked */
func post_handlers():
  run(dequeue(HandlerQueue));
```

Figure 4. Programming model and interfaces.

(Fig. 5). Queue-based abstractions offer non-blocking, async operations, circumventing the overhead of syscalls [26, 55].

Asynchronous copy with Copy Queue (Q_{Copy}). Clients enqueue **Copy Tasks** for copy via Q_{Copy}, as shown in Fig. 5. A Copy Task includes the *source* and *destination*, identified by virtual addresses or pages (used by kernel), and the copy's *length*. [3] Q_{Copy} handles multiple tasks in FIFO order, maintaining copy order for consistency. While it covers basic functionalities, it leads to a long waiting time for the completion of a big copy, affecting the effectiveness of asynchrony. Copier introduces *segment-based copy* to mitigate the issue.

Fine-grained copy-use pipeline with segments. We observe that *apps usually access copied data piece by piece* in a regular pattern (e.g., sequential). Therefore, it is not necessary to complete the *entire* copy before use; ensuring the availability of currently *required* data suffices. Based on this observation, Copier abstractions enable the fine-grained update of a task's progress. Copier partitions a copy into several **segments**, i.e., fixed-size regions, whose size is determined by the *granularity* in Copy Task. Clients are required to specify the address of the task's **descriptor** — a bitmap tracking the copy status of each segment — which is checked by clients to confirm the progress of the copy. Copier updates descriptor's relevant bit after finishing the copy to a segment. This design facilitates copy-use pipelines, enabling parallel data copy and use.

Task promotion with Sync Queue (Q_{Sync}). To prevent the head-of-line blocking caused by FIFO-queue-based abstractions [25–27] — with basic Copy Queue, preceding tasks block subsequent ones even when the preceding data is not used immediately, Copier introduces Q_{Sync}, which enables apps to adjust the execution order of async tasks as needed, i.e., out-of-order execution. When a client finds required segments unready, it submits a **Sync Task** with the required address and length, as shown in Fig. 5 and csync in Fig. 4.

[3] Copier's lib (§5.1) supports overlapping copy (memmove) by splitting it into two tasks and first submitting the task whose source will be overwritten.

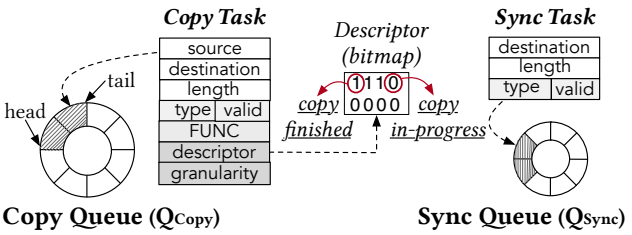


Figure 5. Copy Queue and Sync Queue. *They are mapped to clients. Copier monitors them with optimized polling (§4.5).*

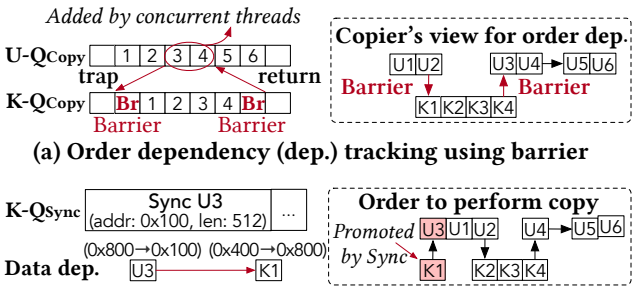


Figure 6. Dependency tracking. *U = userspace, K = kernel.*

This raises the priority of corresponding segments and all tasks it depends on, following data dependency in §4.2.

Delegation-based handler. A challenge for prior zero-copy methods is *post-copy handling*, which involves immediate actions on source buffers after copy, e.g., free in copyUse (Fig. 4). Linux zero-copy socket [24] relies on explicit syscalls to check buffer's ownership, while zIO [7] employs a garbage collector to defer deallocation, introducing complexities and limiting their usage. We propose delegation-based handling. Specifically, we add the func field in Copy Task, encapsulating a function pointer and its arguments, designated to be invoked upon copy completion. We use *KFUNC* for kernel functions and *UFUNC* for user ones. Copier helps to execute the KFUNCs. For UFUNCs, Copier submits a task including the function to Handler Queue. Copier library (§5) checks the queue and executes the handlers (post_handlers in Fig. 4).

4.2 Async but not Chaotic with Dependency Tracking

Although the async abstractions move copy out of the critical path, they introduce challenges in maintaining correctness. Executing the async tasks in the correct order is challenging for two reasons: (1) Separated async queues across privilege levels lack efficient synchronization mechanisms. (2) Out-of-order execution occurs because Sync Tasks prioritize some copies. Copier tracks *order dependency* and *data dependency* to address these challenges respectively.

4.2.1 Order Dependency. For kernel's security, we maintain two sets of CSH Queues for each process by default: user-mode (u-mode) queues for the app, and kernel-mode (k-mode) queues for kernel services sharing the process's context (e.g., syscalls). It is necessary to monitor the *submission order* of tasks in u-mode and k-mode queues. E.g., the kernel submits a task (A→B) to the k-mode Q_{Copy} during recv, followed by the app submitting a task (B→C) to the u-mode Q_{Copy} after recv returns. Copier must ensure A→B occurs before B→C for correctness. This dependency is defined as *order dependency*. The order dependency of tasks within u-mode/k-mode queue is naturally determined by their order in the queue, without explicit specification. However, as the queues are non-blocking and user's timestamps are untrustworthy, determining task order across u-mode and k-mode queues poses a challenge.

Cross-Queue Barrier. We observe that we can *utilize trap (e.g., syscall) and return events as indicators* to track dependencies, as shown in Fig. 6-a. Specifically, we introduce *Barrier Tasks*. Each time before kernel submitting the first Copy Task after a trap, it submits a Barrier Task, recording current position of user Copy Queue (and the times it is reused). Similarly, kernel submits a Barrier Task before returning to userspace. With the indicators, Copier knows the dependency between the two queues: $K1$–$K4$ before $U5$ and after $U2$.

A possible (corner) case in multi-threaded apps is that, when kernel submits tasks, other threads may submit tasks to user Copy Queue ($U3$–$U4$ in Fig. 6-a). This concurrency issue also exists in traditional sync copy, e.g., during kernel performing copy_to_user, another concurrent thread performs copy. Same as its original behavior, the order between $K1$–$K4$ and $U3$–$U4$ is undetermined. Copier prioritizes tasks in k-mode queues for simplicity, as shown in Fig. 6-a (right).

4.2.2 Data Dependency. To process Sync Tasks and enable copy absorption (§4.4), Copier has to track *data dependency* — whether a Copy Task's *involved data* is dependent on another due to overlapping memory regions. Data dependency can be easily established by traversing Copy Tasks in reverse using the tracked order (§4.2.1) and comparing the regions (both sources and destinations).

In Fig. 6-b, when processing the Sync Task which prioritizes $U3$, Copier assesses the data dependency of $U3$ and finds it depends on $K1$. Then, Copier adjusts the copy order, ensuring the essential tasks are done first. Notably, although it differs from order dependency, re-ordering is safe as the tasks are independent. Copier handles Sync Tasks in k-mode Sync Queue first, and then the tasks in u-mode queue.

4.3 Harmonizing Copy Units with Task Piggybacking

Copier's OS service role brings opportunities for copies to benefit from full hardware capabilities. By saving AVX register states only upon Copier's activation and restoring them just before it sleeps, we avoid the overhead of frequent state saves and restores with each copy. We also employ parallel copy with AVX and DMA [47] to enhance copy throughput.

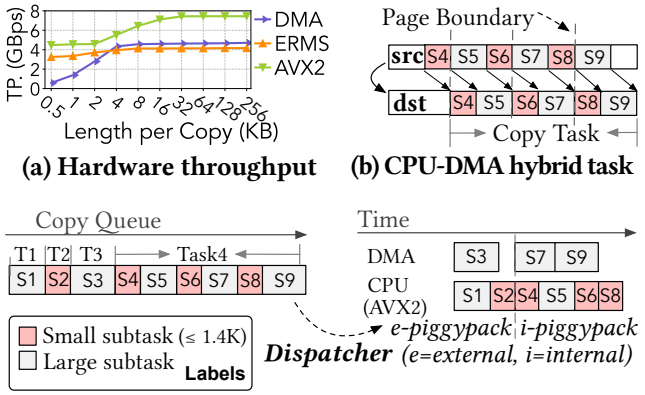


Figure 7. Designs to utilize hardware. *S1 = Subtask1. In (b), we assume all physical pages are non-contiguous.*

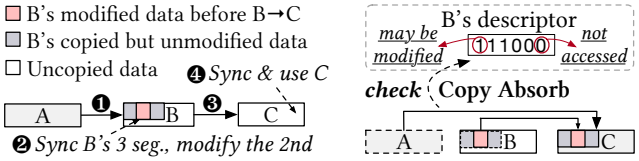


Figure 8. Design of Copy absorption. *seg. = segment.*

Challenges. Although DMA copies without consuming CPU cycles and excels at large copies (≥4KB), it has lower throughput than AVX2, especially for small copies, as shown in Fig. 7-a. Submitting DMA tasks and waiting for them to complete also wastes cycles that could be used for copy. It remains unclear how to integrate diverse hardware capabilities given their varying characteristics.

CPU-DMA hybrid subtasks. DMA requires that both source and destination of the copy have contiguous physical addresses. We define a **subtask** as the basic unit within a Copy Task to execute copy in specific hardware, e.g., DMA or AVX, which is the largest unit meeting the continuity requirement. E.g., in Fig. 7-b, non-contiguous pages divide the subtasks.

We observe that DMA is inefficient for small subtasks due to the overhead of submitting DMA tasks, whose costs in our server are sufficient to copy 1.4KB using AVX2. Based on the observation, we propose *hybrid subtasks* — for a single task, Copier considers its subtasks with sufficient sizes as *DMA candidates*, e.g., S5/S7/S9, and uses CPU for others.

Piggyback-based hardware dispatcher. Copier introduces the *dispatcher* to allocate subtasks between heterogeneous copy units. As DMA is notably slower than AVX, improper allocation may result in non-trivial waiting time within CPU to enforce dependencies. We propose the *piggyback mechanism* to pair and synchronize subtasks executed by CPU with those performed by DMA. The dispatcher works in rounds, it first schedules subtasks, and then executes them:

• **Packed scheduling.** Copier piggybacks DMA copy onto AVX copy to overlap the DMA copy with AVX copy, avoiding the CPU consumption to wait for DMA completion. Specifically, (1) for a large task (≥12KB), Copier picks subtasks from the DMA candidates *within the task* and assigns them to DMA. It assigns other subtasks to AVX, as shown by *i-piggyback* in Fig. 7-c. Copier tries to ensure the completion times of the subtasks on CPU and DMA are close based on hardware metrics (Fig. 7-a). (2) If the task's length is below the threshold, Copier uses the *e-piggyback* method. Different from i-piggyback, it picks subtasks from *several adjacent tasks* in the queue, provided they have no data dependency. Copier picks DMA subtasks from the task's latter part (i-piggyback), or the latter tasks (e-piggyback), as they typically have longer Copy-Use windows, e.g., although S1 and S5 in Fig. 7-c are DMA candidates, they are assigned to AVX.

• **Parallel execution.** Copier first submits DMA subtasks (in batch) to the device queue, then executes AVX subtasks, and finally confirms the completion of DMA subtasks.

The piggyback mechanism ensures high utilization of both copy units without wasting cycles waiting for DMA completion. It can be applied to more heterogeneous units [56, 57].

Address Transfer Cache *(ATCache).* DMA requires translating virtual addresses (VAs) to physical addresses (~240 cycles/page). We note that apps' copy addresses often demonstrate high locality, as they tend to use recycled buffer pools and fixed I/O buffers to minimize (de)allocation overhead, e.g., the address recurrence in Redis surpasses 75%. We propose *ATCache*, which caches VA, pages, and length, to reduce translation overhead. The memory subsystem will notify ATCache to invalidate entries when the mappings change.

4.4 Merging Redundant Copy with Copy Absorption

Continuous copies, especially those across privilege levels, are common in I/O-intensive apps, e.g., when Redis processes a SET, the value is first copied from kernel buffer (*K*) to input I/O buffer (*I*) through recv(), and then copied to the database (*D*) after parsing the request. In many cases, only a small portion of the intermediate buffer (e.g., *I*) is accessed [7]. Copy absorption, by design, eliminates the unnecessary copy (K→I) of untouched data and merges the copies into a "short-circuit" copy (K→D). zIO [7] utilizes a similar observation but is limited to user-mode. With order dependency tracking (§4.2.1), the OS service role endows Copier the global view of copies, thus enabling cross-privilege copy absorption.

Challenges. Fine-grained granularity brings challenges. Consider two copies in the queue, A→B and B→C, when *C* is used, a naive design is to copy A→C directly. It is correct for coarse-grained copy where *B* must be fully copied once it is used, but not for Copier's fine-grained copy. A case (Fig. 8-a) is that after ❶ submitting A→B, the client ❷ modifies part of the data in *B*, and then ❸ submits B→C. When ❹ *C* is synced, we cannot simply copy A→C as part of *B* is modified.

Layered copy absorption. Recall the segment-based Copier abstractions (§4.1), since clients sync the data before use, a segment could not have been accessed if its bit in descriptor is not marked. Conversely, it might be modified if its bit is marked. We propose layered absorption — instead of copying from one source, Copier copies from sources with the latest data. In Fig. 8-b, when *C* is synced, Copier copies the first 3 segments from *B* and others from *A*, as *B*'s 3 segments are marked in the descriptor and might contain newer data.

Optimizations with app semantics. We present *Lazy Copy Task* to further exploit copy absorption. Clients can mark a Copy Task as lazy in the type field. It has the lowest priority and will be handled only when: a Copy/Sync Task depends on it, or a specified period has elapsed. A Lazy Task usually is not executed but acts as a mediator in copy absorption, which is beneficial in cases where the data is not accessed but just moved. E.g., a proxy reads a message from kernel (K1→U), and sends it out (U→K2) after redirecting. As the proxy only uses a few data of the message (e.g., header), it can mark the first task as lazy. With copy absorption, Copier performs copy about K1→K2 during `send()`. A special Sync Task, `abort`, is introduced to explicitly discard unnecessary Copy Tasks, e.g., the proxy can submit an abort task to discard K1→U (which is still queued) after finishing sending the message. Notably, Copier does not implicitly discard any tasks.

4.5 Fair and Isolated Multi-client Serving

As an OS service, Copier has its own (k)threads (called Copier threads) to handle requests from clients. It needs to manage, isolate, and schedule copy resources, and handle faults.

4.5.1 Copier Threads. Copier threads leverage and enhance existing async mechanisms within OS kernels to poll requests from the queues. We implement it based on io_uring (with *SQPOLL*) in Linux, where the io_uring kthreads poll the clients' queues. We introduce the following two extensions.

Scenario-driven polling. Copier supports two polling modes: (1) NAPI mode [58] (default): Copier reuses io_uring's NAPI support [59] to balance performance and polling overhead. (2) Scenario-driven mode: Copier thread is activated only when a target scenario is detected and sleeps when the scenario changes. This is particularly beneficial for devices with limited cores and a critical focus on power consumption, e.g., our practice on commercial smartphones (§5.3).

Copier thread auto-scaling. We design an auto-scaling mechanism to support dynamic loads, which keeps the average load between predefined low_load and high_load. Copier launches more threads when sustained high loads are detected, and puts threads to sleep when the load is low. Clients are preferentially assigned to NUMA local Copier threads.

4.5.2 Resource Isolation with Extended Cgroup. With Copier, *copy is managed as a basic resource like CPU time and memory*. Based on this perspective, we extend Linux control group (cgroup) [60] with a controller named copier. Users can configure the share of Copier resources of each cgroup (copier.shares). We do not use CPU slices used by Copier as copy resources, acknowledging copies' completion times can vary due to cache and TLB states. Instead, Copier uses *copy length*, the length copied, as the resources. The extension is compatible with existing OSes, and can be easily used by users and frameworks [61] to achieve resource isolation.

4.5.3 Scheduling. Each Copier thread schedules clients following CFS [62] — the key consideration is fairness. Specifically, Copier threads maintain a *total copy length* for each process and select the process with the minimum total copy length to serve each time it schedules, akin to CFS's strategy. Administrators can adjust Copier's *copy slice*, defining the maximum copy length upon each scheduling. The scheduler works in each cgroup to maintain fairness and cooperates with the copier controller to schedule among cgroups.

4.5.4 Multi-Address Spaces and Fault Handling. Serving multiple processes faces challenges with the memory subsystem. The processes submit tasks with virtual addresses (VAs) specific to their own address spaces, which cannot be directly used by Copier. More complex still, these VAs might not be backed by physical memory due to on-demand paging, CoW, or even malicious behavior, triggering page faults that are hard to handle in Copier's context. A naive design is to switch Copier's page table to use the VAs directly; however, it is costly and cannot effectively handle page faults.

Proactive fault handling. Copier adopts a proactive approach to handle the challenges. Instead of waiting for potential faults, Copier proactively triggers and handles them. Specifically, Copier uses VMAs [63] and page tables to confirm the VAs' physical mapping and locks the mapping until the copy is completed [64]. The address translation can be optimized with ATCache (§4.3). If some pages are not mapped, Copier constructs exception parameters and invokes fault handlers. Copier also applies security checks, e.g., address boundary checks. If the faults cannot be resolved or the copy has security issues (e.g., illegal kernel addresses), Copier drops the task and signals the process as before (sigsegv).

4.6 Target Scenarios and Scope

Like other OS services [25, 54], Copier is not a silver bullet for all cases. We summarize Copier's target scenarios and scope as follows. For the unsuitable cases, developers can fall back to prior sync copy.

Copy size. With sufficient Copy-Use windows, Copier outperforms a sync copy when the copy time exceeds the time of task submission and csync, e.g., kernel copies of ≥ 0.3KB and userspace copies of ≥ 0.5KB on our server. Due to better utilization of hardware, Copier also benefits large copies even without sufficient Copy-Use windows (e.g., CoW), e.g., kernel copies of ≥ 2KB and userspace copies of ≥ 12KB.

Table 2. Major APIs introduced by Copier.

Levels	APIs	Descriptions
libCopier (high-level)	amemcpy(*dst*, *src*, *size*); amemmove (..);	Async memcpy and memmove. They use the per-process default queues.
	csync(*addr*, *size*);	Ensures prior async copy is finished. It submits Sync Task and polling waits if some segments are not ready.
	csync_all(*void*);	Ensures all async copies and FUNCs finish.
	shm_descr_bind(*shm*, *descr*, *segment_len*);	Bind descriptor (on shared memory) to shared memory that apps use.
libCopier (low-level)	_amemcpy(.., *fd*, *FUNC*, *desc*, *opts*); _amemmove(..);	Allows customized management of descriptors, FUNCs, and lazy copy. *desc = descriptor. fd = file descriptor.*
	_csync(*offset*, *size*, *fd*, *descriptor*, *opts*);	Allows customized descriptor management. Set fd to -1 to use the default queues.
	copier_create_ mapped_queue(*len*);	Creates a client with user and kernel queues, and mmaps the user queues.
OS (Syscall)	copier_create_ queue(*len*);	Creates Copier queues, and returns a fd on success.
	set_copier_opt(*opts*);	Sets global parameters.
	copier_awaken(*fd*);	Wakes the Copier thread during sleep.

Data access pattern. Copier is suitable for apps with regular access patterns (e.g., sequential) and sufficient Copy-Use windows, which is common in I/O-intensive and memory-intensive apps (e.g., KV store, network protocol processing, file I/O, IPC, etc.) [7]. It is unsuitable for apps that access data randomly.

Sensitivity. Like other polling-based solutions [44, 65–67], Copier may consume more CPU cycles. Copier is efficient when there are underutilized cores (common in datacenters [68, 69]). When CPU resources are fully utilized, it benefits latency-critical apps but may hurt throughput-critical apps. For apps with copy chains or large copies, copy absorption and hardware capabilities can save more cycles than those consumed by polling, benefiting throughput-critical apps even with fully utilized cores. We evaluate this in §6.3.4.

5 Practice of Copier

As with prior OS services [25, 54], Copier requires effort to leverage its capabilities to bring benefits. We present the toolchain that is designed to ease the use of Copier and the guidelines (§5.1), and two OSes supporting Copier (§5.2 and §5.3), with cases where we utilize Copier to optimize OS services, frameworks, and apps.

5.1 Copier Toolchain

The toolchain includes three parts: libCopier to provide APIs for development, CopierSanitizer to provide tools for debugging, and CopierGen to automate the porting.

5.1.1 Development with libCopier. We design libCopier with both high-level and low-level APIs, as documented in Table.2. Most developers can use high-level APIs, e.g., ***amemcpy*** (async-memcpy), with similar interfaces as sync copy. We introduce a new primitive, ***csync*** (copy-sync), which simply requires an address and the size, used to ensure immediate consistency and order. From an app's view, it performs copy as usual and syncs the data before use using csync(). We also add csync_all() to sync all uncompleted async copies.

Internal implementation. To keep the high-level APIs simple, libCopier prepares and maintains default queues for each process. To accelerate the creation of descriptors during task submission, libCopier maintains a descriptor pool and pre-allocates descriptors with different sizes. When libCopier requires a descriptor, it fetches the proper one from the pool.

Optimizations with low-level APIs. libCopier provides low-level APIs (e.g., _amemcpy) for expert developers for better performance. These APIs are used to optimize frameworks (e.g., Binder or gRPC) which can benefit many high-level apps or a few significant apps. Specifically, low-level APIs enable two optimizations: (1) *customized descriptor management*: developers can re-use the descriptor of the same buffer (e.g., I/O buffers) to avoid costs of descriptor allocation and recycling, and use pre-determined descriptors for _csync to avoid table lookup costs. (2) *multi-queue supports*: developers can create per-thread queues for a process, and use the specific queues with the fd. As Copier only tracks the dependencies among paired kmode and umode queues, per-thread queues are used by apps whose copies in different threads do not have dependency, e.g., common web servers [70, 71].

Multithreading and concurrency. Copier utilizes a lock-free ring buffer as the underlying implementation of CSH Queues (§4.1). To submit a task, libCopier *acquires* a task buffer by moving the queue *head* using fetch-and-add. After filling the fields, it sets the task's *valid bit*. Upon discovering a valid task at the queue *tail*, Copier processes it and advances the tail (the order of tasks follows the order of *acquiring*).

Csync guidelines. We summarize guidelines for csync insertion based on practices. (1) *Direct data access*: sync before reading/writing destinations (dst) and writing sources (src). (2) *Buffer free*: sync before dst/src buffers are freed, or use post-copy handler (§4.1). (3) *Used by external lib/func*: e.g., sync before passing the buffer to strchr. (4) *Visible to (unmodified) external threads*: e.g., sync before updating page table during CoW fault handling.

Correctness and semantic equivalence with formal verification. We have verified that: *once csync is correctly added according to the guidelines, the semantics of amemcpy refine those of memcpy in both single-threaded and multi-threaded contexts.* [4] This means, Copier will not introduce any new bugs (compared with memcpy) once csync is correctly used.

Shared memory. libCopier supports asynchronous copy to shared memory. Processes exchanging data through shared memory (*shm*) need to establish a dedicated shared buffer

[4]The **formal proofs** are attached in the appendix for reference.

(*Dshm*) for descriptors, and bind (*shm_descr_bind*) it with the shared memory to use high-level csync. libCopier uses data's offset to the start of *shm* to locate the data's descriptor on *Dshm*. csync for shared memory will wait until the descriptor is marked ready. Android Binder IPC, which we discuss and evaluate in the paper, is a use case of Copier on shared memory.

Although csync is lightweight, too frequent use of it (e.g., per-byte csync or sync the same data repeatedly) may lead to performance degradation. However, in practice, its overhead is limited. First, memory accesses in typical apps exhibit locality, e.g., sequential/strided access, so apps can sync once every one to few KB of data used. Second, copied data in copy-bottlenecked apps is often one-time use, e.g., in OpenSSL [39] the data is never reused after being decrypted. Apps can sync the whole copy before the first use to avoid excessive csyncs.

Based on our practice on 10 apps and kernel services, developers can port typical apps with acceptable efforts (§6.3.1).

5.1.2 Bug Detection with CopierSanitizer.

Like other services and APIs, developers may misuse amemcpy or csync, causing potential bugs. We develop CopierSanitizer, a bug detection tool to help users find omitted or improper csyncs.

Key idea. We utilize *shadow memory* to detect Copier-related bugs, which is widely used in prior industrial bug detection tools like AddressSanitizer [72], etc. [73, 74]. Specifically, CopierSanitizer uses shadow memory to record the metadata of program memory, enabling the detection of errors by tracking memory accesses and identifying illegal accesses. It identifies memory accesses during compilation and inserts lightweight instrumentation code to record the states and detect the errors.

Internal implementation. We implement CopierSanitizer based on AddressSanitizer [72]. Specifically, when a program calls amemcpy (and amemmove, etc.), CopierSanitizer marks the source and destination memory as "inaccessible" ("poisoned" [75]). After the program calls csync, it marks the region involved as "accessible". In this way, memory accesses and frees without csync will be captured and reported to developers. CopierSanitizer supports multi-threaded apps.

CopierSanitizer now focuses on user-mode apps, and can be extended to kernel by incorporating KASan [76].

5.1.3 Porting with CopierGen.

To ease the burden of porting existing apps, we explore the method of utilizing compiler tools to automate the porting, following prior works like Mira [77]. The key insight is that IRs preserve high-level abstractions like variables, but with data access constrained to a few operations, e.g., load and store [78], providing insertion points for csync. We design CopierGen with a series of Passes [79], based on LLVM and MLIR [78]. Basically, CopierGen identifies variables corresponding to sources and destinations of copies, and inserts csync before access to them. We implement and validate it for basic cases like arrays, but leave handling complex issues (e.g., pointer passing) as future work. Mira faces a similar problem (checking whether the data is remote or local before access) to Copier and supports complex apps, making us believe in CopierGen's practicality.

5.2 The Cases of Copier in Linux

We implement Copier-Linux[5] based on Linux-5.15.131. Copier is used to boost the performance of OS components, frameworks, and apps.

Network stack. We use Copier to optimize copies during recv() and send(), where copy is used to transfer data between userspace and kernel. For recv(), the Copy-Use window exists between kernel copying data to userspace and apps using the data. We modify the network stack to submit Copy Tasks with a descriptor provided by libCopier for apps to check. The Copy Tasks include a KFUNC to reclaim the socket buffers for reuse. For send(), the Copy-Use window exists between kernel copying data to socket buffers and driver submitting the packets to hardware queues. With checksum offloaded to NICs [80, 81], the TCP/IP layers no longer require data access, but just use packet metadata. The network stack submits Copy Tasks at the socket layer and syncs the data before the driver enqueues packets into the NIC TX queues.

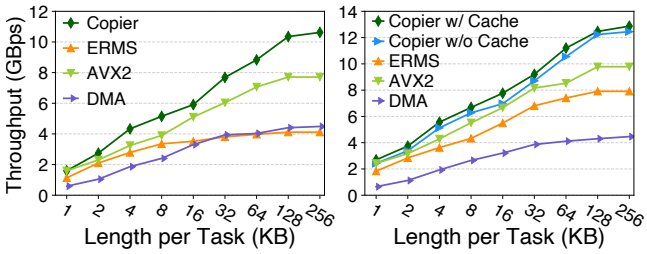
Copy-On-Write fault handling. CoW enhances forking efficiency but incurs runtime overhead from page fault handling [82, 83], where copy is a notable bottleneck. In Copier-Linux, kernel submits Copy Tasks after allocating a new page, and syncs before updating the page table. Instead of offloading the whole copy, Copier-Linux divides the work between CoW handler and Copier to reduce the waiting time.

Android Binder IPC framework. Binder IPC [37] entails a two-step data transfer: client's data is copied to a kernel buffer by Binder driver, then IPC server maps the kernel buffer into its address space. Apps utilize Parcel [84] framework to manage Binder IPC, which enables typed data writing into or reading from Binder messages. Copier optimizes the copy with low-level APIs. Copier-Linux places the descriptor at the front of messages (shared memory) and Parcel utilizes _csync before use. The copy is hidden by Binder driver's processing (e.g., scheduling the server thread) and server's processing. As all the changes are made inside Binder and Parcel, ***apps can benefit without any modifications***.

5.3 Practice of Copier on HarmonyOS

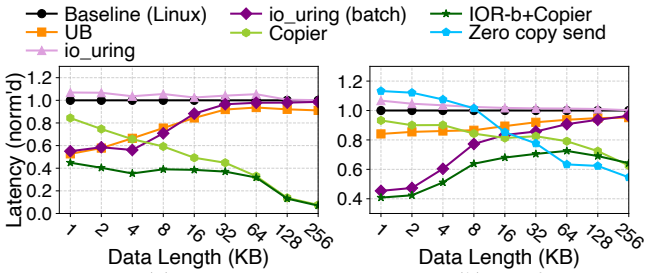
We integrate Copier with HarmonyOS 5.0 (HongMeng kernel [4]) on smartphones experimentally. A significant challenge is that the energy sensitivity of smartphones makes uncontrolled polling unsuitable. In response to this, we use the **scenario-driven** design (§4.5) and avoid long-term polling.

[5]The source code is available at https://github.com/SJTU-IPADS/Copier.



(a) No buffer repetition **(b) 75% buffer repetition**

Figure 9. Copy throughput. *The benchmark submits sufficient copy tasks and records the completion times. For the baselines, we replace Copier's copy method with glibc AVX-memcpy and kernel-mode methods (ERMS, DMA).*



(a) Recv **(b) Send**

Figure 10. Average latency of *send()* and *recv()* syscalls. *The load is generated by echo servers and clients. Zero-copy* recv() *is not evaluated as it requires special NIC architecture [88, 89]. IOR-b = io_uring (batch).*

Copier is utilized to enhance the performance of video recording and playback, which involve numerous copies and are critical scenarios for current smartphones.

6 Evaluation

In this section, we aim to clarify the following questions.

- How is the performance of Copier in handling copy requests? (§6.1.1)
- How does Copier benefit other OS services? (§6.1.2)
- How does Copier benefit real-world applications? (§6.2)
- Is Copier effective and energy efficient in smartphone settings? (§6.2.4)
- What are the development efforts to adopt applications to Copier? (§6.3.1)
- Can Copier make benefits when the CPU cores are fully utilized? (§6.3.4)

Experimental setup. The experiments on servers are conducted on servers with 2 Xeon E5-2650 v4 CPUs [85] with a constant 2.9GHz frequency and 128GB DDR4 memory, unless otherwise specified. The baselines are run on Linux-5.15.131, which is the system Copier is based on. Linux 5.15.131 still represents SOTA kernel copy implementations because, despite some engineering improvements [86], the way Linux performs memory copy (i.e., using the ERMS instructions) has not changed in newer Linux versions. The experiments on smartphones are conducted on Huawei Mate 60 Pro, with Kirin 9000S CPU and HarmonyOS 5.0. In all experiments, Copier uses one dedicated core to copy.

Baselines We compare Copier with the following state-of-the-art works: Linux zero-copy socket [24], Userspace Bypass (UB) [87], and zIO [7]. We set zIO's *threshold* to 4KB, which caps the minimum copy to apply it.

6.1 Micro Benchmarks

6.1.1 Copy Performance. We evaluate the throughput of Copier to handle Copy Tasks of different sizes. Results (Fig.9) show that the throughput of Copier, which uses AVX2 and DMA to copy in parallel, increases by up to 158% (55% for 4KB) compared to the default copy method in the kernel (ERMS [29]), and by up to 38% (33% for 4KB) compared to the userspace copy method (AVX2) when there is no buffer repetition. When the rate of buffer repetition is 75%, Copier's throughput increases by up to 63% (53% for 4KB) compared to ERMS, and by up to 32% (30% for 4KB) compared to AVX2; the ATCache brings a 2%–11% higher throughput.

6.1.2 Performance of OS Services with Copier

Syscalls (send and recv). We compare Copier with the following syscall optimization methods and systems:

- **Kernel offloading** (Case: UB [87]) reduces the number of context switches by offloading userspace code into kernel.
- **Asynchronous syscalls** [26] (Case: io_uring [25]) parallelize execution of syscalls and app to hide the overhead.
- **Batch submitting and processing** (Case: io_uring with batching [25]) can reduce the number of context switches and improve locality. The batch size is configured as 100.
- **Zero-copy** (Case: zero-copy socket [24]) reduces the copy costs by sharing memory between kernel and apps.

Results (Fig.10) show that Copier reduces the latency of send by 7%–37% compared to baseline (normal syscalls) and by 27%–59% with io_uring batching. Copier reduces the latency of recv by 16%–92% and by 55%–93% with batching. UB's effect diminishes as data size increases since copy dominates the costs. Zero-copy send performs better than Copier with large data (≥ 32KB), but has little effect for smaller data, and introduces a non-trivial burden to manage the shared buffers, as discussed in §2.2. io_uring does not reduce the latency of syscall execution, and the optimization effect depends on the apps. Batched io_uring improves performance of both normal syscalls and syscalls with Copier, but its effect diminishes as the data size increases for normal syscalls.

Android Binder IPC. We use a benchmark [14] to measure the end-to-end latency for the IPC client to send a message containing *n* strings of 1KB, the server to read these strings one by one, and then send a reply to the client. This pattern is widely present in data storage, image rendering, and audio playback apps. Experimental results show that Copier

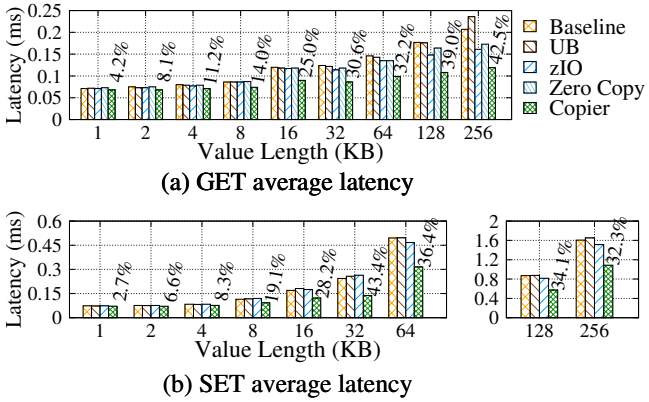


Figure 11. Results of Redis GET and SET. *Workload is generated by official redis-benchmark [90] with 8 parallel (closed-loop) clients. For SETs, improvement diminishes for values of ≥64KB due to the increased overhead of request sending. Redis version: v6.2.14.*

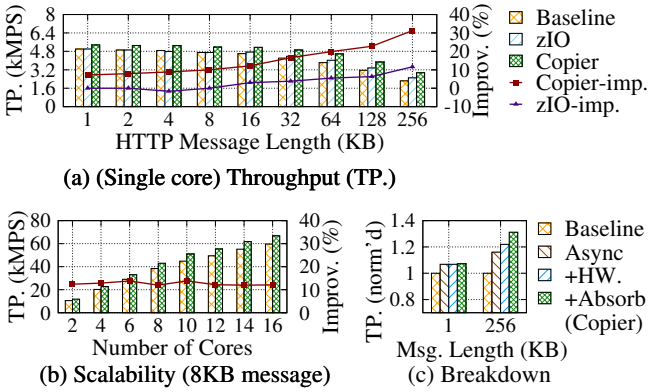


Figure 12. TinyProxy's throughput. *MPS: messages per second. imp. = improvement. Load is generated by echo servers and clients proxied by TinyProxy. Fig.-b is conducted on servers with Xeon Gold 6238R CPUs [91]. TinyProxy version: v1.11.1.*

reduces the average end-to-end latency by 9.6%–35.5% when n varies between 10–800.

CoW Handling. We use a benchmark that continuously triggers CoW page faults and measures the average latency of fault handling. Results show that Copier reduces the average thread blocking time per page fault by 71.8% for 2MB pages and 8.0% for 4KB pages.

6.2 Application Benchmarks

6.2.1 Redis. Copier optimizes the 5 copies in Redis: (1) copy of requests from kernel to I/O buffer in recv(); (2) when processing SETs, copy of value from the I/O buffer to the value's buffer; (3) when processing GETs, copy of value from the value's buffer to the I/O buffer; (4) copy of replies from I/O buffer to kernel in send(); (5) one of internal copies during processing. Due to *proactive fault handling*, Copier also moves page faults during copy out of the critical path.

Results (Fig. 11) show that compared to baseline, Copier reduces end-to-end average latency by 2.7%–43.4% for SETs and 4.2%–42.5% for GETs. Copier reduces tail latency (P99) by 5.9%–33.4% for SETs, and 5.59%–47.8% for GETs. It improves throughput by 2.4%–50.0% for SETs and 4.2%–32.0% for GETs.

We evaluate Linux zero-copy send(), which is only efficient when the value length is ≥32KB due to its alignment constraints, TLB flush costs, and additional syscalls for ownership management. We compare Copier with zIO [7] and UB [87]. UB can only optimize SETs and GETs of ≤4KB because it slows down the program's memory access. zIO reduces latency by up to 20.0% for GETs because it efficiently eliminates one userspace copy. However, because Redis always reuses the input buffer and causes page faults, zIO is only effective for SETs of ≥64KB (up to 6.1% improvement). Copier outperforms these SOTA solutions because of its feasibility and effectiveness in optimizing diverse copy cases.

6.2.2 TinyProxy. Network proxies are widely used in scenarios such as load balancing, reverse proxying, etc. [92]. We observe that many proxy functions (e.g., redirecting, load balancing, rate limiting, etc.) do not need to use most of the data in a message, but typically only use the request line and message headers. This presents opportunities to apply copy absorption (and lazy copy). TinyProxy [9] is a widely used lightweight proxy. It first reads the message and uses the request line and header to decide on the upstream server. Then it uses memcpy() to organize the message and sends the message out. Copier can optimize the three copies involved into a single (async) copy. We evaluate the throughput when TinyProxy performs HTTP message forwarding (Fig. 12-a). Compared with the baseline, Copier increases the throughput by 7.2%–32.3%. zIO achieves up to an 11.6% improvement compared with the baseline; it can only eliminate one userspace copy because of its inability to handle inter-boundary copies. Furthermore, due to the use of page remapping, zIO cannot optimize messages that do not occupy entire pages and is effective only for messages of ≥16KB.

6.2.3 Frameworks and Libraries

Protobuf [36] is the serialization library used by gRPC [93]. The apps receive a serialized message from network and deserialize it into an object with Protobuf. With Copier, copying the message to userspace is executed in parallel with deserialization. Copier reduces the average latency of receiving and deserializing a message by 4%–33%, as shown in Fig. 13-a.

OpenSSL [39] is a famous TLS/SSL library. On receiving a message, it first copies it to userspace via recv() and then decrypts it. Copier enables the copy to be executed in parallel with decryption. Copier reduces the average latency of SSL_read() (AES-GCM) by 1.4%–8.4%, as shown in Fig. 13-b.

zlib [38, 94] is a widely used compression library. It uses a sliding window as the interval for pattern matching, whose

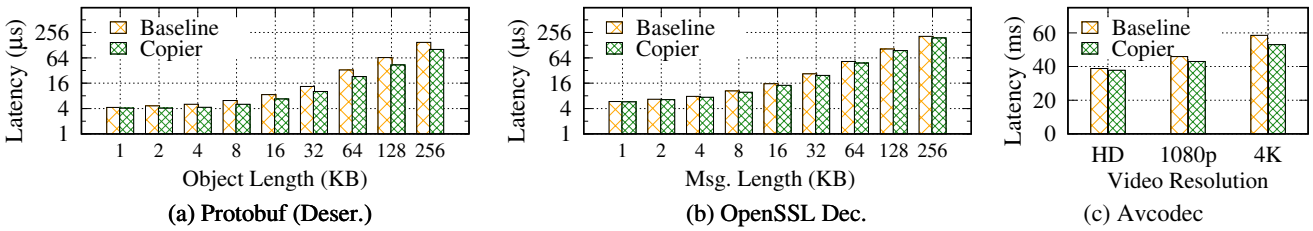


Figure 13. Results of libraries and HarmonyOS (smartphone). *Since the maximum size of a TLS record is 16KB, the speedup of OpenSSL with Copier remains stable for sizes ≥ 16KB. Copier reduces the average latency by up to 33%, 8.4%, 10% for the three scenarios, respectively. Protobuf version: v4.23.0 OpenSSL version: v3.4.0.*

movement relies on data copy. Copier enables copying data to the sliding window to be executed in parallel with pattern matching, and achieves up to an 18.8% speedup when compressing (deflate_fast()) data under 256KB.

6.2.4 Copier on Smartphone OS. Avcodec is HarmonyOS 5.0's unified audio/video coder/decoder framework. It copies data from inner buffers to the frame buffer after decoding. Copier enables the copy to be performed in parallel with subsequent logic before data usage (passing to rendering). We use production benchmarks which decodes video for 300s. Results (Fig. 13-c, on Huawei Mate 60 Pro with Kirin 9000S CPU) indicate that Copier reduces the decoding latency per frame by 3%–10% with a 0.07%–0.29% increase in energy consumption, which reduces frame drops during video playback by up to 22%. Copier's scenario-driven polling design saves energy (§ 4.5.1), where Copier is activated only for copy-intensive workloads, and sleeps when queues are empty.

6.3 Extended Studies

6.3.1 Efforts to Adapt Applications. The efforts for legacy apps and OS services are moderate, as much complexity is handled by libCopier, as shown in Table. 3. CopierSanitizer helps us locate positions to add csync and avoid omissions. It took a graduate student 1-2 weeks to port complex apps with many copies (e.g., Redis and TinyProxy), and 1-2 days to port relatively simple apps (e.g., zlib and Protobuf).

App/OS Service	LoC	App/OS Service	LoC	App/OS Service	LoC
recv()	58	Tinyproxy	27	Redis (SET&GET)	37
send()	56	Protobuf	14	CoW	42
zlib (deflate)	18	OpenSSL (AES-GCM dec.)	31	Android Binder IPC	55 driver, 48 Parcel
Avcodec	94				

Table 3. Development efforts to adapt 10 apps and OS services evaluated.

6.3.2 Multi-threading Scalability. We evaluate Copier's scalability using Tinyproxy, with the default per-process queues. Results (Fig. 12-b) show that Copier scales well with 16 threads and more than 130K tasks submitted per queue per second, thanks to Copier's lock-free queue design.

6.3.3 Performance Breakdown. We analyze the impacts of async copy (§ 4.1, § 4.2), hardware capability (§ 4.3), and copy absorption (§ 4.4). Results (Fig. 12-c) show that for small copy (1KB), as it can be fully overlapped, async copy plays a major role; for large copy (256KB) that is hard to be fully overlapped, hardware and absorption also matter significantly.

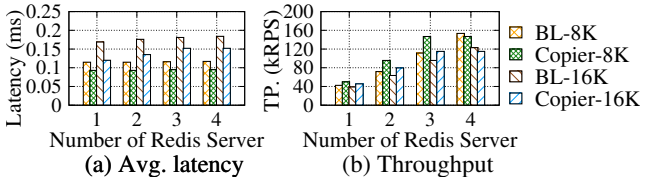


Figure 14. Redis SET performance with 4 cores. *BL = Baseline. When Redis number = 4, at most 3 instances are running simultaneously in Copier environment. We use 8 parallel clients per instance.*

6.3.4 Whole System Resource Utilization. We evaluate how Copier's design of using dedicated cores affects the system's overall performance. We limit the total resources to 4 cores and incrementally increase Redis instances until all cores are fully utilized. Results (Fig. 14) show that when there are idle cores, Copier can optimize both latency and throughput of the system. When resources are fully utilized, Copier reduces the request latency (by 18.8% for 8KB values and 17.4% for 16KB values). But as some cycles are used for task submitting and polling, the overall throughput decreases (by 4.3% for 8KB and 6.5% for 16KB). For apps with copy chains or large copies, since Copier can utilize hardware capabilities and copy absorption to reduce the CPU cycles used by copy, using Copier can still increase the throughput, e.g., for TinyProxy (§ 6.2.2, § 6.3.2) using equally 16 cores, Copier increases the throughput by 7.7%.

6.3.5 Microarchitectural Impact of Copier. Copier can mitigate pollution of the cache by decoupling large copy and app execution, thus reducing cache miss rate, which aligns with the observations of DPDK and glibc [95, 96]. When performing a large copy, app's hot data may be evicted from the top-level cache, resulting in a higher cache miss rate during execution. Copier reduces the eviction of hot data, thus reducing cache miss rate and optimizing CPI (cycles per instruction). Although Copier results in copied data not

being placed in top-level cache, processors have efficient cache prefetching, and apps typically prefer access patterns friendly to prefetching, e.g., sequential access. We record the numbers of cycles and instructions for Redis processing 50K SETs/GETs, and remove those of copy and polling. Copier reduces the CPI of copy-irrelevant code by 4%–16% for SETs and 6%–9% for GETs when value size varies between 4KB–64KB. In cache-sensitive scenarios [97–99], apps can switch to sync copy if Copier yields unfavorable results.

7 Discussion

The prospects of Copier as a hardware primitive. We believe Copier can be further integrated as a CPU hardware primitive to reduce CPU usage caused by polling and enhance the performance of async copy. Prior efforts [57, 100, 101] have integrated some memory copy optimizations to ISA primitives or memory controller functions. Although they do not provide async copy capabilities, they demonstrate the feasibility of developing more efficient hardware primitives for copy. Advanced AI accelerators, such as GPUs (e.g., Nvidia Tensor Memory Accelerator [102]) and NPUs (e.g., Ascend NPU Memory Transfer Engine [103]), provide hardware primitives for async copy, but their domain-specific (e.g., tensor-specific) primitives are difficult to adapt to the complex software forms and copy requirements on CPUs.

The applicability of Copier in more OS services and scenarios. Besides the OS services discussed (send/recv syscalls, Binder IPC and page fault handler), we believe that Copier can benefit more OS services, including file I/O, device virtualization, tiered memory management, etc. As a unified OS service, Copier provides a general interface for all OS services to asynchronously copy memory. We also believe that Copier demonstrates potential applicability in a wider range of scenarios, such as disaggregated memory and tiered memory systems, etc.

The security of Copier. Copier avoids Time-of-Check-to-Time-of-Use risks of zero-copy because, instead of sharing the ownership, it ensures the data has been copied to private buffers, which cannot be modified by attackers, before being checked. In scenarios where components that need to transfer data do not trust each other, Copier can provide a secure and efficient data passing method. LionsOS [104] proposes an OS copy service (also named Copier) to ensure secure data transfer between untrusted components, but it does not expose the opportunities of asynchrony and optimizations.

8 Related Work

Async syscalls and I/O parallelize OS and app execution, e.g., FlexSC [26, 105], etc. [106, 107]. The multikernel [108] proposes async messaging. Demikernel [55] proposes async queue-based I/O abstractions. SKYROS [109] accelerates replicated storage by making nil-externalizing operations async, and Lazylog [110] abstraction makes log ordering async. We are the first to utilize the async idea to *copy as an OS service*.

Zero-copy for specific scenarios. IX [111], etc.[112–115] propose new APIs to achieve zero-copy I/O stack. Cornflakes [116] trades off between zero-copy serialization and copy-based ones. They have similar issues as discussed in §2.2, which can be mitigated with Copier.

Fast kernel-userspace communication. Queues and shared memory achieve lightweight K-U communication [25–27]. μSWITCH [117] proposes implicit context switching. mTCP [115] consolidates syscalls into a shared memory access. Copier adopts the same idea, and proposes copy-use pipeline to further exploit async copy benefits.

Rearchitecting library-based solutions as system services. mRPC [118] implements RPC marshaling as a system service. Shenango et al. [119, 120] propose standalone scheduling services. FSP [121] rearchitects FS as a user process for fast development. Copier follows this line but focuses on a different challenge, *how to copy memory efficiently*.

Hardware-assisted copy optimizations. Mondrian [122, 123], etc. [124] share memory through hardware capability. M3 [125] proposes DTU for effective copy. RowClone [57] adds memcpy primitive to ISA. MC^2 [100] and SDAM [101] modify the memory controller to support lazy copy and software-defined address mapping. They require critical hardware changes. DTO [28] provides a library to accelerate large userspace copy with Intel DSA [56], but fails to maintain fairness and isolation when multiple apps use DSA.

9 Conclusion

We present Copier, the first effort to reconstruct asynchronous memory copy as a standalone OS service. Copier leverages Copy-Use windows and its OS service role to optimize copies throughout the entire system. Many novel design spaces are explored, including OS abstractions, hardware dispatcher mechanisms, copy absorption, management and scheduling, and programming primitives. Our practices confirm Copier's generality and efficiency.

Acknowledgments

We are grateful to our shepherd Jason Nieh for his detailed suggestions, which significantly improved the paper. We thank the anonymous SOSP reviewers for their constructive feedback. We thank Haoru Zhao, whose suggestions significantly improved the paper, and Jiawei Wang, who helped us identify copy-intensive scenarios in HarmonyOS 5.0. We also thank Erhu Feng, Jiahao Chen, Dingji Li, Fangnuo Wu, and Mingkai Dong for their valuable suggestions. This work was supported in part by the Fundamental Research Funds for the Central Universities.

Corresponding authors: Dong Du (dd_nirvana@sjtu.edu.cn), Yubin Xia (xiayubin@sjtu.edu.cn), and Haibo Chen (haibochen@sjtu.edu.cn).

A Simulation Proof of the Equivalence between Async Copy with *csync* and Sync Copy [6]

We prove at the protocol level that the semantics of async copy contextually refine those of sync copy. Specifically, we consider the refinement relation between two versions of an arbitrary multi-threaded program ($P \stackrel{\text{def}}{=} C_1 \| \ldots \| C_n$), one using memcpy, written as (P_{sync}), and the other using amemcpy and csync, written as (P_{async}). To prove P_{async} refines P_{sync}, we only need to show that, for each thread, there is a rely-guarantee-based simulation (RGSim) [126] from C_{i-sync} to $C_{i-async}$, i.e., for each step $C_{i-async}$ makes, C_{i-sync} can make corresponding steps to preserve the consistency relation and generate the same observable behavior.

Program transformation. We define the transformation from C_{i-sync} to $C_{i-async}$ as follows: (1) amemcpy replaces memcpy in all occurrences, (2) C_{i-sync}'s code that corresponds to the handler is replaced by a placeholder in $C_{i-async}$, which identifies the point where C_{i-sync} executes the handler, (3) before reading or writing the dst range [addr, addr+len), a csync(addr, len) (or csync_all()) is inserted, (4) before writing to the source (src) range [addr, addr+len], a $csync(addr - src + dst, len)$ (or csync_all()) is inserted, and (5) before a destination (dst) range is visible to another thread, a csync_all() is inserted. Note that amemcpy does not count as a read or write access.

State model. For P_{async}, the memory maps each address to a value (v) or a list of value pairs ($vl \stackrel{\text{def}}{=} (v, id) :: vl'$) in case there are pending amemcpy operations on the address. Each value pair consists of a value and the amemcpy's identifier. When an amemcpy operation starts to copy memory, it is assigned a unique identifier. A csync operation on an address truncates the list to the *latest* value, i.e., the one with the largest identifier. Reading a list of value pairs (only allowed by amemcpy, other reads use csync first) returns the latest value. Writing to a list of value pairs (only allowed by amemcpy) appends a new value pair to the list.

P_{async} also has an auxiliary state, i.e., a list of amemcpy operations and their status. Each item in the list is (args, id, csynced, passph, handler). The first two fields are the arguments and the identifier of the amemcpy operation. The third field is true or false, indicating whether the amemcpy has been fully csynced or not. The fourth field is also a Boolean value, marking whether the execution has passed the placeholder or not. The last field records the handler for future execution.

Semantics modelling. We model memcpy(dst, src, size) as a **while**-loop, with each round copying a byte atomically from src to dst until size bytes are copied. In cases where src and dst have overlapping parts, memcpy(dst, src, size) will copy either forward or backward to avoid overwriting data.

We model amemcpy(dst, src, size, handler) as follows: (1) a while loop with each round atomically reading a byte from src and writing to dst until size bytes are copied (same as memcpy except that writing will append to the value list), and (2) upon invocation, recording the amemcpy's status (args, id, false, false, handler) in the auxiliary list.

The step of the placeholder updates the passph field to true. If the csynced field is true, we will execute the handler.

The csync operation atomically truncates the target address range. Once an amemcpy is fully csynced, (1) we update its csynced field to true and (2) if the passph field is already true, we execute the handler.

Consistency relation. If passph is true and csynced is false, it means some free operation in C_{i-sync} has executed while the corresponding handler in $C_{i-async}$ has not. We collect such freed memory in all threads as M_{free}. Then $dom(M_{async}) = dom(M_{sync} \uplus M_{free})$.

For each memory address a in M_{sync}, its value, $M_{sync}(a)$, equals the corresponding latest value in M_{async}, $latest(M_{async}(a))$, where $latest((v, id) :: vl')$ returns the value with the largest id, and $latest(v)$ returns v.

Simulation proof.

Proof.

We divide the steps of $C_{i-async}$ into the following categories: steps of amemcpy, step of csync, step of placeholder, reading or writing a dst range of amemcpy, writing to a src range of amemcpy, and other steps (non-amemcpy related steps).

Rely and guarantee conditions specify that each step preserves consistency relation. Given consistency relation holds before the step, we prove it still holds after each step of $C_{i-async}$.

1 **Steps of** amemcpy. For each byte copied in amemcpy, C_{i-sync} executes memcpy to copy the corresponding byte. The memory domains do not change. For each address of dst range, $M_{sync}(a) = latest(M_{async}(a))$. The value of other addresses does not change.
2 **Step of csync**. After truncating the value list, $M_{sync}(a) = M_{async}(a)$. In the case when an amemcpy is fully csynced and passph is true, the execution of the handler will free the part of memory that has already been freed in C_{i-sync}. Therefore, memory domains still obey the consistency relation.
3 **Step of placeholder**. C_{i-sync} correspondingly executes the free operation. If csynced is false, M_{free} will include the part of memory newly freed in C_{i-sync}, thus $dom(M_{async}) = dom(M_{sync} \uplus M_{free})$ still holds. If csynced is true, then the handler will be executed. Consequently, C_{i-sync} and $C_{i-async}$ will free the same memory, thus the consistency relation still holds.
4 **Reading or writing to dst range**. C_{i-sync} correspondingly executes reads or writes. Because a csync is inserted

[6] The appendix was not peer-reviewed.

before the command, C_{i-sync} and $C_{i-async}$ will read the same value, or after the write, the values are still equal.

5 **Writing to src range**. C_{i-sync} correspondingly executes writes. Because a csync is inserted before the command, the values of the address are still equal.

6 **Non-amemcpy related steps**. C_{i-sync} correspondingly executes the same step. The consistency relation still holds.

According to the compositionality and soundness of RGSim, we prove that P_{async} refines P_{sync}.
Qed.

References

[1] Jingbo Su, Jiahao Li, Luofan Chen, Cheng Li, Kai Zhang, Liang Yang, and Yinlong Xu. 2023. Revitalizing the Forgotten On-Chip DMA to Expedite Data Movement in NVM-based Storage Systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 363–378. https://www.usenix.org/conference/fast23/presentation/su

[2] Jiaxing Qiu, Zijie Zhou, Yang Li, Zhenhua Li, Feng Qian, Hao Lin, Di Gao, Haitao Su, Xin Miao, Yunhao Liu, and Tianyin Xu. 2024. vSoC: Efficient Virtual System-on-Chip on Heterogeneous Hardware. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 558–573. https://doi.org/10.1145/3694715.3695946

[3] Mark Mansi, Bijan Tabatabai, and Michael M. Swift. 2022. CBMM: Financial Advice for Kernel Memory Managers. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 593–608. https://www.usenix.org/conference/atc22/presentation/mansi

[4] Haibo Chen, Xie Miao, Ning Jia, Nan Wang, Yu Li, Nian Liu, Yutao Liu, Fei Wang, Qiang Huang, Kun Li, Hongyang Yang, Hui Wang, Jie Yin, Yu Peng, and Fengwei Xu. 2024. Microkernel Goes General: Performance and Compatibility in the HongMeng Production Microkernel. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 465–485. https://www.usenix.org/conference/osdi24/presentation/chen-haibo

[5] Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. 2021. NrOS: Effective Replication and Sharing in an Operating System. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 295–312. https://www.usenix.org/conference/osdi21/presentation/bhardwaj

[6] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 460–477. https://doi.org/10.1145/3132747.3132770

[7] Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. 2022. zIO: Accelerating IO-Intensive Applications with Transparent Zero-Copy IO. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 431–445. https://www.usenix.org/conference/osdi22/presentation/stamler

[8] 2025. Redis Documentation. https://redis.io/docs/.

[9] 2024. Tinyproxy: lightweight http(s) proxy daemon. https://tinyproxy.github.io.

[10] 2024. Nginx. https://nginx.org/en/.

[11] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, USA, 307–320. https://www.usenix.org/legacy/events/osdi06/tech/full_papers/weil/weil_html/

[12] 2024. MongoDB. https://www.mongodb.com/.

[13] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 65–77. https://doi.org/10.1145/3452296.3472888

[14] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. 2019. XPC: architectural support for secure and efficient cross process call. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 671–684. https://doi.org/10.1145/3307650.3322218

[15] 2022. Fast memcpy, A System Design. https://www.sigarch.org/fast-memcpy-a-system-design/.

[16] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. 2017. Mallacc: Accelerating Memory Allocation. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 33–45. https://doi.org/10.1145/3037697.3037736

[17] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 158–169. https://doi.org/10.1145/2749469.2750392

[18] Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanovic, and Parthasarathy Ranganathan. 2023. Profiling Hyperscale Big Data Processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 47, 16 pages. https://doi.org/10.1145/3579371.3589082

[19] 2018. Avoid barrier_nospec() in 64-bit copy_from_user() (Linux patch). https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0fc810ae3ae110f9e2fcccce80fc8c8d62f97907.

[20] 2022. AArch64: Add SVE memcpy (glibc patch). https://sourceware.org/pipermail/glibc-cvs/2024q2/084846.html.

[21] 2022. Imporve 64bit memcpy performance for Haswell CPU with AVX instruction (glibc patch). https://sourceware.org/pipermail/libc-alpha/2014-June/051959.html.

[22] Changwoo Min, Woonhak Kang, Mohan Kumar, Sanidhya Kashyap, Steffen Maass, Heeseung Jo, and Taesoo Kim. 2018. Solros: a data-centric operating system architecture for heterogeneous computing. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 36, 15 pages. https://doi.org/10.1145/3190508.3190523

[23] 2025. Intel Advanced Vector Extensions 2 (Intel AVX2). https://edc.intel.com/content/www/us/en/design/ipla/software-development-platforms/client/platforms/alder-lake-desktop/12th-generation-intel-core-processors-datasheet-volume-1-of-2/002/intel-advanced-vector-extensions-2-intel-avx2/.

[24] 2024. MSG_ZEROCOPY Document. https://www.kernel.org/doc/html/v4.18/networking/msg_zerocopy.html.

[25] Jonathan Corbet. 2020. The rapid growth of io_uring. https://lwn.net/Articles/810414/.

[26] Livio Soares and Michael Stumm. 2010. FlexSC: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 33–46. https://www.usenix.org/conference/osdi10/flexsc-flexible-system-call-scheduling-exception-less-system-calls

[27] Kyu-Jin Cho, Jaewon Choi, Hyungjoon Kwon, and Jin-Soo Kim. 2024. RFUSE: Modernizing Userspace Filesystem Framework through Scalable Kernel-Userspace Communication. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. 141–157.

[28] Reese Kuper, Ipoom Jeong, Yifan Yuan, Ren Wang, Narayan Ranganathan, Nikhil Rao, Jiayu Hu, Sanjay Kumar, Philip Lantz, and Nam Sung Kim. 2024. A Quantitative Analysis and Guidelines of Data Streaming Accelerator in Modern Intel Xeon Scalable Processors. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 37–54. https://doi.org/10.1145/3620665.3640401

[29] 2016. (GNU C library patch) Initial Enhanced REP MOVSB/STOSB (ERMS) support. https://patchwork.ozlabs.org/project/glibc/patch/20160328204830.GA402@intel.com.

[30] Jochen Liedtke. 1993. Improving IPC by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (Asheville, North Carolina, USA) *(SOSP '93)*. Association for Computing Machinery, New York, NY, USA, 175–188. https://doi.org/10.1145/168619.168633

[31] 2024. sendfile(2) — Linux manual page. https://man7.org/linux/man-pages/man2/sendfile.2.html.

[32] 2024. vmsplice(2) — Linux manual page. https://man7.org/linux/man-pages/man2/vmsplice.2.html.

[33] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: the operating system is the control plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 1–16. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter

[34] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. 2022. XRP: In-Kernel Storage Functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 375–393. https://www.usenix.org/conference/osdi22/presentation/zhong

[35] 2024. HarmonyOS 5.0.0 Release Notes. https://developer.huawei.com/consumer/en/doc/atomic-releases/atomic-releasenotes-500.

[36] 2024. Protocol Buffers Documentation. https://protobuf.dev.

[37] 2025. Deep Dive into Android IPC/Binder Framework at Android Builders Summit 2013. https://events.static.linuxfound.org/images/stories/slides/abs2013_gargentas.pdf.

[38] 2024. zlib - A Massively Spiffy Yet Delicately Unobtrusive Compression Library. https://github.com/openssl/openssl.

[39] 2024. OpenSSL - TLS/SSL and crypto library. https://github.com/openssl/openssl.

[40] 2025. Libpng is the official PNG reference library. http://www.libpng.org/pub/png/libpng.html.

[41] 2024. ffmpeg - A complete, cross-platform solution to record, convert and stream audio and video. https://ffmpeg.org.

[42] 2010. The new linux 'perf' tools. http://oldvger.kernel.org/~acme/perf/lk2010-perf-paper.pdf.

[43] 2024. F-Stack: High Performance Network Framework Based On DPDK. http://www.f-stack.org/.

[44] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP Acceleration as an OS Service. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 24, 16 pages. https://doi.org/10.1145/3302424.3303985

[45] 2022. Intel Architecture Instruction Set Extensions and Future Features. https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html.

[46] 2018. Why can the kernel not use SSE/AVX registers and instructions? https://unix.stackexchange.com/questions/475956.

[47] 2006. Accelerating High-Speed Networking with Intel I/O Acceleration Technology. https://www.intel.com/content/dam/doc/white-paper/i-o-acceleration-technology-paper.pdf.

[48] Hsiao-keng Jerry Chu. 1996. Zero-copy TCP in Solaris. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference* (San Diego, CA) *(ATEC '96)*. USENIX Association, USA, 21. https://www.usenix.org/conference/usenix-1996-annual-technical-conference/zero-copy-tcp-solaris

[49] Alex Markuze, Igor Smolyar, Adam Morrison, and Dan Tsafrir. 2018. DAMN: Overhead-Free IOMMU Protection for Networking. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 301–315. https://doi.org/10.1145/3173162.3173175

[50] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. 2021. Breakfast of champions: towards zero-copy serialization with NIC scatter-gather. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 199–205. https://doi.org/10.1145/3458336.3465287

[51] 2020. Tencent Photo Cache traces - SNIA. http://iotta.snia.org/traces/parallel/27476.

[52] 2020. Twitter Memcached traces - SNIA. http://iotta.snia.org/traces/key-value.

[53] 2020. Alibaba Block Traces. https://github.com/alibaba/block-traces.

[54] 2024. Kernel Samepage Merging. https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html.

[55] 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. 2021. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 195–211. https://doi.org/10.1145/3477132.3483569

[56] 2025. Intel Data Streaming Accelerator (Intel DSA). https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/data-streaming-accelerator.html.

[57] 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. 2013. RowClone: fast and energy-efficient in-DRAM bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (Davis, California) *(MICRO-46)*. Association for Computing Machinery, New York, NY, USA, 185–197. https://doi.org/10.1145/2540708.2540725

[58] 2024. NAPI Document. https://docs.kernel.org/networking/napi.html.

[59] 2022. io_uring: add napi busy polling support. https://lwn.net/Articles/915657/.

[60] 2024. cgroups - Linux control groups. http://man7.org/linux/man-pages/man7/cgroups.7.html.

[61] 2025. Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io.

[62] Chandandeep Singh Pabla. 2009. Completely fair scheduler. *Linux J.* 2009, 184, Article 4 (Aug. 2009). https://dl.acm.org/doi/fullHtml/10.5555/1594371.1594375

[63] 2025. Linux Virtual Memory Area (VMA). https://litux.nl/mirror/kerneldevelopment/0672327201/ch14lev1sec2.html.

[64] 2019. Explicit pinning of user-space pages. https://lwn.net/Articles/807108/.

[65] 2024. DPDK. https://www.dpdk.org/.

[66] 2021. Storage Performance Development Kit. https://spdk.io/.

[67] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 1–16. https://www.usenix.org/conference/nsdi19/presentation/kalia

[68] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 153–167. https://doi.org/10.1145/3132747.3132772

[69] Huangshi Tian, Suyi Li, Ao Wang, Wei Wang, Tianlong Wu, and Haoran Yang. 2022. Owl: performance-aware scheduling for resource-efficient function-as-a-service cloud. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 78–93. https://doi.org/10.1145/3542929.3563470

[70] 2024. memcached - a distributed memory object caching system. https://www.memcached.org.

[71] 2024. Apache HTTP Server - the Number One HTTP Server On The Internet. https://httpd.apache.org.

[72] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 28. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[73] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, New York, USA) *(WBIA '09)*. Association for Computing Machinery, New York, NY, USA, 62–71. https://doi.org/10.1145/1791194.1791203

[74] 2025. Valgrind Memcheck: a memory error detector. https://valgrind.org/docs/manual/mc-manual.html.

[75] 2015. AddressSanitizer Manual of the Poisoning APIs. https://github.com/google/sanitizers/wiki/AddressSanitizerManualPoisoning.

[76] 2024. Kernel Address Sanitizer (KASAN) — The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[77] Zhiyuan Guo, Zijian He, and Yiying Zhang. 2023. Mira: A Program-Behavior-Guided Far Memory System. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 692–708. https://doi.org/10.1145/3600006.3613157

[78] 2024. Multi-Level Intermediate Representation Overview. https://mlir.llvm.org/docs/Tutorials/DataFlowAnalysis/.

[79] 2024. Passes - MLIR. https://mlir.llvm.org/docs/Passes/.

[80] 2024. Checksum Offloads. https://www.kernel.org/doc/html/v5.8/networking/checksum-offloads.html.

[81] 2024. OFED Documentation Rev 4.9-2.2.4.0 LTS - Checksum Offload. https://docs.nvidia.com/networking/display/mlnxofedv492240/checksum+offload.

[82] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. 2019. A fork() in the road. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 14–22. https://doi.org/10.1145/3317550.3321435

[83] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. 2008. Difference engine: harnessing memory redundancy in virtual machines. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 309–322. https://doi.org/10.5555/1855741.1855763

[84] 2024. Parcel. https://developer.android.com/reference/android/os/Parcel.

[85] 2016. Intel Xeon Processor E5-2650 v4. https://www.intel.com/content/www/us/en/products/sku/91767/intel-xeon-processor-e52650-v4-30m-cache-2-20-ghz/specifications.html.

[86] 2023. Linus Torvalds Cleans Up The x86 Memory Copy Code For Linux 6.4. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=a5624566431de76b17862383d9ae254d9606cba9.

[87] Zhe Zhou, Yanxiang Bi, Junpeng Wan, Yangfan Zhou, and Zhou Li. 2023. Userspace Bypass: Accelerating Syscall-intensive Applications. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 33–49. https://www.usenix.org/conference/osdi23/presentation/zhou-zhe

[88] 2018. Zero-copy TCP receive. https://lwn.net/Articles/752188/.

[89] 2021. Header-Data Split Architecture. https://learn.microsoft.com/en-us/windows-hardware/drivers/network/header-data-split-architecture.

[90] 2024. Redis Benchmark Documentation. https://redis.io/docs/management/optimization/benchmarks/.

[91] 2020. Intel Xeon Gold 6238R Processor. https://www.intel.com/content/www/us/en/products/sku/199345/intel-xeon-gold-6238r-processor-38-5m-cache-2-20-ghz/specifications.html.

[92] 2024. Knative is an Open-Source Enterprise-level solution to build Serverless and Event Driven Applications. https://knative.dev/docs/.

[93] 2024. gRPC Documentation. https://grpc.io/docs/.

[94] 2024. zlib replacement with optimizations for "next generation" systems. https://github.com/zlib-ng/zlib-ng.

[95] 2006. Add memcpy_uncached_read, a memcpy that tries to reduce cache pressure. https://lwn.net/Articles/213971/.

[96] 2022. DPDK non-temporal memcpy. https://mails.dpdk.org/archives/dev/2022-August/247795.html.

[97] Rishabh Iyer, Katerina Argyraki, and George Candea. 2024. Automatically Reasoning About How Systems Code Uses the CPU Cache. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 581–598. https://www.usenix.org/conference/osdi24/presentation/iyer

[98] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. 2010. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European Conference on Computer Systems* (Paris, France) *(EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 335–348. https://doi.org/10.1145/1755913.1755947

[99] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. 2021. DMon: Efficient Detection and Correction of Data Locality Problems Using Selective Profiling. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 163–181. https://www.usenix.org/conference/osdi21/presentation/khan

[100] Aditya K Kamath and Simon Peter. 2024. MC^2: Lazy MemCopy at the Memory Controller. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 1112–1128. https://doi.org/10.1109/ISCA59077.2024.00084

[101] Jialiang Zhang, Michael Swift, and Jing (Jane) Li. 2022. 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* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 70–83. https://doi.org/10.1145/3503222.3507774

[102] 2022. NVIDIA H100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c.

[103] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. 2021. Ascend: a Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing : Industry Track Paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 789–801. https://doi.org/10.1109/HPCA51647.2021.00071

[104] Gernot Heiser, Ivan Velickovic, Peter Chubb, Alwin Joshy, Anuraag Ganesh, Bill Nguyen, Cheng Li, Courtney Darville, Guangtao Zhu, James Archer, Jingyao Zhou, Krishnan Winter, Lucy Parker, Szymon Duchniewicz, and Tianyi Bai. 2025. Fast, Secure, Adaptable: LionsOS Design, Implementation and Performance. arXiv:2501.06234 [cs.OS] https://arxiv.org/abs/2501.06234

[105] Livio Soares and Michael Stumm. 2011. Exception-less system calls for event-driven servers. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Portland, OR) *(USENIX ATC'11)*. USENIX Association, USA, 10. https://www.usenix.org/conference/usenixatc11/exception-less-system-calls-event-driven-servers

[106] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. 2015. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. *ACM Trans. Comput. Syst.* 32, 4, Article 10 (Jan. 2015), 47 pages. https://doi.org/10.1145/2699681

[107] Tim Harris, Martin Abadi, Rebecca Isaacs, and Ross McIlroy. 2011. AC: composable asynchronous IO for native languages. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 903–920. https://doi.org/10.1145/2048066.2048134

[108] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 29–44. https://doi.org/10.1145/1629575.1629579

[109] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. Exploiting Nil-Externality for Fast Replicated Storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 440–456. https://doi.org/10.1145/3477132.3483543

[110] Xuhao Luo, Shreesha G. Bhat, Jiyu Hu, Ramnatthan Alagappan, and Aishwarya Ganesan. 2024. LazyLog: A New Shared Log Abstraction for Low-Latency Applications. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 296–312. https://doi.org/10.1145/3694715.3695983

[111] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: a protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 49–65. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay

[112] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 43–56. https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata

[113] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: a new programming interface for scalable network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 135–148. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/han.

[114] Yang Zhan, Alexander Conway, Yizheng Jiao, Nirjhar Mukherjee, Ian Groombridge, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. 2020. How to Copy Files. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 75–89. https://www.usenix.org/conference/fast20/presentation/zhan

[115] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 489–502.

[116] Deepti Raghavan, Shreya Ravi, Gina Yuan, Pratiksha Thaker, Sanjari Srivastava, Micah Murray, Pedro Henrique Penna, Amy Ousterhout, Philip Levis, Matei Zaharia, and Irene Zhang. 2023. Cornflakes: Zero-Copy Serialization for Microsecond-Scale Networking. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 200–215. https://doi.org/10.1145/3600006.3613137

[117] Dinglan Peng, Congyu Liu, Tapti Palit, Pedro Fonseca, Anjo Vahldiek-Oberwagner, and Mona Vij. 2023. μSwitch: Fast Kernel Context Isolation with Implicit Context Switches. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2956–2973.

[118] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Thomas Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. 2023. Remote Procedure Call as a Managed System Service. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 141–159. https://www.usenix.org/conference/nsdi23/presentation/chen-jingrong

[119] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[120] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 281–297. https://www.usenix.org/conference/osdi20/presentation/fried

[121] Jing Liu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Sudarsun Kannan. 2019. File Systems as Processes. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotstorage19/presentation/liu

[122] Emmett Witchel, Josh Cates, and Krste Asanović. 2002. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California) *(ASPLOS X)*. Association for Computing Machinery, New York, NY, USA, 304–316. https://doi.org/10.1145/

605397.605429

[123] Emmett Witchel, Junghwan Rhee, and Krste Asanović. 2005. Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom) *(SOSP '05)*. ACM, New York, NY, USA, 31–44. https://doi.org/10.1145/1095810.1095814

[124] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. 20–37. https://doi.org/10.1109/SP.2015.9

[125] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. 2016. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. ACM, New York, NY, USA, 189–203. https://doi.org/10.1145/2872362.2872371

[126] Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 455–468. https://doi.org/10.1145/2103656.2103711