

Computation Is Fast, Use THREADLET!

Efficient Threading for μs-Scale Computing via OS/Hardware Co-Design

Yiming Yao[†,¶] Xiaohe Qin[†,¶] Yi Fan[†] Yuanlong Li[‡] Kang Hu[†] Liujia Li[†]
Xiaolin Wang[†,¶] Tao Xie[†,¶] Zhenlin Wang[*] Yuval Tamir[§] Yingwei Luo[†,¶] Diyu Zhou[†,¶]
[†] *Peking University*, [‡] *EPFL*, [*] *Michigan Technological University*, [§] *UCLA*
[¶] *Beijing Tongming Lake Information Technology Application Innovation Center (TLAIC)*

Abstract

The conventional threading model multiplexes software threads onto hardware cores. This model inherently suffers from the overhead of 1) context switching, 2) scheduling, and 3) system event notifications (*e.g.*, I/O interrupts). As computing enters μs-scale, such overheads become the key bottleneck of a wide range of domains: networking, concurrency control, and microservice scheduling, etc. Existing software solutions, such as cooperative scheduling and polling, sacrifice correctness and/or CPU efficiency, while hardware offers only point solutions with suboptimal performance.

To eliminate the threading overheads without introducing difficult tradeoffs, our key insight is that hardware should directly expose schedulable, basic units of execution (*i.e.*, software threads) instead of computational resources (*i.e.*, cores). We propose THREADLET, a new hardware abstraction that is lighter than a software thread by removing uncritical state (*e.g.*, floating-point registers). A core caches a few active THREADLETS in its register file, while spilling the rest to the memory hierarchy. Thus, context switching incurs minimal overhead by simply resuming execution with the next THREADLET's state. Moreover, THREADLET comes with hardware support that can express diverse OS scheduling policies, enabling fast and flexible scheduling. Finally, as a hardware abstraction, THREADLETS directly interface with microarchitectural mechanisms (*e.g.*, cache coherence) for efficient event notifications. We build a supporting architecture on an FPGA and a research OS for THREADLET, outperforming prior works by 3× to 47× on real-world applications. Our artifact is publicly available at https://github.com/TELOS-syslab/Threadlet.



This work is licensed under a Creative Commons Attribution 4.0 International License.
SOSP '26, September 29–October 2, 2026, Prague, Czech Republic
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2585-2/26/09
https://doi.org/10.1145/3830418.3843863

1 Introduction

Conventional computing offers a split threading model. Hardware exposes cores as computational resources. Privileged software (*e.g.*, an OS) multiplexes threads, each with its state, to manage core resources. Such a threading model dates back to the 1960s [17, 20, 49], and has since enjoyed wide success.

Yet, this threading model inherently suffers from four types of inefficiencies. Multiplexing threads over cores incurs 1) context switching (*i.e.*, register save/restore), and 2) scheduling overhead (*i.e.*, deciding the next thread to run). Moreover, computing frequently involves event notifications, including 3) thread-to-thread notification (*e.g.*, a thread waking up another), 4) device-to-thread notification (*e.g.*, I/O interrupts). As threads are transparent to hardware, privileged software must mediate these notifications, incurring extra delays.

As hardware advances push computing into μs-scale [8, 13, 37, 39, 41, 57], a wide range of domains suffer from these inefficiencies. Data center networking is a typical example susceptible to all four of them. NIC I/O incurs device-to-thread notifications, while dispatching requests across worker threads leads to thread-to-thread notifications. Further, to maximize CPU efficiency, data centers [24, 25, 58] often co-locate other workloads with networking applications to utilize idle cycles when no request arrives. Yet, co-location leads to context switching and scheduling overheads. Other affected domains include concurrency control [22, 109], microservice scheduling [12, 40], system calls [93], and microkernel IPC [10, 21, 65].

Existing solutions mitigate these inefficiencies, but introduce hard-to-resolve tradeoffs. For example, software minimizes context switching overhead with cooperative scheduling [59, 71, 108], but can lead to starvation. Moreover, software uses polling for both device-to- and thread-to-thread notifications. Yet, despite optimizations like spin-then-park [19, 29], polling still either wastes CPU cycles or harms performance.

Hardware works [37, 55, 67, 74, 88, 90] only offer point solutions to a subset of these inefficiencies, while important scenarios such as networking discussed above, demand a holistic solution. Moreover, these point solutions are suboptimal. For

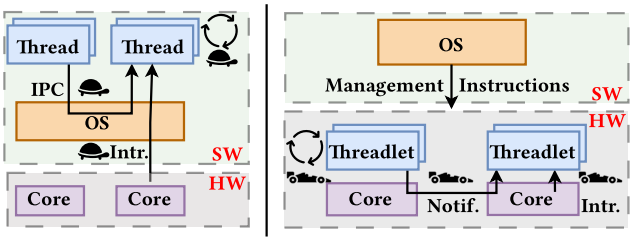


Figure 1: A comparison of conventional threading model and Loom.

example, works on hardware context switching [38, 69, 70] still swap thread state to memory and thus suffer from the key overhead of moving data between registers and memory. Also, works on hardware thread scheduling [18, 37, 68–70] often hard-code policies, limiting flexibility.

This paper presents threadlet, a new threading abstraction, and Loom, its supporting architecture, which together address all four inefficiencies. Moreover, we design our systems to be 1) **generic**, applying to a wide range of domains, 2) **flexible**, avoiding fixed policies, and 3) **practical**, realizable through lightweight extensions to commodity hardware.

Our insight is that a unified, rather than split, threading model addresses all four inefficiencies, as shown in Figure 1. Hardware no longer exposes the computation-resource abstraction (*i.e.*, cores) but rather the abstraction of schedulable units of execution (*i.e.*, software threads). This change of perspective eliminates multiplexing, and thus the associated context switching and scheduling overheads. Moreover, exposing threads to hardware allows them to directly interface with microarchitectural mechanisms (*e.g.*, cache coherence), thus eliminating software mediation for event notification.

We propose the threadlet abstraction as a hardware representation of software threads. As its name suggests, a threadlet is lightweight, maintaining only the critical state, including 1) general-purpose registers; 2) key control registers (*e.g.*, program counter); and 3) scheduling semantics (*e.g.*, priority, deadline, event triggers). The OS only manages threadlets on rare occasions, such as creation/deletion, and setting scheduling policies, via a new set of instructions; the accompanying hardware, Loom, handles the rest.

Loom[1] addresses a key design challenge: enabling a large number of threadlets per core, without sacrificing common-case performance. Software threads are abundant, and so must be threadlets. Google data centers [53] average 50 threads per core, with some reaching around 500. A conventional design [104] would require a large register file, increasing instruction execution latency by tens of cycles, which is unacceptable for μs-scale computing. Loom exploits a key observation: most of the time, only a small number of threads are runnable (often <2%; see Figure 5).

Hence, while Loom supports hundreds of threadlets per core, the register file stores only a few (8 in the current design) runnable high-priority threadlets, incurring minimal latency overhead. The rest of the threadlets reside in the memory hierarchy. Critically, as threadlets carry scheduling semantics, Loom can efficiently predict which ones are likely to run next. Loom therefore asynchronously prefetches off-chip threadlets into an on-chip staging buffer, minimizing performance impact. With that, Loom only performs context switching among the threadlets in the register file and incurs minimal overhead by eliminating state save/restore.

For efficient scheduling, Loom proposes two scheduling primitives that are 1) easy to implement in hardware, 2) but can be composed to express various policies. First, each threadlet is associated with a three-tuple: a) priority, b) time slice, and c) deadline. Higher-priority threadlets preempt lower-priority ones, while deadline-aware round-robin scheduling is used for threadlets with the same priority. Second, Loom allows an OS to register a dedicated scheduling threadlet that executes a scheduling policy when key events occur (*e.g.*, a threadlet wakes up or blocks). Together, these primitives minimize OS involvement, and thus its associated scheduling overhead. Moreover, despite many threadlets, scheduling remains highly efficient by optimizing for the common case of a few runnable ones.

Finally, Loom unifies the two types of notifications using memory address monitoring [8, 34]. Under this scheme, a waiting threadlet registers a memory address with Loom and yields the core. Loom then snoops cache coherence events and, upon a write to the monitored address, wakes the threadlet. This design is strictly faster than interrupts (or any other software mechanisms) and matches polling, since memory writes are the earliest signals of ready data.

We built Loom using the Rocket RISC-V core on an FPGA and further developed a research OS, LoomOS. Thanks to the hardware support, LoomOS resolves long-standing tradeoffs across a wide range of domains, providing: 1) μs-scale (as polling) work-conserving (as interrupts) networking; 2) efficient blocking concurrency control; 3) delegation-style system calls; and 4) nanosecond-scale EEVDF scheduling. Evaluation shows that LoomOS outperforms prior works by 3× to 47× across these domains on real-world applications.

Related work (§7). Loom advances multithreading processors [91, 92, 102, 103]: Threadlets depart from the hardware thread abstraction, thereby solving systems problems (*e.g.*, software thread scheduling and I/O), rather than exploiting microarchitecture. Many ideas overlap with a prior workshop [34] and a vision paper [8]. We extend these high-level proposals and make technical contributions to realize them.

In summary, this paper makes the following contributions.

- **Insights.** We argue that hardware should directly expose schedulable units of execution (*i.e.*, threads) as the abstraction, rather than resources (*i.e.*, cores).
- **Interface design.** We design threadlet and the scheduling primitives to be simple enough for hardware realization, while preserving flexibility for software.

[1] A loom "manages" numerous threads, weaving them into new patterns.

- **Practical realization.** We show that lightweight extensions to commodity hardware solve long-standing challenges across many system domains.

2 Inefficiencies in Current Threading

2.1 The Current Threading Abstractions

The conventional threading model consists of two layers. Hardware exposes the core abstraction, a form of resource, offering the capability to compute, access main memory, and perform I/O. Privileged software (e.g., an OS) manages the core resource by multiplexing threads over it. This widely used model has been unchanged for decades [17, 20, 49].

Inefficiencies. However, this model adds an extra level of indirection, namely threads, that is largely invisible to hardware, leading to four forms of inefficiency.

First, multiplexing threads on a core incurs the extra overhead of 1) context switching, *i.e.*, saving and restoring threads' architectural state; and 2) scheduling, *i.e.*, the OS scheduler decides the next thread to run. Moreover, as hardware is unaware of threads, any notification to threads must be mediated by software. Thus, both 3) thread-to-thread (*e.g.*, condition variables, signals) and 4) device-to-thread notifications (*e.g.*, interrupts upon I/O completion) suffer.

Limitation for μs-scale computing. An implicit assumption of the current model is that computations are long-running and involve minimal interaction with external entities. Hence, these four overheads are trivial. However, faster CPUs, memory, and I/O devices [32, 55, 90] have pushed computation to μs-scale [8, 37], thereby breaking this assumption.

Hence, as further elaborated in §5, many existing research works, across a wide range of domains, essentially address these four inefficiencies. For example, computation often involves preempting long-running tasks to prioritize short ones, as in networking runtimes, where tasks correspond to requests, and datacenter scheduling, where tasks correspond to applications. Such preemption requires fast context switching and scheduling, leading to much research in networking and/or datacenter scheduling [25, 39, 41, 57, 75]. Moreover, both microkernels [10, 21, 28, 65] and concurrency control [22, 80, 109] require fast thread-to-thread notifications for IPC and wakeups. These domains thereby form the key use cases of Loom.

To quantify the four inefficiencies, we use our evaluated networking workloads (§6.3) as a representative example. With these workloads, device-to-thread notification and thread-to-thread notification can happen up to 900 thousand times per second. Furthermore, at peak load, the average context switching interval is as small as 5.5 μs.

We next experimentally analyze these four overheads and discuss the limits of existing software/hardware solutions.

2.2 High Scheduling and Context-Switch Overheads

Environment. Across the section, we evaluate with an Intel Xeon Silver 4216 machine, running Ubuntu 24.04.2 and Linux kernel 6.12.20. We report the average of 2000 runs.

Workload. To measure context switching and scheduling overheads, we use a microbenchmark that runs 64 threads on a core, each executing an idle loop for 5 μs, and then yielding the core. We instrument the code with `rdtsc`.

Results. Figure 2(a) and (b) show the results, from which we draw the following observations. First, both context switching (0.6 μs) and scheduling (1.0 μs) incur high, μs-scale overheads. Second, even in this optimal case with almost no cache pollution, saving and restoring architectural state to and from memory remains expensive. Third, due to its inherent complexity, even with optimized data structures such as red-black trees, the current Linux scheduling policy EEVDF [98] still incurs high overhead (thread descriptor update: 0.3 μs, red-black tree access and update: 0.6 μs).

Software solutions. To mitigate both overheads, software solutions propose new abstractions, such as user-level threads [40, 53, 71, 75, 79, 83] and coroutines [51, 108], to bypass the kernel scheduling. While elegant, these solutions suffer from the following limitations.

Limitations in context switching. User-level threads often adopt cooperative scheduling [30, 59, 71, 108], which, in the extreme case, turns context switching into function calls. It thus minimizes the architectural state that must be saved and restored, thereby reducing the associated overhead.

However, cooperative scheduling entirely depends on tasks being well-behaved. A buggy or malicious task can easily starve the entire system, harming correctness. Moreover, it is not general-purpose, as it requires significant manual effort to handle 1) blocking operations via asynchronous programming [72, 108], and 2) long-running tasks by inserting yield points [51]. For these reasons, we argue that cooperative scheduling is not a broad solution for μs-scale computing.

Limitations in scheduling. Software solutions to scheduling inefficiency involve using simple, customized scheduling policies. This approach effectively reduces scheduling overhead, at the cost of reduced generality, thereby resulting in suboptimal scheduling outside the target scenario.

For example, most prior works adopt a combination of round-robin and processor-sharing policies [42, 57, 63] to ensure fairness. These policies struggle with generic workloads, as they overlook subtleties that make EEVDF complex, such as 1) past CPU usage, 2) weighted fairness based on priority, and 3) burst bias. Also, customizing policies for specific scenarios incurs duplicated effort, and developing one is non-trivial even for kernel experts [35, 66].

2.3 Slow System Event Notification

Thread-to-thread notification. For thread-to-thread notification, we measure the time to wake up a pinned kernel thread sleeping on a lock. We believe this is (almost) the minimum overhead for thread-to-thread notification, as it avoids many other overheads in cases like signals and futexes.

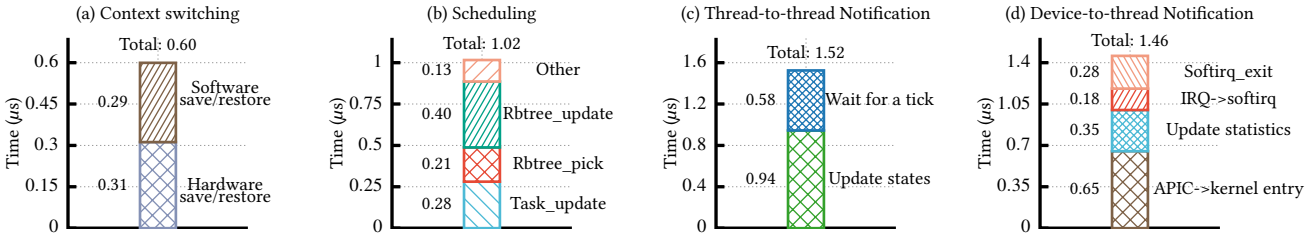


Figure 2: Overhead breakdown of the four inefficiencies in the current threading model.

Figure 2(c) shows the results. The overhead is surprisingly high, at 1.5 μs. It comes from two sources: 1) updating the target thread's kernel state (*e.g.*, leaving its waitqueue and entering a runqueue), and 2) interestingly, after becoming runnable, the thread must still wait until the next scheduler tick (often a timer interrupt) to execute, incurring extra delay.

Device-to-thread notification. For device-to-thread notification, we measure, in Linux NIC processing, the overhead of the interrupt path relative to the polling one.

Figure 2(d) shows the results. An interrupt adds at least 1.46 μs of overhead, due to 1) asserting APIC, 2) updating kernel statistics, 3) raising and entering a softirq, and 4) exiting from the softirq. Steps 3) and 4) are because the handler runs with interrupts disabled and thus must stay short for system responsiveness. Hence, it defers the remaining work to a softirq handler outside the interrupt context.

Software solutions. Interestingly, for both types of notification, current software relies on the same approach: polling for minimal overhead. For example, threads often spin on shared memory for concurrency control and IPC. Fast I/O frameworks [11,55,78,90,97,101], networking runtimes [25,41,75], and even kernel I/O interfaces [100] poll for device events.

However, polling is a compromise. Its well-known drawback is wasting CPU cycles, which must be avoided at scale, as reported by Google [56]. Optimizations such as spin-then-park [19,29] and adaptive polling [86,96] are ad-hoc, as their heuristics often fail, hurting performance. CPU waste is further exacerbated by the fact that polling is effective only when it monopolizes a dedicated core; otherwise, polling even underperforms interrupts [36,50].

2.4 Hardware Solutions

Table 1 summarizes the current hardware proposals for threading inefficiencies. A key limitation is that they address only a subset of the problem, failing to provide a holistic solution, which is a must for key use cases. For example, μs-scale networking requires addressing all four inefficiencies (§5.2). Moreover, even for the inefficiencies they target, they remain suboptimal in performance or generality, as detailed next.

Context Switching. Many prior works support context switching [38,69,70] by swapping thread context to memory. A notable example is Intel's hardware task switching [38], which uses memory-resident Task State Segments to save and restore architectural state. However, such an approach does not address the main overhead: data movement between the processor and memory (§2.2). Another line of work (*e.g.*, nMPRA [26] and hardware multithreading techniques such as SMT [92, 102, 103]) avoids data movement by expanding the register file and replicating pipeline state. However, this approach can support only a small number of hardware contexts (*e.g.*, at most 8 hyperthreads in commodity processors [89]), due to the increased latency to access the register file and area/power overheads.

	Ctx Switch	Scheduling	T-T Notifi.	D-T Notifi.
Intel HTS [38]	✓-	✗	✗	✗
ESMT [69]	✓-	✓-	✗	✗
Q-Zilla [70]	✓-	✓-	✗	✗
nMPRA [26]	✓-	✓-	✓-	✗
μManycore [95]	✓-	✓-	✗	✗
Duplexity [68]	✓-	✓-	✗	✗
NanoPU [37]	✗	✓-	✗	✓-
ALTOCUMULUS [110]	✗	✓-	✗	✗
RPCValet [18]	✗	✓-	✗	✗
RTM [46]	✗	✓-	✓-	✗
SYNCBUS [61]	✗	✓-	✗	✗
HyperPlane [67]	✗	✗	✗	✓-
Loom	✓	✓	✓	✓

Table 1: A summary of current hardware proposals. ✗: no support; ✓-: suboptimal solution; ✓: supported. T-T: thread to thread; D-T: device to thread.

Scheduling. Prior work executes scheduling policies in hardware [18,37,45,46,61,70,84], but targets narrow use cases and hardwires policies. For example, much work on networking runtimes, exemplified by NanoPU [37], embeds specialized policies such as JBSQ [45], an RPC load-balancing policy that does not generalize beyond its target setting. Works on real-time systems [46,61,84] support either fixed policies or a narrow set of priority-based policies [26].

Notifications. Commodity hardware and research works [67, 107] propose idle-loop optimization instructions (*e.g.*, `MONITOR`/`MWAIT` in x86 [38,106]). These instructions are largely designed to reduce the power waste of polling; they do not directly address the CPU waste. User interrupts [30], an emerging hardware feature, reduce the overhead of both notifications [30,33,40,50]. However, user interrupts still incur sub-microsecond overhead [30,50] (*e.g.*, 0.6 μs as reported in [50]), which remains significant.

3 The THREADLET Abstraction

This section presents THREADLET, a new hardware abstraction for efficient threading. We present its design goals (§3.1), overview (§3.2), detailed design and rationale.

	Sw. Threads	Hw. Cores/Threads	Hw. THREADLETS
Num per Core	Many	A few, typically 2	512
General-Purpose Regs	Full	Full	Full
Control/Status Regs	Full	Full	Only critical
FPU/DR/PMC	Full	Full	None
Scheduling Metadata	Full	None	Only critical
Management instructions	No	No	Yes

Table 2: A summary of threads, cores and THREADLETS.

3.1 Design Goals

Our design aims to meet the following goals:

- **Efficient threading.** Our design achieves high performance without sacrificing correctness or CPU efficiency, thereby resolving the long-standing tradeoffs in existing software solutions.
- **General.** As a hardware proposal, our design should benefit a wide range of use cases, rather than being limited to a specific workload.
- **Flexible.** Our design follows the principle of separating policy from mechanism, avoiding hard-coding policies.
- **Practicality.** Our design minimizes changes to existing commodity hardware and the resulting overhead.

3.2 An Overview of the THREADLET Abstraction

Insights. To meet our design goals, our key insight is that these four inefficiencies can be simultaneously addressed by moving the software thread, the abstraction of the unit of execution, into hardware. To realize this, hardware must be aware of not only the architectural state of software threads, but also their scheduling semantics, while exposing interfaces for the OS to manage these units of execution (*e.g.*, create/delete). This removes software threads, the extra level of indirection, thereby minimizing the need for OS mediation. As a result, hardware can directly switch, schedule, and notify these units of execution, thereby addressing the root cause of these inefficiencies (§2.1).

THREADLET. As one way to realize this insight, we design the THREADLET abstraction to represent a unit of execution in hardware. Table 2 compares threads, cores, and THREADLETS; Section 7 further expands the comparison.

3.3 Lightweight Execution Context

To facilitate hardware design, THREADLET's execution context is intentionally much lighter weight than that of software threads or hardware cores, as its name suggests. First, it includes only the critical architectural state: 1) all general-purpose registers; and 2) a few key control and status registers, *e.g.*, the program counter, stack pointer, page table register, and privilege level. Second, it includes only minimal scheduling metadata: execution state (*i.e.*, runnable, blocked), priority, deadline, and time slice. We selected only those fields from Linux's 700-byte thread scheduling structure, as they are sufficient for most scheduling decisions (§3.4).

Design rationales. The lightweight execution context is one of the key enablers to cheaply support many THREADLETS on a single core. As a result, context switching between THREADLETS happens entirely in the core, eliminating data movement between the core and memory, which is the main source of overhead, as we show in §2.2. As a result, THREADLET context switching costs only a few cycles (§6.2).

With this design, THREADLET excludes rarely used architectural state in μs-scale computation, such as debug registers, performance counters, and floating-point (and vector) registers. Rather, all THREADLETS on a core share them, with special instructions selecting the owner THREADLET (Table 3).

We believe this tradeoff is necessary and practical; for example, on modern x86, the total size of floating-point registers is between 256 bytes and 2 KB, being 2× to 16× larger than general-purpose registers. Yet, in most applications, the primary use of floating-point registers is for bulk memory operations such as `memcpy`, which μs-scale computation often avoids through techniques such as zero-copy [81, 111]. This design also aligns with OS practice: lazy FPU context switching reduces switching overhead by deferring FPU state save/restore until a thread executes an FPU instruction [94]. We discuss how to virtualize the excluded state in §5.1.

3.4 Scheduling Semantics

As a unit of execution, THREADLET carries OS scheduling semantics, enabling hardware scheduling of it.

Scheduling points. The THREADLET scheduling is triggered at two scheduling points: 1) at each scheduling tick (configurable, by default 32 cycles), and 2) when the current THREADLET yields, blocks, or terminates. This design balances scheduling overhead and responsiveness.

First, each THREADLET has a 3-tuple of scheduling parameters: 1) priority, 2) deadline, and 3) time slice. Upon each scheduling point, the hardware picks a runnable THREADLET with the highest priority. Among THREADLETS with the same priority, it selects the one with the earliest deadline. Finally, if none has a deadline, it uses round-robin. Each THREADLET executes for its specified time slice unless i) it yields, blocks, or terminates, or ii) a THREADLET with a higher priority or an earlier deadline preempts it. When a THREADLET exhausts its time slice, the deadline is cleared to zero, and the time slice is reset to the specified value.

Second, hardware supports the OS to register a scheduling THREADLET triggered when key scheduling events occur (Figure 3). These scheduling events include THREADLET creation, wakeup, and blocking, etc. The scheduling THREADLET, together with the 3-tuple, enables a wide range of scheduling policies, as we detail next.

Design rationales. The 3-tuple scheduling parameters enable a scheduling fast path: hardware can directly rank THREADLETS and select the next one to run using these simple parameters. The selection and ranking logic is simple to implement in hardware and can complete in tens of cycles, being much faster than the sub-microsecond-scale overhead of software approaches (§2.2).

THREADLET_CREATED
THREADLET_TERMINATED
THREADLET_BLOCKED
THREADLET_YIELD
TIMESLICE_EXHAUSTED
THREADLET_WAKEUP
AFTER_CERTAIN_TIMER_TICK

```
1 if (not flag):
2     # register the monitored address.
3     tlet_monitor(flag)
4
5     if (not flag):
6         # yield with itself blocked.
7         tlet_wait()
```

Figure 3: Left: Events that trigger the scheduling THREADLET. **Right:** Pseudocode for THREADLET's memory notification interface. We note that this interface is similar to x86's MWAIT/MONITOR.

The scheduling THREADLET serves as a slow path that maps the scheduling policy to the 3-tuple. It periodically adjusts the 3-tuple upon key events, thereby supporting advanced policies such as Linux's EEVDF (§5.3), which would be expensive to implement directly in hardware.

With the separation of the fast and slow paths, THREADLET scheduling efficiency depends on how often the slow path is invoked, which in turn depends on how well the policy fits the 3-tuple. Policies that fit poorly require frequent execution of the scheduling THREADLET to adjust the 3-tuple.

Hence, we design the 3-tuple to cover a vast majority of the modern scheduling policies. For example, common policies, such as FIFO, round-robin, and Earliest Deadline First, can be directly expressed by the 3-tuple without invoking the slow path. More advanced policies, such as CFS [99] and EEVDF, can be expressed by the 3-tuple with relatively infrequent slow path invocations. This is because these policies are close to the 3-tuple. For example, EEVDF's virtual deadlines and weighted time slices naturally map to the deadline and time slice in the 3-tuple. As a result, it requires only infrequent invocations of the slow path, as we show in §5.3.

Finally, we note that many real-world policies can be expressed with the 3-tuple. One example is Snap's scheduling policy [62]: during each period, LC threads run for a configurable time slice, while for the remaining time, other threads are scheduled using CFS. Another example is the Secure VM Core Scheduling policy in Tableau [105], which schedules each runnable VM thread for a fixed amount of time in every period. Lastly, the Google search policy supported by GhOSt [35] orders threads by their elapsed runtime and selects the thread with the least elapsed runtime for execution.

3.5 OS Management and Notification

Similar to software threads, THREADLETS are fully manageable by the OS. The underlying hardware provides a set of dedicated instructions for this purpose (Table 3), offering three types of management: 1) lifecycle control (*e.g.*, create or destroy THREADLETS); 2) scheduling operations (*e.g.*, mark a THREADLET as runnable or blocked); 3) registering notification events, as we detail next.

Memory notification. THREADLETS unify thread-to-thread and device-to-thread notification through a single memory notification interface [8, 34]. This interface allows a THREADLET to monitor a memory address, yield the core, enter the `blocked` state, and be woken up when another THREADLET or device writes to the address.

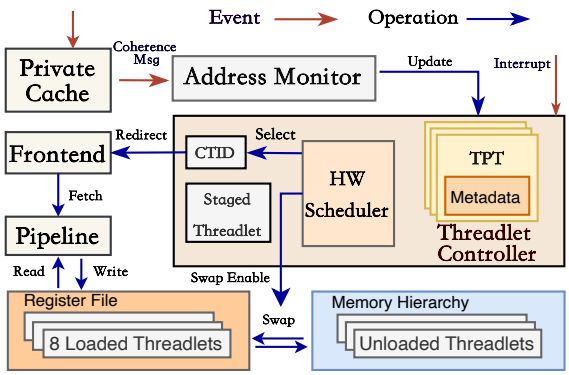


Figure 4: An overview of LOOM.

Figure 3 shows the pseudocode for memory notification. Two instructions are needed: 1) `tlet_monitor` to register the monitored memory address (L3); and 2) `tlet_wait` (L7) to perform the yield and block. Similar to x86's MWAIT/MONITOR, the flag checks (L1 and L5) handle the case where, *e.g.*, the monitored address is updated after `tlet_monitor` but before `tlet_wait`, which would otherwise cause a missed wakeup.

Design rationales. The above interface allows a THREADLET or device to directly notify a THREADLET instead of a core, avoiding overheads of, *e.g.*, kernel data structure updates, APIC programming, and softirq (§2.3). This interface delivers notifications in tens of cycles, matching polling's latency, as both check for memory updates. Yet, unlike polling, the hardware scheduling allows a waiting THREADLET to be preempted by a runnable one, thereby avoiding CPU waste.

Generality. Finally, our experience building an OS (§5) based on THREADLET suggests that this abstraction is general. Our OS uses THREADLETS to subsume many scheduling constructs, including threads, coroutines, softirqs, and workqueues. Its notification interface also replaces conventional mechanisms, like futexes and condition variables.

4 The LOOM Architecture

4.1 LOOM Overview

Figure 4 overviews LOOM, the architecture that supports THREADLETS. The main component is the THREADLET controller, performing THREADLET scheduling, notification, and metadata management. The THREADLET controller interfaces with the local timer to trigger scheduling ticks (§3.4), and with the local cache to detect memory updates on THREADLET-monitored addresses (§3.5). The other key change is an enlarged register file, as detailed next. Section 4.8 describes how LOOM is integrated with an out-of-order and SMT processor.

4.2 Challenge: Supporting Many THREADLETS

A key design challenge in LOOM is to efficiently support comparable numbers of THREADLETS per core to software threads. This allows software to create THREADLETS freely to match

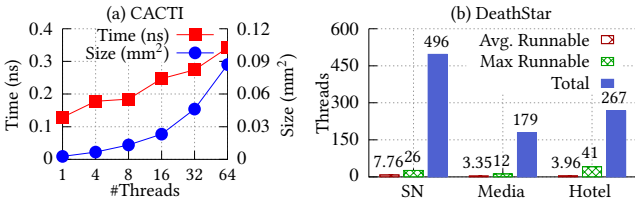


Figure 5: Left: Register file access latency and area vs. THREADLET count, as reported by CACTI [52]. Each THREADLET uses 32 general-purpose registers using 22 nm technology node. **Right:** Average and maximum numbers of runnable and total threads in DeathStarBench, running on 4 cores; we sample every 100 μs.

workload needs, without being constrained by the hardware. However, software threads are abundant in modern workloads, leading to heavy thread oversubscription (*i.e.*, more threads than cores); prior work [53] reports that Google data centers [9] average 50 threads per core, with some reaching around 500. Thus, we design LOOM to support 512 THREADLETS per core.

Yet, scaling to that level introduces two challenges: 1) storing architectural states; and 2) scheduling THREADLETS. For the former challenge, an ideal design would keep every THREADLET's architectural state in the register file. However, even with minimized execution contexts (§3.3), Figure 5 suggests that a register file can efficiently support only a small number (*e.g.*, eight) of THREADLETS. The access latency increase is especially undesirable for μs-scale computation, as it slows down every instruction. This forces LOOM toward a design that departs from most multithreading processors (§7).

The latter challenge arises because a naive THREADLET scheduling design would globally compare all THREADLETS by their 3-tuple scheduling parameters (§3.4). With many THREADLETS, the latency, power consumption, and area of the required logic become prohibitive.

4.3 Critical Path Among Loaded THREADLETS

Observation. We overcome the above challenge with a key observation. In μs-scale workloads with heavy thread oversubscription, only a small number of threads are runnable at any given time. This is because, in such workloads, most threads are either 1) short-lived (*e.g.*, created for a specific task and later terminated) or 2) blocked most of the time, waiting for I/O or for requests or responses from other threads.

We validate the above observation by running `DeathStar`, a microservice benchmark suite [27]. We issue requests using a Poisson distribution with 10K RPS and sample the number of threads every 100 μs. We define "total threads" as the maximum number of threads observed at any sampling point. As shown in Figure 5, the ratio of the average (and maximum) runnable threads to total threads is only 1.5% to 1.9% (and 5.2% to 15.4%, respectively).

Based on the above observation, LOOM's design targets the common case where only a small number of THREADLETS are runnable at a certain time. Specifically, LOOM adopts a tiered storage design. The register file holds the states

```python
def RR_dist(tid_a, tid_b: int):
    return (tid_b - tid_a + n_threadlets) % n_threadlets

# Return the higher-ranked threadlet
def compare_threadlet_rank(a, b):
    # runnable status comparison
    if a.is_runnable != b.is_runnable:
        return a if a.is_runnable else b

    # priority comparison
    if a.priority != b.priority:
        return a if a.priority > b.priority else b

    # deadline comparison
    if a.deadline != b.deadline:
        return a if a.deadline < b.deadline else b

    # round-robin distance
    return a if RR_dist(ctid, a.tid) < RR_dist(ctid, b.tid) else b
```

Figure 6: Pseudocode of the ranking algorithm in LOOM. `CTID`: the ID of the current THREADLET, serving as the round-robin pointer. Note that this algorithm forms a total order among all THREADLETS.

of eight THREADLETS, which we term *loaded* THREADLETS. The other THREADLETS are *unloaded*, and their architectural states reside in the memory hierarchy. Note that an unloaded THREADLET can also be runnable.

Selecting the loaded THREADLETS. The THREADLET controller (§4.1) decides which THREADLETS to load into the register file based on which one will be scheduled first. This design leverages the scheduling semantics carried by THREADLETS (§3.4), enabling the controller to make informed decisions that outperform heuristics such as LRU.

Specifically, given any two THREADLETS, the controller ranks them (*i.e.*, determines which should be scheduled first) based on their runnable state, priority, deadline, and distance from the round-robin pointer (Figure 6).

The controller selects the next THREADLET to run only from the small set of loaded ones, yielding two benefits.

Fast context switching. The first benefit is minimizing context switching latency. As the states of both the current and the next THREADLET are already in the register file, context switching is merely redirecting instruction fetch and decode to the new THREADLET, without any data movement.

Fast scheduling. The second benefit is that the scheduling decision is only made among the loaded THREADLETS, rather than a costly global comparison across all THREADLETS. The controller uses a three-level compare-select tree to perform the required comparison, costing at most two cycles.

If more than eight THREADLETS are runnable, the controller swaps THREADLETS between the register file and the memory hierarchy as their scheduling status changes (*e.g.*, when they block or become runnable). As detailed next, LOOM hides much of this overhead through asynchronous swapping.

4.4 Asynchronous Swapping of THREADLETS

Staged THREADLET. A key optimization in LOOM to hide swapping overhead is a staging buffer that holds the state of one additional THREADLET, termed the *staged* THREADLET.

The staged THREADLET is the highest-ranked THREADLET other than the loaded ones (*i.e.*, the 9th-ranked THREADLET), and is therefore the most likely to be loaded next. With the staged THREADLET, when a THREADLET's rank changes, LOOM only needs to compare it with the staged THREADLET to maintain the set of loaded THREADLETS. In this way, LOOM forgoes the need to scan all THREADLETS.

Maintaining loaded THREADLETS. To ensure that the eight top-ranked THREADLETS are loaded, the controller must handle two key scenarios: 1) an unloaded THREADLET is promoted (*e.g.*, by becoming runnable, being created, increasing its priority, or decreasing its deadline); or 2) one of the loaded THREADLETS is demoted (*e.g.*, by blocking, etc.).

In scenario 1), a promotion emits a rank-change event, which triggers the controller to compare the promoted THREADLET with the staged THREADLET. If the promoted THREADLET has a higher rank, the controller swaps it with the staged THREADLET. This process is performed asynchronously, without blocking the execution or scheduling of the loaded THREADLETS, thereby hiding the swapping latency. Next, the controller compares the new staged THREADLET with all the loaded THREADLETS for a potential swap.

For scenario 2), the controller compares the demoted THREADLET with the staged THREADLET. If its rank is lower, the controller swaps the two THREADLETS. Next, the controller asynchronously compares the new staged THREADLET with the unloaded THREADLETS, which we detail in §4.4.1.

4.4.1 Microarchitecture Design

The controller includes the following specialized storage components: 1) a THREADLET policy table (TPT) that stores each THREADLET's priority (2 bits) and deadline (16 bits, measured in scheduling ticks), totaling 1.2 KB for 512 THREADLETS; 2) a 9-bit current THREADLET ID register (CTID); 3) a 512-bit placement bitmap that tracks which runnable THREADLETS are unloaded.

Deciding a new staged THREADLET from unloaded ones is a two-step process: First, the controller sequentially iterates over unloaded runnable THREADLETS' scheduling states to find the one with the highest rank. Second, the controller compares the winner with the current staged THREADLET.

The first step operates as follows. The controller copies the placement map to a temporary register, and initializes a winner register to track the current winner THREADLET. Next, in each iteration, the controller 1) uses a find-first-set (FFS) circuit to select an unloaded runnable THREADLET from the temporary register, 2) accesses the TPT to retrieve the THREADLET's scheduling state, and 3) compares it with the current winner to update the winner register if necessary.

The controller pipelines each step in the iteration. It runs step 1 for the next THREADLET while performing steps 2 and 3 for the current THREADLET. Step 1 takes two cycles with a hierarchical FFS partitioning the temporary register into 8 64-bit chunks, while steps 2 and 3 each take one cycle. Hence, with, say, 64 unloaded THREADLETS, the latency is only 128 cycles. One could further trade area for lower latency by, *e.g.*, comparing multiple THREADLETS in parallel.

4.5 Supporting Memory Notification

To support the memory notification interface (§3.4), LOOM adds a per-core address monitor table (MAT) that, logically, stores the monitored address for each THREADLET.

LOOM supports the memory notification by snooping cache coherence. The `tlet_monitor` instruction reads the cache line containing the monitored address, brings it into a local cache in the Shared/Exclusive state, and records a) the address; and b) the THREADLET ID in the MAT.

Upon a cache line invalidation, the local cache receives the coherence message from the directory, and forwards the cache line address to the MAT. The MAT matches the invalidated address with its stored address and, upon a match, forwards the corresponding THREADLET ID to the controller. The controller then wakes up the corresponding THREADLET.

MAT realization. There are two ways to realize the MAT. If the size of the local cache is small (*e.g.*, an L1 cache with tens of KBs), LOOM expands the cache tag array to store the 9-bit THREADLET ID for each cache line. If the size of the local cache is large (*e.g.*, an L2 cache with hundreds of KBs), LOOM maintains a separate MAT next to the local cache, and uses the physical address to look up the corresponding THREADLET. A 512-entry MAT is implemented as a set-associative structure to avoid comparing one incoming address against all entries. With a typical 8-way implementation and 48-bit PAs, a 512-entry MAT costs 2.8 KB of on-chip storage.

Spurious wakeups. Similar to `MWAIT`, if the monitored cache line is evicted, the controller also wakes up the waiting THREADLET, causing a spurious wakeup.

We next perform a back-of-the-envelope analysis to estimate its impact. The frequency of spurious wakeups depends on how quickly new data enters the cache. Consider SPEC CPU2017, a representative workload suite reported [73] to incur about 8.1 private-cache (L2) misses per thousand instructions. Assuming a 3 GHz core with an IPC of 1, this corresponds to about 1.6 GB/s of new data being brought into the L2 cache, taking about 0.7 ms to fill a 1 MiB L2 cache. Hence, spurious wakeups occur at a similar frequency, which we consider infrequent for μs-scale THREADLETS.

One could further mitigate spurious wakeups by pinning monitored cache lines (up to a per-set cap) or throttling the execution of THREADLETS whose monitored lines are frequently evicted.

4.6 Privilege and Protection

Privilege and address space. LOOM associates each THREADLET with an address-space identifier (ASID), a page-table base register (PTBR), and a privilege level. LOOM keeps this state on chip for loaded THREADLETS, and for unloaded ones, saves it with the remaining context. Each in-flight instruction carries its THREADLET ID and privilege level. The

MMU uses the above metadata to select the matching address space and check access permissions. Therefore, switching a THREADLET does not require rewriting the related control registers or flushing unrelated in-flight instructions, thereby minimizing overhead.

Defense against Meltdown and Spectre. For Meltdown, LOOM defends by preventing loads or stores from accessing the L1D speculatively before a THREADLET-ID-aware permission check. For Spectre, LOOM provides the same level of security as the baseline core and does not introduce extra vulnerabilities. Existing mitigations, such as MuonTrap, CleanupSpec, and Speculative Privacy Tracking (SPT), can be applied to both LOOM and the baseline core [1, 15, 85].

4.7 Implementation

We built LOOM on a Xilinx FPGA by extending the Rocket core [5], a five-stage, in-order, single-issue RISC-V CPU, with 16 KB L1 instruction and data caches, a 512 KB shared L2 cache, and 16 GB DRAM. We also built a multi-queue NIC based on ICENET [14]. As the MMU in Rocket cores does not support multiple ASIDs, in our prototype, THREADLETS on the same core share the same address space.

Interrupt handling. LOOM provides two interrupt handling paths: a fast path for modern I/O using address monitoring (§4.5), and a slow path for conventional devices. For the slow path, the OS registers a single interrupt-handler THREADLET, which is made runnable whenever an interrupt arrives (see §5.1 for more details). The controller achieves this by monitoring the interrupt signal. To avoid conflicts, I/O devices in the fast path should not register interrupts.

4.7.1 Practical Considerations

Storage cost. The per-core on-chip storage cost of LOOM is around 5.8 KB, including an increase of 1.8 KB in the register file, a 1.2 KB TPT (§4.4.1), and a 2.8 KB MAT (§4.5).

Area cost. The main contributors are the hardware scheduler, the read/write arbitration logic for the eight loaded slots, and the TPT access arbitration logic. Other costs include the scanner logic for maintaining the staged THREADLET, the replacement FSM, and the THREADLET control instructions.

Power and timing overhead. Vivado reports 1) only a 5% increase in total on-chip power and 2) that the critical path is still in Rocket's DTLB and L1D access logic. Because LOOM is not subject to a stringent latency constraint (in terms of clock cycles), one can always perform retiming (i.e., adding pipeline registers) to cut timing-critical paths.

Pipeline complexity. LOOM incurs minimal pipeline complexity, as the extra logic is only required for the following two components: 1) PC generation logic, which must also consider the THREADLET scheduler's output to allow a newly selected THREADLET to begin execution; and 2) the expanded register file to store the context of each loaded THREADLET.

Correctness and verification. LOOM's complexity can be effectively controlled through the standard verification process. LOOM neither introduces complex execution mechanisms (like OoO) nor significantly changes the core microarchitecture. Rather, it only combines existing simple building blocks such as a (single-cycle) arbiter and a finite-state machine. The FPGA prototype further serves as a working proof to show that this complexity is manageable.

4.8 Discussion

Integrating with SMT and OoO processors. We next discuss how to extend LOOM to SMT and OoO processors. With an N-way SMT core, the key change is to let the hardware scheduler select up to N THREADLETS each cycle.

With an OoO core, LOOM must include THREADLET-indexed rename state: committed rename table for every loaded context and speculative rename state for THREADLETS with in-flight instructions. An OoO core requires extra physical registers. Nevertheless, LOOM only needs the extra physical registers for THREADLETS with in-flight instructions, so the register file grows by a limited amount.

Limitations and future work. LOOM currently does not provide hardware support for migrating THREADLETS across cores. Designing generic hardware support for this remains an open question, and we implemented this task in software with the scheduling THREADLET (§5.1).

LOOM does not target indirect costs of context switching, such as cache/TLB misses and branch mispredictions. However, as quantitatively evaluated in prior work [57], cache-related indirect costs do not dominate performance with a locality-aware scheduling policy (*e.g.*, by placing threads of the same process on the same core). Thus, LOOM can apply a locality-aware policy to combat such indirect costs.

5 Use Cases: LoomOS

This section presents LoomOS, an OS that exploits LOOM to address a range of system challenges caused by threading, as summarized in Table 4. We built LoomOS by extending a small Rust OS [6] supporting RISC-V.

We present how LoomOS manages THREADLETS (§5.1), and how it uses LOOM to enhance key OS components: networking (§5.2), THREADLET scheduling (§5.3), concurrency control (§5.4), and system call handling (§5.5). We also discuss promising but unimplemented use cases (§5.6). In total, we developed 9,517 LOC.

5.1 THREADLET Management

Table 3 summarizes THREADLET's management instructions.

Threading model. LoomOS unifies the interrupt-handler THREADLET (§4.7) and scheduling THREADLET (§3.4) into a single per-core event THREADLET. To ensure responsiveness, the event THREADLET runs at the highest priority and always resides in the register file, leaving 7 THREADLETS available for other purposes. The event THREADLET determines the event type by reading a control register updated by LOOM.

THREADLET creation/deletion. As in conventional designs, LoomOS mediates THREADLET creation and deletion.

Instruction	Semantics
`tlet_create`	Create a THREADLET. **Input: program address**.
`tlet_exit`	Delete a THREADLET. **Input: THREADLET ID**.
`tlet_wakeup`	Wake up a blocked THREADLET. **Input: THREADLET ID**.
`tlet_yield`	Yield the core and trigger a rescheduling.
`tlet_(un)monitor`	(Dis)Arm a virtual address.
`tlet_wait`	Use together with `tlet_monitor`.
`tlet_get(set)_reg`	Set (get) a THREADLET's register. **Input: THREADLET ID, Src/Dst Register IDs.**
`tlet_reg_event`	Register a THREADLET to wake up upon certain events. **Input: a THREADLET ID and event ID(s)**.
`tlet_acq(rel)_state`	Acquire (release) the core-shared states.
`tlet_set_sched`	Set a THREADLET's 3-tuple scheduling parameters. **Input: THREADLET ID and parameters**.

Table 3: Threadlet management instructions. The return value is stored in `a0`. Most inputs are encoded in the instruction.

	Ctx Switch	Scheduling	T-T Notifi.	D-T Notifi.
Networking	✓	✓	✓	✓
Blocking Lock	✓	✓	✓	✗
Thread Sched.	✓	✓	✗	✗
System Calls	✓	✗	✓	✗
TLB Shootdown*	✓	✗	✓	✗
Microkernel IPC*	✓	✗	✓	✗

Table 4: A summary of LOOM's use cases. *: not yet implemented.

Upon THREADLET creation (*e.g.*, via `fork()` or kernel background thread creation), LOOMOS first prepares the thread descriptor. It next invokes `tlet_create` passing the program counter as input, and obtains a THREADLET ID. It then initializes its architectural state and scheduling parameters via `tlet_set_reg` and `tlet_set_sched`, respectively. Finally, LOOMOS makes the THREADLET runnable using `tlet_wakeup`. Deletion mostly follows the reverse process.

Integration with the kernel. LOOMOS currently runs traditional OS tasks (*e.g.*, memory management and TCP/IP) using privileged THREADLETS. Each background task is assigned a dedicated THREADLET, running at a low priority. These privileged THREADLETS run on a dedicated set of cores rather than sharing cores with user THREADLETS (§5.5). We leave a more advanced design for future work.

Cross-core migration. Policy-wise, LOOMOS follows conventional policies to assign THREADLETS to cores, considering core load, CPU affinity, and hardware topology (*e.g.*, NUMA). Cross-core migration is also triggered in the rare case where a core runs out of all 512 THREADLETS. Mechanism-wise, LOOMOS migrates a THREADLET by 1) creating a new THREADLET on the target core, 2) copying the architectural and scheduling state to it, and 3) deleting the original one.

Handling non-critical state. A THREADLET context does not include non-critical state (§3.3), such as floating-point registers. Such state is shared among sibling THREADLETS on a core. To use such state, a THREADLET first acquires ownership of the corresponding unit, and then releases it after use. There are two special instructions for this purpose (Table 3); accessing state without acquiring ownership leads to an exception. These two are privileged instructions, enabling LOOMOS to virtualize such state among THREADLETS.

5.2 Microsecond-Scale Networking

Motivation. We assume a conventional networking architecture [39, 41, 53, 57], where a packet dispatcher core 1) receives NIC packets, and 2) forwards packets to worker cores using a load-balancing policy [45]. Also, 3) low-priority, best-effort tasks are co-located on these cores to utilize CPU cycles when the networking workload is light.

Under this architecture, all four inefficiencies emerge. Step 1) involves device-to-thread notification, while step 2) leads to thread-to-thread notification overhead. Step 3) leads to context switching and scheduling overheads. Much prior work proposes exciting designs to address these inefficiencies [39–41, 57, 108], yet they rely on, *e.g.*, coroutines, user threads, and polling, inheriting key drawbacks (§2.2, §2.3).

LOOMOS. LOOMOS comes with an efficient userspace networking runtime: a dedicated core hosts a networking THREADLET, while other cores run multiple worker THREADLETS and best-effort THREADLETS.

The networking THREADLET efficiently receives packets by monitoring NIC queues using the address monitoring mechanism (§3.5). It then processes packets through its protocol stack and dispatches them to worker THREADLETS.

Each worker THREADLET communicates with the networking THREADLET through shared memory, using the address monitoring mechanism. In this way, they yield the core to best-effort THREADLETS when no request is available. Moreover, the worker THREADLETS are assigned a higher priority than best-effort THREADLETS, ensuring responsiveness upon new networking requests. Finally, the worker THREADLET has a 5 μs scheduling time slice. Therefore, it gets preempted upon long request processing, allowing other worker THREADLETS to run, thereby minimizing tail latency [39, 41, 57]. We evaluate this design in §6.3.

5.3 Fast Thread Scheduling with EEVDF

EEVDF background. EEVDF [98] is the default scheduling policy in Linux. With EEVDF, each task is assigned 1) an execution time slice, which is proportional to its weight (*i.e.*, nice value); 2) a virtual runtime, *i.e.*, the CPU time the task has actually received (normalized by its weight); and 3) a virtual deadline, *i.e.*, the expected time when its next time slice completes. EEVDF picks the next task in two steps. First, it identifies eligible tasks, *i.e.*, those whose virtual runtime is less than the current virtual time, meaning that they have not received their expected CPU time. Next, among all eligible tasks, it picks the task with the earliest virtual deadline.

LOOMOS realizes EEVDF as follows. Upon THREADLET creation or deletion, LOOMOS recalculates the execution time slice, virtual runtime, and virtual deadline of each THREADLET to account for the resulting weight changes. The execution time slices, virtual runtimes, and virtual deadlines are all calibrated in units of THREADLET's 32-cycle scheduling ticks (§3.4), making them much finer-grained than Linux's.

LOOMOS uses `tlet_reg_event` to trigger the schedul-

ing THREADLET at the end of the current round, *i.e.*, when all runnable THREADLETS have exhausted their time slices. Once triggered, the scheduling THREADLET advances all THREADLETS' virtual runtimes and updates their virtual deadlines. LOOMOS then identifies all eligible THREADLETS, programs their time slices and virtual deadlines into LOOM as the corresponding fields of the 3-tuple parameters via `tlet_set_sched`, and rearms the scheduling THREADLET. LOOM then runs the THREADLET with the earliest virtual deadline. LOOMOS thus meets all key invariants of EEVDF.

5.4 Efficient Blocking Locks

Motivation. In the manycore era, the efficiency of synchronization primitives, especially locks, is a critical performance factor, leading to much research [19, 29, 31, 44, 48, 82].

Yet, the current threading model introduces a difficult trade-off: if/when should a thread waiting for a lock yield the CPU? The ideal design is to yield immediately on failing to acquire the lock, maximizing CPU sharing, known as a blocking lock. However, due to current threading inefficiencies (Table 4), the status quo is largely opposite, where the waiting thread simply keeps spinning and never yields [29, 44, 48].

LOOMOS design. LOOMOS comes with an efficient blocking lock using LOOM's memory monitoring mechanisms. Instead of spinning, a waiting THREADLET yields the core while monitoring the memory address that indicates when the lock becomes available to it (*e.g.*, the lock variable in a simple spinlock). We implemented and evaluated the classical MCS lock [64] based on the above design (§6.4).

5.5 Delegation-Style System Calls

Motivation. Conventional system call handling incurs two kinds of overhead: 1) kernel trapping, and 2) cache and TLB pollution. FlexSC [93] proposes a model that eliminates both by dedicating a set of cores to handling system calls. Applications run on separate cores and communicate with system call cores via shared memory. Linux's `io_uring` [7] already supports this model at a heavy cost: for efficiency, both system calls and application threads spin for notifications.

LOOMOS reaps FlexSC's performance benefits without wasting CPU. The user and kernel THREADLETS communicate through shared memory, and they use LOOM's memory monitoring mechanism, eliminating spinning.

5.6 Other Use Cases

Efficient TLB shootdowns. Most commodity hardware does not provide TLB coherence, due to complexity. Thus, upon page table changes, a core sends costly IPIs to remote cores for TLB shootdown, which motivates prior work to reduce IPI frequency [3, 47, 60] by, *e.g.*, batching and/or deferring TLB shootdowns. Though clever, these techniques are error-prone and mitigate, rather than eliminate, IPI overhead.

We believe that LOOM's thread-to-thread notification more directly addresses this problem. We envision assigning each core a dedicated TLB-flush THREADLET, which communicates with its peers through the address monitoring mechanism, thereby avoiding IPIs and the associated overhead.

Microkernel IPC. While the modularity of microkernels offers many benefits, such as reliability, security, and fault isolation, IPC overhead prevents their wide deployment. Software solutions are either limited to a specific scenario (*e.g.*, thread migration [23, 76]), or sacrifice modularity (*e.g.*, placing multiple services in the same address space) [10, 77].

LOOMOS. We expect that LOOM can eliminate much of the IPC overhead by adopting the approach in §5.5.

6 Evaluation

Our evaluation answers the following questions:

- How efficient is THREADLET context switching (§6.2)?
- How fast and flexible is THREADLET scheduling (§6.2)?
- Does LOOM reduce the overhead of thread-to-thread and device-to-thread notifications (§6.2)?
- How does LOOMOS's networking perform (§6.3)?
- Does LOOMOS achieve lock efficiency (§6.4)?
- Does LOOMOS improve syscall performance (§6.5)?

6.1 Evaluation Setup

Our evaluation uses FireSim [43], a cycle-accurate, FPGA-accelerated simulator running on a Xilinx VCU118 board [2]. The simulated target is a four-core Rocket processor described in Section 4.7, with an x86 host running a C++ switch that connects client traffic to the target. The LOOMOS experiments use the Loom-enabled Rocket core, whereas the Linux baseline runs Linux 6.6.0.

Limitations. Our evaluation is limited by the baseline Rocket core, a simple in-order core that cannot capture advanced hardware effects, such as out-of-order execution, hyperthreading, and performance constraints imposed by power and thermal budgets.

6.2 Microbenchmark

Workload. Each run involves creating 8 thread(let)s on one core. Each thread(let) spins for 10 μs and then yields. For context switching, we report the cycles spent switching architectural contexts after a yield.

Context switching. Figure 7(a) reports the context switching latency for LOOMOS and Linux across 8 threads on a core. LOOMOS takes 17 cycles, compared with 728 cycles for Linux. This reduction is because LOOM's register file stores THREADLET's context (§4.3).

Scheduling. Figure 7(b) reports scheduling overhead using EEVDF. With LOOMOS, it only takes 20 cycles to select the next THREADLET, while the policy maintenance in the scheduling THREADLET (§5.3) amortizes to 72 cycles per selection with 8 THREADLETS. This leads to a total of 92 cycles, which is 25× lower than Linux's 2,315 cycles.

Figure 8 reports the throughput and scheduling overhead as the number of thread(let)s varies. At 8 THREADLETS, LOOMOS delivers throughput of 99.2 KTPS, which is 1.34×

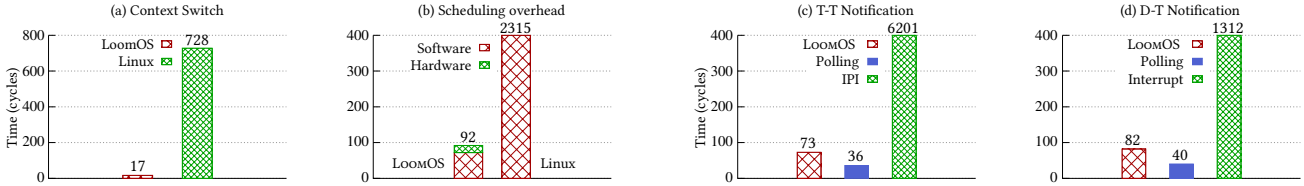


Figure 7: Context switching, scheduling, and notification overhead. Lower is better.

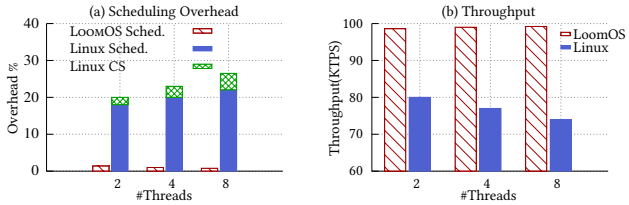


Figure 8: Results of multithreaded scheduling. **Left:** Overhead, lower is better; **Right:** Throughput, higher is better.

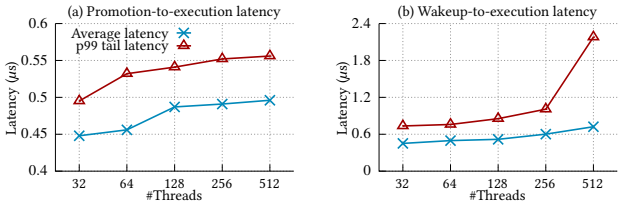


Figure 9: Wakeup latency as the number of THREADLETS increases. Lower is better.

that of Linux, while reducing scheduling overhead from 22% to 0.8%. For LoomOS, increasing the number of THREADLETS does not affect hardware scheduling overhead, whereas Linux's overhead increases significantly.

Notification. The thread-to-thread experiment places the receiver on one core and the sender on another. The receiver monitors an address and sleeps; the sender wakes it up by writing to that address. The device-to-thread experiment monitors a NIC descriptor's completion bit, which the NIC updates after its DMA write. We measure from the sender's store instruction or the NIC's completion update until the receiver resumes execution.

Figure 7(c–d) compares Loom with polling, IPI, and interrupt. Loom achieves comparable performance to polling, thanks to the direct coherence notification (§4.5). Compared with IPIs (and interrupts), Loom reduces thread-to-thread (and device-to-thread) notification latency by up to 84.9× (and 16.0×, respectively).

Wakeup latency. A THREADLET may experience delay after wakeup or promotion while, e.g., waiting for the controller to identify the staged THREADLET or for its context to be loaded from main memory.

We quantify this delay with two experiments. First, we measure promotion-to-execution latency with 32–512 runnable THREADLETS on one core. In each of 1,000 trials, we randomly promote one THREADLET to the highest rank and measure the time until it begins execution. The selected THREADLET runs an idle loop for 0.5 μs.

Figure 9(a) reports the results. With 512 THREADLETS, the average and p99 latencies are 0.50 μs and 0.56 μs, respectively.

We next measure wakeup-to-execution latency with 32–512 THREADLETS on one core. We drive this experiment by capturing and replaying the DeathStarBench [27] trace. Specifically, in the capture phase, upon a thread sleeping and waking up, we record the timestamps to compute the interval. With these intervals, we compute a distribution by calculating the average and standard deviation. During replay, each THREADLET runs an idle loop for 0.5 μs and then sleeps. Upon the sleep, we use the computed distribution to decide the time for waking up the THREADLET. We note that multiple THREADLETS may wake up simultaneously in this experiment, and thus, the latency may be higher than in the promotion-to-execution experiment, which selects only one THREADLET to promote.

Figure 9(b) reports the results. With 512 THREADLETS, the average and p99 latencies are 0.7 μs and 2.1 μs, respectively.

6.3 μs-Scale Networking

To demonstrate the efficiency of LoomOS with hardware THREADLET abstraction, we evaluate our design using several workloads across use cases 1) network, 2) lock, and 3) syscall.

Baseline. We compare LoomOS with NanoPU [37], Shinjuku [41], Shenango [75], and the Linux kernel stack using polling. NanoPU delivers packets directly to registers and uses hardware thread scheduling. Shinjuku and Shenango are high-performance software-based userspace networking runtimes. For Linux, we configure NAPI to poll for packets.

We do not include recent works [33, 40, 53, 54] that use userspace interrupts, which the Rocket core does not support. However, we note that userspace interrupts still incur sub-μs latency (§2.4), which is much higher than LoomOS's notification latency shown in Figure 7(c–d).

Configuration. We use four target cores. For `Shinjuku`, `Shenango`, and LoomOS, one core runs the networking and three cores run the application. For NanoPU, all four cores run applications, as NanoPU handles all networking in hardware. In LoomOS, each application core runs 8 worker THREADLETS with a dedicated request queue, while the Linux version, `Shenango`, and `Shinjuku` each dedicate one worker thread to every worker core. The Linux and `Shenango` workers finish each assigned request before taking another, while `Shinjuku` centrally schedules preemptible request contexts onto these workers.

Workloads. We use synthetic server workloads and an in-

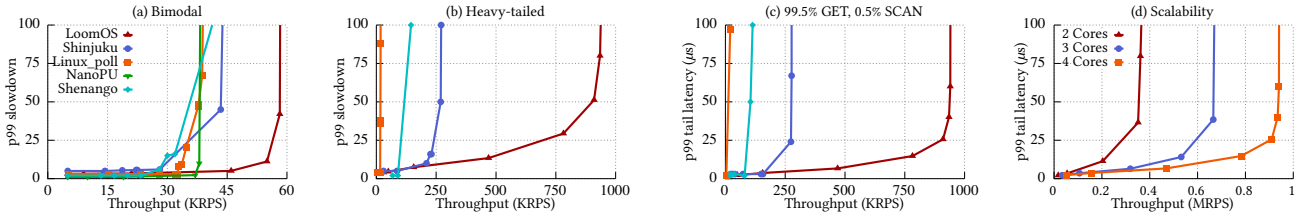


Figure 10: Networking performance. Subfigures (a–c) compare LoomOS with networking baselines in different benchmarks. Subfigure (d) shows the networking scalability of LoomOS. NanoPU is omitted from workloads on which it loses most packets.

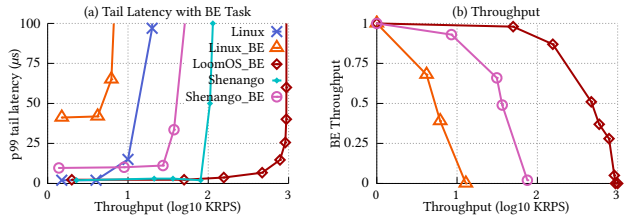


Figure 11: The performance of task co-location. (a) The tail latency and throughput of the LC task and (b) BE-task throughput versus request throughput. Note that the X-axis is in log scale.

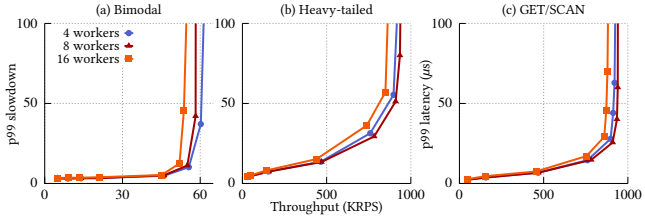


Figure 12: Sensitivity of networking performance with 4, 8, and 16 worker THREADLETS per worker core.

memory key-value store, which is a simplified version of LMDB [16]. The synthetic server spins for each request's service time, following [39, 57]. `Bimodal` uses 50% 1 μs requests and 50% 100 μs requests. `Heavy-tailed` uses 99.5% 1 μs requests and 0.5% 100 μs requests. The LMDB workload contains 99.5% GET operations and 0.5% SCAN operations; a SCAN takes approximately 1,000× as long as a GET.

Figures 10(a–b) show the results for synthetic workloads. On Bimodal, LoomOS reaches 1.3× `Shinjuku`'s, 1.5× Linux polling's, and 1.3× `Shenango`'s maximum throughput. On Heavy-tailed, the improvements are 3.5×, 53.1×, and 6.5×, respectively. For LMDB, Figure 10(c) shows that the respective throughput gains are 3.4×, 47.0×, and 8.25×. These gains are from all four aspects of improvements shown in Figure 7.

Sensitivity analysis and scalability. Figure 12 reports results with 4, 8, and 16 worker THREADLETS per core. Maximum throughput changes less than 10% across these configurations, showing that Loom's results are relatively stable.

Figure 10(d) scales the number of LMDB workload cores from 1 to 3 (total cores: 2 to 4). Relative to one worker core, two and three worker cores improve throughput by approximately 1.9× and 2.7×, respectively.

CPU resource sharing. We next colocate latency-critical (LC) networking tasks with best-effort (BE) tasks, where we add one BE thread(let) per core that performs idle looping.

Figure 11(a) shows that the tail latency of LC tasks increases significantly under both Linux and Shenango upon colocation. Figure 11(b) reports BE throughput normalized to running the task alone. With a 10 μs p99 latency, LoomOS achieves 600 KRPS while retaining 37% BE throughput.

6.4 Efficient Blocking Locks

Baseline. For locking, we compare LoomOS with 1) the `mutex` lock in the Linux kernel, 2) `MCS-B`, a blocking variant of the `MCS` lock [64], and 3) a blocking lock implemented in `Argobots` [87], a user-space threading runtime. All threads contend for one lock with a 10 μs critical section. We use four cores and distribute the thread(let)s evenly.

Figure 13 reports throughput (*i.e.*, completed critical sections per second) and average latency (*i.e.*, the average interval from one unlock to the next successful acquisition). With 32 thread(let)s, LoomOS reduces latency by 28.5×, 17.1×, and 26.5× over Linux `mutex`, `Argobots`, and `MCS-B`, respectively, while delivering 2.17×, 1.67×, and 1.93× their throughput. Moreover, as the number of THREADLETS increases from 4 to 32, LoomOS's average latency increases by 28.6%, while throughput changes by less than 1%.

Figure 14(a) breaks down the latency for the 4-core, 4-thread configuration. The LoomOS blocking lock completes a hand-off in 277 cycles, with only 40.1% of the latency spent on operations other than the unlock itself, owing to efficient notification, fast context switching, and hardware scheduling. In comparison, the other baselines incur substantially higher hand-off overheads, and their fractions of latency spent outside the unlock operation are 72.1% to 94.2%.

6.5 Delegation-Style System Calls

Figure 14(b) shows the improvement of LoomOS in system call performance. We measure the average latency of a system call, from issuance in user space to the start of kernel processing, as well as from the end of kernel processing to when the result is received in user space. LoomOS reduces overhead by 46.2% compared with the traditional system call in Linux, while only slightly increasing the overhead compared with the polling version in Linux, due to its efficient thread-to-thread notification mechanisms. In these two phases, the primary overhead in LoomOS comes from writing the parameters or results into shared memory.

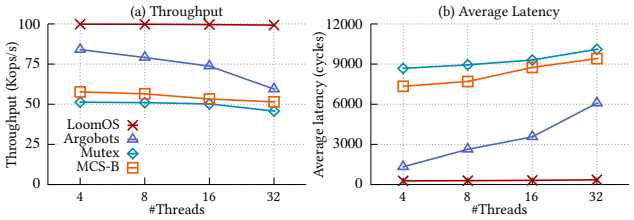


Figure 13: Lock performance with one shared lock and a 10 μs critical section. (a) Throughput; higher is better. (b) Latency; lower is better.

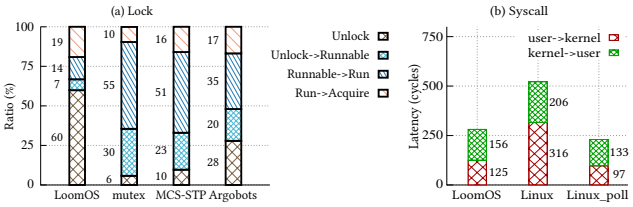


Figure 14: Left: Breakdown of lock hand-off latency. We separate lock release, waiter wakeup, scheduling, and lock acquisition. **Right:** Breakdown of system-call overhead. Lower is better.

7 Related Work

Our work is heavily inspired by the seminal vision paper from Barroso *et al.* [8], which proposes: 1) supporting many hardware threads per core with fast context switching; and 2) an I/O thread that yields the CPU and waits for device notifications using memory-monitoring interfaces. Humphries *et al.* [34] propose more detailed proposals, leading to many new interesting designs. The most relevant ones include: 1) expanding the register file and/or leveraging caches for storing hardware threads' architectural state; and 2) these two ideas enable various use cases, including a new I/O model that replaces polling/interrupts, and microkernel IPC.

Compared with these prior high-level proposals, our contribution is 1) to answer two key research questions necessary for turning these proposals into practical systems; and 2) to provide a concrete design with detailed analysis and evaluation. First, these two works do not address the central challenge of supporting many threads per core. In contrast, our analysis reveals a key insight: the number of runnable threads is often small (§4.3). This insight enables Loom to support 512 threadlets per core with a moderately sized register file and a staging threadlet (§4), rather than requiring a huge register file. Second, while these two works (and others; see §2.4) propose hardware scheduling, it remains unclear how hardware should preserve flexible OS scheduling policies. Loom addresses this challenge through a combination of 1) the 3-tuple scheduling parameters (§3.4), which serve as a hardware fast path (§4.3), and 2) the scheduling threadlet, offering a software slow path. Finally, we propose and evaluate new use cases (§5), such as concurrency control and a complete network runtime.

Hardware threading. We view threadlets (and Loom) as fundamentally different from existing hardware threading techniques (and multithreaded processors) for the following reasons. First, they target different problems. threadlets solve system problems (§5) by removing software threads from the critical path. In contrast, hardware threading reaps microarchitectural performance benefits by maximizing pipeline utilization by, *e.g.*, hiding memory stalls. Second, they are different abstractions. threadlets come with OS scheduling semantics (§3.4) and manageability (*e.g.*, threadlet creation/deletion; see Table 3) that hardware threads lack. Indeed, threadlets and hardware threads are complementary, and they can coexist as explained in §4.8. Finally, they are different in architectural design. In particular, we do not expand register files to support many threadlets, while most hardware threading techniques do [92, 102, 103].

Scheduler activations [4] share with threadlet the goal of bridging the abstraction mismatch between units of execution and computing resources. The key difference is that Loom downloads the thread abstraction to hardware, while scheduler activations expose the virtual processor abstraction to user-level software. Therefore, scheduler activations remain a software-only solution and cannot achieve the high performance of Loom, which leverages hardware support. In addition, as its name suggests, scheduler activations only address the scheduling problem, while Loom additionally addresses 1) context switching, and 2) notification (§2.1).

8 Conclusion

This paper presents threadlet, a new hardware abstraction, and Loom, its co-designed architecture, which together address the four long-standing inefficiencies in the current threading model. Our key insight is that the basic units of execution (namely threads) should be supported directly by hardware rather than by software as an extra level of indirection. As one way to realize this insight, we propose the threadlet abstraction with three key properties: 1) a minimized, deliberately chosen execution context, 2) simple yet expressive scheduling primitives, and 3) full OS manageability via dedicated instructions. Moreover, Loom's design effectively scales to hundreds of threadlets by exploiting the observation that, at any given time, only a small number of threadlets are runnable. With that, Loom delivers a practical hardware design, by only slightly expanding the register file, while effectively hiding the memory access latency. Evaluation shows that, across various domains, Loom improves performance by 3× to 47× on real-world applications.

Acknowledgments

We thank our shepherd, Aurojit Panda, and the anonymous reviewers for their insightful feedback. We also thank the members of TELOS Systems Lab for their valuable support throughout this work. This work was supported by the National Science Foundation of China (No.62372011, 92582118, 62032001). Diyu Zhou is the corresponding author.

References

[1] Sam Ainsworth and Timothy M. Jones. MuonTrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *Proceedings of the 47th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 132–144, Virtual, September 2020.

[2] AMD. *VCU118 Evaluation Board User Guide*, 2023.

[3] Nadav Amit, Amy Tai, and Michael Wei. Don't shoot down TLB shootdowns! In *Proceedings of the Fifteenth European Conference on Computer Systems*, Heraklion, Greece, April 2020.

[4] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[5] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Palmer Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Benjamin Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The Rocket Chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, 2016.

[6] Asterinas Team. Asterinas. https://github.com/asterinas/asterinas. GitHub repository, accessed 2026-04-02.

[7] Jens Axboe. Efficient IO with io_uring. https://www.kernel.dk/io_uring.pdf, October 2019.

[8] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Commun. ACM*, 2017.

[9] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *Commun. ACM*, April 2016.

[10] 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. Microkernel Goes General: Performance and Compatibility in the HongMeng Production Microkernel. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Santa Clara, CA, July 2024.

[11] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Tom Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. Remote procedure call as a managed system service. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, April 2023.

[12] Liao Chen, Shutian Luo, Chenyu Lin, Zizhao Mo, Huanle Xu, Kejiang Ye, and Chengzhong Xu. Derm: SLA-aware Resource Management for Highly Dynamic Microservices. In *Proceedings of the 51st ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Buenos Aires, Argentina, June 2024.

[13] Youmin Chen, Jiwu Shu, Yanyan Shen, Linpeng Huang, and Hong Mei. Rearchitecting the Thread Model of In-Memory Key-Value Stores with μTPS. In *Proceedings of the 31st ACM Symposium on Operating Systems Principles (SOSP)*, Seoul, South Korea, October 2025.

[14] Chipyard Project. IceNet. https://chipyard.readthedocs.io/en/stable/Generators/IceNet.html. Chipyard documentation (IceNet/IceNIC), accessed 2026-04-01.

[15] Rutvik Choudhary, Jiyong Yu, Christopher W. Fletcher, and Adam Morrison. Speculative privacy tracking (SPT): Leaking information from speculative execution without compromising privacy. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 607–622, 2021.

[16] Howard Chu. MDB: A memory-mapped database and backend for OpenLDAP. In *Proceedings of the 3rd International Conference on LDAP (LDAPCon)*, 2011.

[17] Fernando J. Corbató, Marjorie Merwin-Daggett, and Robert C. Daley. An Experimental Time-Sharing System. In *AFIPS Spring Joint Computer Conference 1962*, 1962.

[18] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. RPCValet: NI-Driven Tail-Aware Balancing of μs-Scale RPCs. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Providence, RI, April 2019.

[19] Dave Dice. Malthusian Locks. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Belgrade, RS, April 2017.

[20] Edsger W. Dijkstra. The structure of "THE"-multiprogramming system. *Communications of the ACM*, 1968.

[21] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. XPC: Architectural Support for Secure and Efficient Cross Process Call. In *Proceedings of the 46th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Phoenix, AZ, June 2019.

[22] Tamer Eldeeb, Xincheng Xie, Philip A. Bernstein, Asaf Cidon, and Junfeng Yang. Chardonnay: Fast and General Datacenter Transactions for On-Disk Databases. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, July 2023.

[23] Kevin Elphinstone and Gernot Heiser. From L3 to seL4: What Have We Learnt in 20 Years of L4 Microkernels? In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, November 2013.

[24] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Inigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making Kernel Bypass Practical for the Cloud with Junction. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Santa Clara, CA, April 2024.

[25] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, October 2020.

[26] Vasile Gheorghita Gaitan, Nicoleta Cristina Gaitan, and Ioan Ungurean. CPU architecture based on a hardware scheduler and independent pipeline registers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(9):1661–1674, September 2015.

[27] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Providence, RI, April 2019.

[28] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing Performance and Isolation in Microkernels with Efficient Intra-kernel Isolation and Communication. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, Virtual, July 2020.

[29] Rachid Guerraoui, Hugo Guiroux, Renaud Lachaize, Vivien Quéma, and Vasileios Trigonakis. Lock-unlock: Is that all? a pragmatic analysis of locking in software systems. *ACM Transactions on Computer Systems*, 2018.

[30] Linsong Guo, Danial Zuberi, Tal Garfinkel, and Amy Ousterhout. The Benefits and Limitations of User Interrupts for Preemptive Userspace Scheduling. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Philadelphia, PA, April 2025.

[31] Vishal Gupta, Kumar Kartikeya Dwivedi, Yugesh Kothari, Yueyang Pan, Diyu Zhou, and Sanidhya Kashyap. Ship your critical section, not your data: Enabling transparent delegation with TCLOCKS. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, July 2023.

[32] Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Virtual, April 2021.

[33] Kang Hu, Shuqi Dong, Chuandong Li, Ran Yi, Zonghao Zhang, Yiming Yao, Bo An, Jie Zhang, Xiaolin Wang, Yingwei Luo, Zhenlin Wang, and Diyu Zhou. SBB: Eliminating centralized bottlenecks in userspace network runtime. In *Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 473–489, Seattle, WA, July 2026.

[34] Jack Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. A Case Against (most) Context Switches. In *18th USENIX Workshop on Hot Topics in Operating Systems (HotOS) (HotOS XVIII)*, Virtual, May 2021.

[35] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Koblenz, Germany, October 2021.

[36] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting Linux Storage Stack for µs Latency and High Throughput. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, October 2021.

[37] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A Nanosecond Network Stack for Datacenters. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, October 2021.

[38] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B, 3C, & 3D): System Programming Guide*, March 2025. See Chapter 9 "Task Management," Section 10.10.4 "MONITOR/MWAIT Instruction," and Section 10.10.5 "Monitor/Mwait Address Range Determination".

[39] Rishabh R. Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, Koblenz, Germany, October 2023.

[40] Yuekai Jia, Kaifu Tian, Yuyang You, Yu Chen, and Kang Chen. Skyloft: A General High-Efficient Scheduling Framework in User Space. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP)*, Austin, TX, October 2024.

[41] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for µsecond-scale Tail Latency. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, February 2019.

[42] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Koblenz, Germany, October 2021.

[43] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy H. Katz, Jonathan Bachrach, and Krste Asanovic. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *Proceedings of the 45th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Los Angeles, CA, June 2018.

[44] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scalable NUMA-aware blocking synchronization primitives. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[45] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[46] Paul Kohout, Brinda Ganesh, and Bruce L. Jacob. Hardware support for real-time operating systems. In *Proceedings of the First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 45–51, 2003.

[47] Mohan Kumar Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselý, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. LATR: Lazy translation coherence. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Williamsburg, VA, March 2018.

[48] Victor Laforet, Sanidhya Kashyap, Calin Iorgulescu, Julia Lawall, and Jean-Pierre Lozi. FlexGuard: Fast mutual exclusion independent of subscription. In *Proceedings of the 31st ACM Symposium on Operating Systems Principles (SOSP)*, Seoul, South Korea, October 2025.

[49] Butler W. Lampson. A scheduling philosophy for multiprocessing systems. *Communications of the ACM*, 1968.

[50] Chuandong Li, Ran Yi, Zonghao Zhang, Jing Liu, Changwoo Min, Jie Zhang, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, and Diyu Zhou. Aeolia: A Fast and Secure Userspace Interrupt-Based Storage Stack. In *Proceedings of the 31st ACM Symposium on Operating Systems Principles (SOSP)*, Seoul, South Korea, October 2025.

[51] Quanxi Li, Hong Huang, Ying Liu, Yanwen Xia, Jie Zhang, Mosong Zhou, Xiaobing Feng, Huimin Cui, Quan Chen, Yizhou Shan, and Chenxi Wang. Beehive: A Scalable Disaggregated Memory Runtime Exploiting Asynchrony of Multithreaded Programs. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Philadelphia, PA, April 2025.

[52] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2011.

[53] Yueying Li, Nikita Lazarev, David Koufaty, Tenny Yin, Andy Anderson, Zhiru Zhang, G. Edward Suh, Kostis Kaffes, and Christina Delimitrou. LibPreemptible: Enabling Fast, Adaptive, and Hardware-Assisted User-Space Scheduling. In *Proceedings of the 30th IEEE Symposium on High Performance Computer Architecture (HPCA)*, Edinburgh, United Kingdom, March 2024.

[54] Jiazhen Lin, Youmin Chen, Shiwei Gao, and Youyou Lu. Fast core scheduling with userspace process abstraction. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP)*, pages 280–295, Austin, TX, October 2024.

[55] Bowen Liu, Xinyang Huang, Qijing Li, Zhuobin Huang, Yijun Sun, Wenxue Li, Junxue Zhang, Ping Yin, and Kai Chen. CEIO: A Cache-Efficient Network I/O Architecture for NIC-CPU Data Paths. In *Proceedings of the 2025 ACM SIGCOMM*, New York, NY, September 2025.

[56] David Lo, Liqun Cheng, Rama K. Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Portland, OR, June 2015.

[57] Zhihong Luo, Sam Son, Dev Bali, Emmanuel Amaro, Amy Ousterhout, Sylvia Ratnasamy, and Scott Shenker. Efficient Microsecond-Scale Blind Scheduling with Tiny Quanta. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Diego, CA, April 2024.

[58] Zhihong Luo, Sam Son, Sylvia Ratnasamy, and Scott Shenker. Harvesting Memory-bound CPU Stall Cycles in Software with MSH. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Santa Clara, CA, July 2024.

[59] Haoran Ma, Yifan Qiao, Shi Liu, Shan Yu, Yuanjiang Ni, Qingda Lu, Jiesheng Wu, Yiying Zhang, Miryung Kim, and Harry Xu. DRust: Language-Guided Distributed Shared Memory with Fine Granularity, Full Transparency, and Ultra Efficiency. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Santa Clara, CA, July 2024.

[60] Steffen Maass, Mohan Kumar Kumar, Taesoo Kim, Tushar Krishna, and Abhishek Bhattacharjee. ecoTLB: Eventually consistent TLBs. *ACM Transactions on Architecture and Code Optimization*, 2020.

[61] Reinhard Männer. Hardware task/processor scheduling in a polyprocessor environment. *IEEE Transactions on Computers*, C-33(7):626–636, July 1984.

[62] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 399–413, Ontario, Canada, October 2019.

[63] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. Efficient Scheduling Policies for Microsecond-Scale Tasks. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Renton, WA, April 2022.

[64] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 1991.

[65] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys)*, Dresden, Germany, March 2019.

[66] Samantha Miller, Anirudh Kumar, Tanay Vakharia, Ang Chen, Danyang Zhuo, and Thomas E. Anderson. Enoki: High Velocity Linux Kernel Scheduler Development. In *Proceedings of the 19th European Conference on Computer Systems (EuroSys)*, Athens, Greece, April 2024.

[67] Amirhossein Mirhosseini, Hossein Golestani, and Thomas F. Wenisch. HyperPlane: A Scalable Low-Latency Notification Accelerator for Software Data Planes. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Virtual, October 2020.

[68] Amirhossein Mirhosseini, Akshitha Sriraman, and Thomas F. Wenisch. Enhancing Server Efficiency in the Face of Killer Microseconds. In *Proceedings of the 25th IEEE Symposium on High Performance Computer Architecture (HPCA)*, Washington, DC, February 2019.

[69] Amirhossein Mirhosseini, Brendan L. West, Geoffrey W. Blake, and Thomas F. Wenisch. Express-Lane Scheduling and Multithreading to Minimize the Tail Latency of Microservices. In *Proceedings of the 2019 IEEE International Conference on Autonomic Computing (ICAC)*, 2019.

[70] Amirhossein Mirhosseini, Brendan L. West, Geoffrey W. Blake, and Thomas F. Wenisch. Q-Zilla: A Scheduling Framework and Core Microarchitecture for Tail-Tolerant Microservices. In *Proceedings of the 26th IEEE Symposium on High Performance Computer Architecture (HPCA)*, San Diego, CA, February 2020.

[71] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Santa Clara, CA, February 2020.

[72] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. LXDs: Towards Isolation of Kernel Subsystems. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[73] Alberto Navarro-Torres, Jesús Alastruey-Benedé, Pablo Ibáñez-Marín, and Víctor Viñals-Yúfera. Memory hierarchy characterization of SPEC CPU2006 and SPEC CPU2017 on the Intel Xeon Skylake-SP. *PLOS ONE*, 14(8):e0220135, 2019.

[74] Lars Nolte, Tim Twardzik, Camille Jalier, Zhigang Huang, Jiyuan Shi, Clara Kowalsky, Thomas Wild, and Andreas Herkersdorf. HAWEN: Hardware Accelerator for Thread Wake-Ups in Linux Event Notification. In *2023 60th ACM/IEEE Design Automation Conference (DAC 2023)*, 2023.

[75] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, February 2019.

[76] Gabriel Parmer. The case for thread migration: Predictable IPC in a customizable and reliable OS. In *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2010)*, 2010.

[77] Yuke Peng, Hongliang Tian, Junyang Zhang, Ruihan Li, Chengjun Chen, Jianfeng Jiang, Jinyi Xian, Xiaolin Wang, Chenren Xu, Diyu Zhou, Yingwei Luo, Shoumeng Yan, and Yinqian Zhang. ASTERINAS: A Linux ABI-Compatible, Rust-Based Framekernel OS with a Small and Sound TCB. In *Proceedings of the 2025 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2025.

[78] Boris Pismenny, Adam Morrison, and Dan Tsafrir. ShRing: Networking with Shared Receive Rings. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, July 2023.

[79] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, October 2017.

[80] Shujian Qian and Ashvin Goel. Massively Parallel Multi-Versioned Transaction Processing. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Santa Clara, CA, July 2024.

[81] Deepti Raghavan, Shreya Ravi, Gina Yuan, Pratiksha Thaker, Sanjari Srivastava, Micah Murray, Pedro Henrique Penna, Amy Ousterhout, Philip Levis, Matei Zaharia, and Irene Zhang. Cornflakes: Zero-Copy Serialization for Microsecond-Scale Networking. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, Koblenz, Germany, October 2023.

[82] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. ffwd: delegation is (much) faster than you think. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, October 2017.

[83] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated Far Memory. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, October 2020.

[84] Sergio Sáez, Joan Vila, Alfons Crespo, and Angel García. A hardware scheduler for complex real-time systems. In *Proceedings of the IEEE International Symposium on Industrial Electronics*, volume 1, pages 43–48, 1999.

[85] Gururaj Saileshwar and Moinuddin K. Qureshi. CleanupSpec: An "undo" approach to safe speculation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 73–86, 2019.

[86] Dongjoo Seo, Jihyeon Jung, Yeohwan Yoon, Ping-Xiang Chen, Yongsoo Joo, Sung-Soo Lim, and Nikil Dutt. DPAS: A Prompt, Accurate and Safe I/O Completion Method for SSDs. In *24th USENIX Conference on File and Storage Technologies (FAST) (FAST 26)*, Santa Clara, CA, February 2026.

[87] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrian Castello, Damien Genet, Thomas Herault, Shintaro Iwasaki, Prateek Jindal, Laxmikant V. Kale, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, Kenjiro Taura, and Pete Beckman. Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):512–526, 2018.

[88] Hamed Seyedroudbari, Srikar Vanavasam, and Alexandros Daglis. Turbo: SmartNIC-enabled Dynamic Load Balancing of μs-scale RPCs. In *Proceedings of the 29th IEEE Symposium on High Performance Computer Architecture (HPCA)*, Montreal, QC, March 2023.

[89] Balaram Sinharoy, James Van Norstrand, Richard J. Eickemeyer, Hung Q. Le, Jens Leenstra, Dung Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, José E. Moreira, D. Levitan, S. Tung, David Hrusecky, James W. Bishop, Michael Gschwind, Maarten Boersma, Michael Kroener, Markus Kaltenbach, Tejas Karkhanis, and K. M. Fernsler. IBM POWER8 Processor Core Microarchitecture. *IBM Journal of Research and Development*, 59(1):2:1–2:21, 2015.

[90] Athinagoras Skiadopoulos, Zhiqiang Xie, Mark Zhao, Qizhe Cai, Saksham Agarwal, Jacob Adelmann, David Ahern, Carlo Contavalli, Michael D. Goldflam, Vitaly Mayatskikh, Raghu Raja, Daniel Walton, Rachit Agarwal, Shrijeet Mukherjee, and Christos Kozyrakis. High-throughput and Flexible Host Networking for Accelerated Computing. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Santa Clara, CA, July 2024.

[91] Burton J. Smith. A Pipelined, Shared Resource MIMD Computer. In *Proceedings of the 1978 International Conference on Parallel Processing*, 1978.

[92] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Real-Time Signal Processing IV*, 1982.

[93] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, October 2010.

[94] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU register state using microarchitectural side-channels. https://arxiv.org/abs/1806.07480, 2018.

[95] Jovan Stojkovic, Chunao Liu, Muhammad Shahbaz, and Josep Torrellas. μManycore: A Cloud-Native CPU for Tail at Scale. In *Proceedings of the 50th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Orlando, FL, September 2023.

[96] Amy Tai, Igor Smolyar, Michael Wei, and Dan Tsafrir. Optimizing Storage Performance with Calibrated Interrupts. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, October 2021.

[97] The DPDK Project. DPDK: Data plane development kit. https://www.dpdk.org/. Project website, accessed 2026-04-01.

[98] The kernel development community. EEVDF scheduler. https://docs.kernel.org/scheduler/sched-eevdf.html, 2024. Linux kernel documentation, accessed 2026-04-01.

[99] The Linux kernel development community. CFS scheduler. https://docs.kernel.org/scheduler/sched-design-CFS.html. Linux kernel documentation, accessed 2026-07-23.

[100] The Linux kernel development community. NAPI. https://docs.kernel.org/networking/napi.html. Linux kernel documentation, accessed 2026-08-27.

[101] The SPDK Project. SPDK: Storage performance development kit. https://spdk.io/. Project website, accessed 2026-04-01.

[102] Radhika Thekkath and Susan J. Eggers. The effectiveness of multiple hardware contexts. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, 1994.

[103] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd ACM/IEEE International Symposium on Computer Architecture (ISCA)*, S. Margherita Ligure, Italy, June 1995.

[104] E. Tune, R. Kumar, D.M. Tullsen, and B. Calder. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Portland, Oregon, December 2004.

[105] Manohar Vanga, Arpan Gujarati, and Björn B. Brandenburg. Tableau: A high-throughput and predictable VM scheduler for high-density workloads. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*, pages 1–16, Porto, Portugal, April 2018.

[106] Yun Wang, Xingguo Jia, Ben Luo, Kenan Liu, Shengdong Dai, Jingdong Han, Weihao Chen, Yicheng Gu, Xingzi Yu, Yibin Shen, Jiesheng Wu, Zhengwei Qi, and Haibing Guan. What are you (M)waiting for: The hidden cost of idle in the hyperscale cloud. In *Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 529–545, Seattle, WA, July 2026.

[107] Yifan Yuan, Jinghan Huang, Yan Sun, Tianchen Wang, Jacob Nelson, Dan R. K. Ports, Yipeng Wang, Ren Wang, Charlie Tai, and Nam Sung Kim. Rambda: RDMA-driven Acceleration Framework for Memory-intensive μs-scale Datacenter Applications. In *Proceedings of the 29th IEEE Symposium on High Performance Computer Architecture (HPCA)*, Montreal, QC, March 2023.

[108] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Koblenz, Germany, October 2021.

[109] Ming Zhang, Yu Hua, and Zhijun Yang. Motor: Enabling Multi-Versioning for Distributed Transactions on Disaggregated Memory. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Santa Clara, CA, July 2024.

[110] Jiechen Zhao, Iris Uwizeyimana, Karthik Ganesan, Mark C. Jeffrey, and Natalie Enright Jerger. ALTOCUMULUS: Scalable Scheduling for Nanosecond-Scale Remote Procedure Calls. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Waikiki, Hawaii, December 2022.

[111] Lingjun Zhu, Yifan Shen, Erci Xu, Bo Shi, Ting Fu, Shu Ma, Shuguang Chen, Zhongyu Wang, Haonan Wu, Xingyu Liao, Zhendan Yang, Zhongqing Chen, Wei Lin, Yijun Hou, Rong Liu, Chao Shi, Jiaji Zhu, and Jiesheng Wu. Deploying User-space TCP at Cloud Scale with LUNA. In *Proceedings of the 2023 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2023.