FastWake: Revisiting Host Network Stack for Interrupt-mode RDMA

Bojie Li[1] Zihao Xiang[1] Xiaoliang Wang[2] Han Ruan[1] Jingbin Zhou[1] Kun Tan[1]
[1]Huawei [2]Nanjing University

ABSTRACT

Polling and interrupt has long been a trade-off in RDMA systems. Polling has lower latency but each CPU core can only run one thread. Interrupt enables time sharing among multiple threads but has higher latency. Many applications such as databases have hundreds of threads, which is much larger than the number of cores. So, they have to use interrupt mode to share cores among threads, and the resulting RDMA latency is much higher than the hardware limits. In this paper, we analyze the root cause of high costs in RDMA interrupt delivery, and present FastWake, a practical redesign of interrupt-mode RDMA host network stack using commodity RDMA hardware, Linux OS, and unmodified applications. Our first approach to fast thread wake-up completely removes interrupts. We design a per-core dispatcher thread to poll all the completion queues of the application threads on the same core, and utilize a kernel fast path to context switch to the thread with an incoming completion event. The approach above would keep CPUs running at 100% utilization, so we design an interrupt-based approach for scenarios with power constraints. Observing that waking up a thread on the same core as the interrupt is much faster than threads on other cores, we dynamically adjust RDMA event queue mappings to improve interrupt core affinity. In addition, we revisit the kernel path of thread wake-up, and remove the overheads in virtual file system (VFS), locking, and process scheduling. Experiments show that FastWake can reduce RDMA latency by 80% on x86 and 77% on ARM at the cost of < 30% higher power utilization than traditional interrupts, and the latency is only 0.3~0.4 μs higher than the limits of underlying hardware. When power saving is desired, our interrupt-based approach can still reduce interrupt-mode RDMA latency by 59% on x86 and 52% on ARM.

CCS CONCEPTS

• **Networks → Data center networks**.

KEYWORDS

RDMA, Host Network Stack, Interrupt, Context Switch

ACM Reference Format:
Bojie Li[1] Zihao Xiang[1] Xiaoliang Wang[2] Han Ruan[1] Jingbin Zhou[1] Kun Tan[1] and [1]Huawei [2]Nanjing University . 2023. FastWake: Revisiting Host Network Stack for Interrupt-mode RDMA. In *7th Asia-Pacific Workshop on Networking (APNET 2023), June 29–30, 2023, Hong Kong, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3600061.3600063

APNET 2023, June 29–30, 2023, Hong Kong, China

ACM ISBN 979-8-4007-0782-7/23/06…$15.00
https://doi.org/10.1145/3600061.3600063

1 INTRODUCTION

High-performance data center systems [3, 18, 36, 37] typically use polling to reduce latency and improve throughput, where each CPU core runs only one thread. However, many applications such as databases [11, 16, 22, 34] and web services [19, 23] often have more threads than CPU cores because they use a synchronous programming model. Rewriting these applications to a fully asynchronous model would require unaffordable development efforts. To enable time sharing of CPU cores among multiple threads, applications have to use interrupt mode, which would introduce 6~10 μs extra delay to deliver the interrupt from the NIC hardware and wake up the thread. In comparison, the latency to access remote memory in RDMA is only 1.6 μs [26], so, the interrupt mode latency is 5~7x of polling mode. Researchers have referred to this challenge as "*the killer microseconds*" [13].

The killer microseconds did not attract much attention in the age of traditional TCP/IP stacks and SSDs because they have > 20 μs latency, where thread wake-up only contributes a small portion. However, the advent of low-latency RDMA networking and fast storage hardware changes the story. To avoid wasting the low latency of RDMA hardware, the interrupt delivery path in host RDMA stack needs a complete revisit. Our goal is to redesign the host RDMA stack to efficiently support microsecond-scale I/O events with *commodity hardware, Linux OS, and unmodified applications.*

We analyze the overheads of host RDMA stack in interrupt mode through measurement. Our study reveals that *interrupt core affinity* is important for interrupt delivery latency. After RDMA NIC delivers the data, it triggers an interrupt to a core and the interrupt handler wakes up a thread, which only needs 4 μs on the same core while requiring 7~10 μs on another core. However, RDMA by default delivers interrupts to a random core, so there is low chance that the thread to wake up runs on the same core. Consequently, most interrupts involve the long delay for the interrupt handler to wake up a thread on another core. Executing interrupt handlers on a same core also limits multi-core throughput. Even if an interrupt wakes up a thread on the same core, the overhead is still significant because the NIC needs to generate events and trigger interrupts, and then the kernel spends time in tasklet, process scheduling, and context switch when waking up a thread.

Our solution, FastWake, is composed of two practical approaches: one completely removes interrupts but has higher power consumption, and the other greatly reduces interrupt latency without increasing power footprint. The system manager can divide the cores into the two approaches to balance latency and power utilization.

Our first approach completely avoid the interrupt overheads while still enabling multiple threads to share a CPU core. The

dilemma of polling and interrupt arises from the fact that multiple threads cannot poll simultaneously on the same core. We design a per-core dispatcher thread to poll the CQs of all threads on the same core, and context switch to the corresponding application thread when a completion event arrives. When the application thread completes processing and waits for the next event, it context switches back to the daemon thread to dispatch a next event. We build a kernel fast path to make context switching (0.3~0.4 μs) much faster than interrupt (3~4 μs) even if the interrupt is on the same core. This approach would keep CPU cores running at 100% utilization, but the latency is almost as low as polling mode.

When high power consumption is not allowed, we have to stick with the interrupts. Our second approach spreads NIC interrupts to different cores and makes use of completion vector to ensure interrupt core affinity. When a thread is rescheduled to a different core or the thread that polls CQ events alters, we leverage a less-known feature of RDMA NICs to dynamically adjust the mapping from CQ to event queue, so as to maintain interrupt core affinity. In addition, we shorten the kernel path of thread wake-up by removing the overheads in tasklet, locking, and process scheduling.

FastWake is fully compatible with existing RDMA applications, e.g., perftest [8]. Experiments show that with 16 threads sharing a core, FastWake can reduce end-to-end RDMA latency by 65%~83% (80% on average) on x86 and 64%~78% (77% on average) on ARM at the cost of up to 30% higher power utilization compared to traditional interrupt-mode RDMA. The resulting latency is only 0.3~0.4 μs higher than polling mode where each thread has its dedicated core. In scenarios with power constraints, our interrupt-based approach can still reduce end-to-end RDMA latency by 26%~64% (59% on average) on x86 and 25%~54% (52% on average) on ARM. As a side product, FastWake also provides new IPC primitives for fast thread wake-up. The latency of standard semaphore and pipe IPC is 3x~10x of FastWake IPC.

In summary, this work makes the following contributions:

(1) An analysis of the high latency in interrupt-mode RDMA revealing the importance of interrupt core affinity.
(2) An approach that introduces a per-core dispatcher thread and fast context switching to fully remove the interrupt overheads.
(3) An power-saving approach that significantly reduces interrupt latency by ensuring interrupt core affinity and shortening the kernel path.
(4) A system, FastWake, that implements the two approaches and is compatible with existing RDMA applications.

2 BACKGROUND

2.1 The Latency Hiding Problem

Many applications have more threads than CPU cores. For example, relational databases [11, 16, 22, 32, 34] have one thread per concurrent SQL query. Due to the I/O latency to access the permanent storage, the number of threads need to be larger than the number of CPU cores to fully utilize the computation power. For another example, many web service frameworks like J2EE [23], Python Flask [19], Python Django [21], Ruby on Rails [12], and PHP [9] use a multi-threaded programming model where each thread processes HTTP requests synchronously. As [13] points out, "synchronous programming leads to code that is shorter, easier-to-understand, more maintainable, and potentially more efficient". To fully utilize all CPU cores, multiple threads need to share a CPU core because the web application may need to wait for other microservices, databases or file storage during the processing of an HTTP request.

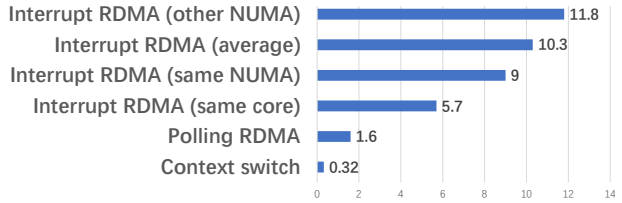


Figure 1: Latency (μs) comparison of RDMA in polling and interrupt modes on an x86 platform.

Utilizing Remote Direct Memory Access (RDMA) efficiently for these multi-threaded applications has long been a challenge [13] because polling mode cannot share a CPU core among multiple threads, while interrupt mode involves significant latency that undermines the low latency of RDMA. For example, RDMA Read in interrupt mode takes 10~13 μs while polling mode only takes 3 μs. Some may argue that busy polling until completion of the RDMA Read can resolve this problem. However, 95% of communication in data centers are RPCs [27] rather than one-sided RDMA. RPCs to other microservices and databases may take tens to thousands of μs, where busy polling wastes significant CPU resources. In addition, worker threads in databases and web services wait for incoming tasks which cannot be implemented with polling when multiple workers share a core.

Hiding this microsecond-scale latency has been a research focus for both programming language, operating system, and computer architecture communities. PL solutions include coroutines [24] for C/C++, and native concurrent programming abstractions in Go, Erlang, and Node.js. Coroutines typically have restrictions on the stack depth [2] and need to rewrite blocking OS APIs to coroutine APIs. As a result, it is hard to rewrite complicated applications such as databases with coroutines. Although goroutines [17], Erlang functions [40], and continuation passing style in Node.js [39] can exploit intra-thread concurrency elegantly, they cannot be applied to the popular web frameworks in other languages. OS solutions reduce microkernels that reduce the IPC overhead by accelerating context switches and bypassing the scheduler, such as L4 [20] and seL4 [28]; or leveraging hardware virtualization such as Skybridge [33] and Shinjuku [25]. However, these works target IPC and cannot reduce interrupt delivery latency. New hardware architectures support fast context switching at the cost of single thread performance [10, 38], or support more SMT threads per core [13]. However, these new hardwares are not readily deployable.

2.2 RDMA in Interrupt Mode

In RDMA, each QP (analogous to a connection) is associated with a CQ (Completion Queue). An RDMA NIC generates a CQE (Completion Queue Event) when a send/recv/read/write/atomic operation completes. When a CQ is created, it is associated with a *Completion Vector*[1] which determines to which CPU cores the NIC delivers interrupts. Each completion vector is implemented as an EQ (*Event*

[1]Although completion vector is named as "vector", it is a scalar value.

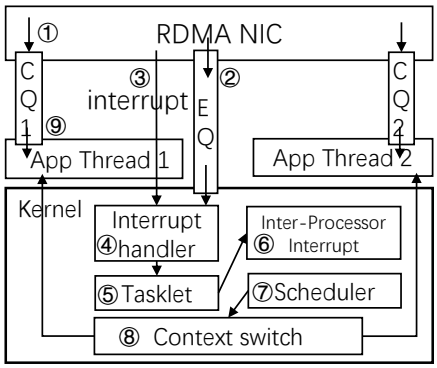


Figure 2: Traditional interrupt delivery path in RDMA, taking significant time in the delivery of NIC event queue entries and interrupts, IPI, tasklet, and thread scheduler.

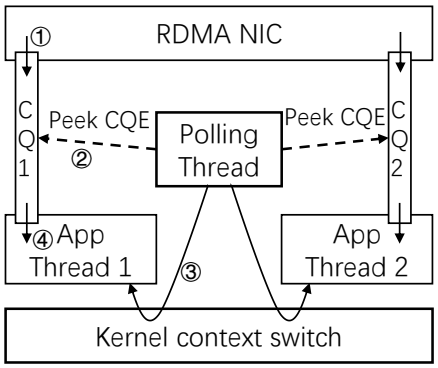


Figure 3: FastWake approach 1: per-core dispatcher thread, which polls all CQs of application threads running on the core and wakes up threads via fast context switch.

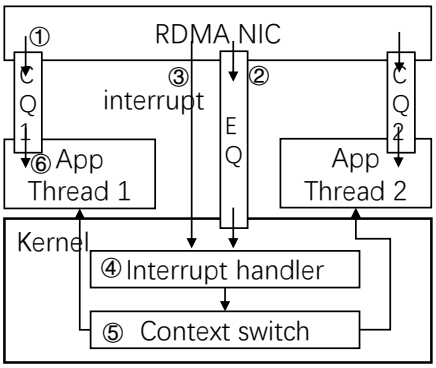


Figure 4: FastWake approach 2: fast interrupt delivery, which still uses NIC interrupts but removes IPI by ensuring interrupt core affinity and shortens interrupt delivery path.

Queue) in the kernel mode host memory. When interrupt is enabled for a CQ, as shown in Figure 2, the NIC first delivers a CQE to the CQ, then generates an EQE (*Event Queue Entry*) to the corresponding EQ, and then triggers the interrupt. The EQE carries the CQ number information so that multiple CQs can share an EQ. Writing the EQE into host memory and triggering the interrupt consumes ~1.5 μs. Each completion vector has an IRQ affinity mask, which is a bitmap of cores. When the CPU hardware receives the interrupt, it dispatches the interrupt to one of its cores specified by the IRQ affinity mask of the completion vector. Modern RDMA NICs support 64~128 completion vectors [6, 7], but many applications only use the default completion vector. The default IRQ affinity mask contains a large number of CPU cores, so the interrupts are spread to the cores randomly, resulting in poor interrupt core affinity.

When a CQ is created, it is also associated with a *Completion Channel*, which works in pure software to enable applications wait on multiple CQs associated with the completion channel. A thread waits on a completion channel via ibv_get_cq_event(), analogous to epoll_wait() in Linux. When an interrupt is dispatched to a CPU core, the interrupt handler in the top half polls the EQ to get an EQE containing the CQ number, and creates a tasklet to continue the remaining thread wake-up procedure asynchronously. In the tasklet, the kernel finds the completion channel associated with the CQ, and wakes up a thread waiting on the completion channel. This thread wake-up procedure in kernel takes ~2.5 μs when the thread is on the same core as interrupt handler. If the thread is on a different core, the kernel uses Inter-Processor Interrupt (IPI) to wake up the thread, which takes an additional 3~6 μs compared to same-core wake-up.

This leads to our first observation: *interrupt delivery to a thread running on the same core is much faster than other cores.* The thread wake-up latency, defined as the difference between RDMA Read latencies of interrupt mode and polling mode, is only 4 μs on the same core, 7 μs on other cores of the same NUMA node, and 10 μs on another NUMA node (see Figure 1). Making good use of interrupt core affinity is essential for low latency interrupt delivery.

To push latency to the limit, we make a second observation: *context switching is much faster than interrupt delivery.* As Figure 1 shows, with two threads context switching to each other using `sched_yield()`, a context switch only requires 0.32 μs in our x86 testbed. This is the fastest context switching path in mainline kernel as a system call already requires 0.2 μs [30] after applying the patches for Meltdown [31] and Spectre [29] vulnerabilities. In comparison, delivering an interrupt and waking up a thread requires 3~10 μs, and IPC via mutex or semaphore also takes 3~5 μs. The large latency gap is mainly due to the interrupt handlers, kernel scheduler, and virtual file system (VFS) for IPC. If each core can poll the CQs directly and context switch to the thread that waits on the CQ, it would eliminate the latency of EQE and interrupt generation, as well as the latency of thread wake-up in the kernel.

3 DESIGN

FastWake is designed to be a practical system that works with *commodity hardware* and *Linux OS*, *without modifications to existing RDMA applications.*

FastWake has two orthogonal approaches to reduce the interrupt latency of unmodified RDMA applications, as illustrated in Figure 3 and 4 respectively. The first approach uses per-core dispatcher threads to achieve minimal latency at the cost of high CPU utilization. The second approach, fast interrupt delivery, is power saving as it keeps using NIC interrupts to wake up the threads while significantly reducing the latency by improving interrupt core affinity and shortening the kernel path. A system manager can configure which cores use the polling thread approach and the others use fast interrupt delivery. The system manager can use standard *taskset* tool to assign core affinity of applications, and therefore choosing between the two approaches.

3.1 Per-core Dispatcher Thread

The first approach makes use of the second observation that context switching is much faster than interrupt delivery. Consider multiple application threads (which may belong to one or more processes) running on a same core, each of which waiting on completion events from a set of CQs. We aim to disable interrupts on the CQs and let the threads context switch to another thread with pending event. However, it is insecure to allow an application thread to peek

the CQEs of other applications. For security, we build a privileged dispatcher thread that polls all the CQs of the threads on the core in a round-robin manner. When the dispatcher polls a completion event (i.e., CQE), it context switches to the corresponding thread of the CQ. When the application thread completes processing of the event, it waits for another completion event via ibv_get_cq_event(), and the FastWake library context switches back to the dispatcher thread to handle the next event.

A challenge is that *the Linux kernel does not support fast context switching to a specific thread.* Existing IPC pritimives such as semaphore and pipe have latency as high as 3~7 μs, which is 10x~20x slower than a context switch. To this end, we design a new system call *switch_to(pid)* to perform the context switch without involving the kernel scheduler. The system call simply saves the context of the current thread, puts the current thread into interruptible state, and loads the context of the target thread, so it only takes 0.3~0.4 μs. Because an application thread may be rescheduled to another core, we check whether the thread is still in the waitqueue of the current core before switching to it. If the thread is rescheduled to another core, the switch_to() system call returns failure to the dispatcher thread, and then it transfers the CQs to the dispatcher thread of the new core where the thread runs.

The per-core dispatcher threads run in a daemon process. When an application creates a CQ, its memory is remapped to the daemon process so that the dispatcher thread can poll the CQ. The dispatcher thread only peeks the CQE but does not pop the CQE out of the CQ. The application thread polls the CQs as usual and pops the CQE out of the CQ. As an optimization, an application may poll all CQs and process all CQEs before switching back to the dispatcher, so the context switch overhead is amortized under heavy workload.

This approach only adds 0.4 μs latency to the 1.6 μs round-trip time of RDMA. However, the cost is that the CPUs run at 100% utilization. When the application threads are idle, the core is busy with the dispatcher thread which polls the CQs. The dispatcher thread has lower priority than the application threads, so, if an OS event (e.g., read completion of the file system or the release of a mutex lock) wakes up an application thread or an application thread is busy with computation tasks, the dispatcher thread will relinquish the CPU for the application.

The switch_to() system call may be abused to bypass kernel scheduling and starve other threads on the same core. So, we restrict the use of switch_to() by introducing a flag bit in the process control block to indicate whether it is a dispatcher thread. Non-dispatcher threads can only switch to dispatcher threads, and dispatcher threads can only switch to non-dispatcher threads. The flag bit can only be set via a root-only ioctl() interface.

3.2 Fast Interrupt Delivery

3.2.1 Interrupt Core Affinity. The second approach is designed for cases where busy polling is not allowed due to power constraints. It is based on the observation that waking up a thread on the same core as interrupt delivery is much faster than other cores. However, ensuring interrupt core affinity is not a simple task. The associativity between CQ and completion vector is determined when the CQ is created, and the completion vector determines which cores the interrupts would deliver to.

A strawman approach would be representing each CPU core with a completion vector and assigning the completion vector to the current core during CQ creation. More concretely, during system initialization, we configure the IRQ affinity mask so that each completion vector corresponds to one CPU core. As modern RDMA NICs support 64~128 completion vectors [6, 7], all CPU cores can be utilized. When a CQ is created, its completion vector is set to the CPU core it is running on. So, when the application polls on the CQ, the interrupt is delivered to the same core, resulting in low thread wake-up delay.

However, in a multi-threaded application, the thread creating the CQ may not be the thread polling the CQ, which is common where the main thread creates the CQ on the control plane and a worker thread polls the CQ on the data plane. In addition, a thread may be rescheduled to another core due to load imbalance among the cores, invalidating the interrupt core affinity.

Our approach leverages a lesser-known feature of commodity RDMA NICs: *CQ-to-EQ remapping* [1]. It enables dynamic updates to the completion vector of a CQ in the NIC. When the FastWake kernel module finds that the application thread is on a different core, it issues a CQ-to-EQ remapping command to the NIC. As a result, future interrupts for this CQ will be delivered to the new CPU core. We do not need to synchronize the CQ-to-EQ mapping strictly between the CPU and NIC because inconsistent mapping only results in temporarily longer delay of interrupts. Because CQ-to-EQ remapping simply updates one field in the CQ context of the NIC [1], it only takes 1 μs to notify the NIC. So, the penalty of an incorrect CQ-to-EQ mapping is the interrupt mode delay plus 1 μs for CQ-to-EQ remapping.

The number of CQ-to-EQ remappings would not exceed the number of CQ polling core migrations multiplied by the number of CQs. CQ polling core migrations have two sources: 1) different threads polling one CQ and 2) thread migration by the kernel scheduler. First, because CQs are protected by locks, high performance applications typically do not share CQs among multiple threads, otherwise lock contention would be a more prominent problem. Second, today's schedulers only migrate threads among cores at coarse granularity (e.g., every 4 ms for Linux) [35]. As a result, most completion events would have interrupt core affinity while thread migration has low penalty.

3.2.2 Shortening the Interrupt Delivery Path. As Sec.2 discussed, delivering an interrupt takes a long path in both hardware and software. The hardware overhead of delivering the EQE and the interrupt cannot be reduced with commodity hardware. However, we can simplify the interrupt delivery path in the kernel. The asynchronous tasklet mechanism aims to prevent long interrupt delivery path in the top half from blocking future interrupts to the CPU core. More importantly, code in the top half cannot use certain types of locks [15], while the wait queue of completion channel needs such locks. Our approach eliminates the tasklet scheduling latency by delivering the interrupt completely in top half. First, temporarily blocking interrupts to the core would not lead to loss of events because the interrupt is only a trigger and the events (i.e., CQ numbers) are contained in the EQ. Second, our approach simplifies the in-kernel data structure of the completion channel to remove the locks.

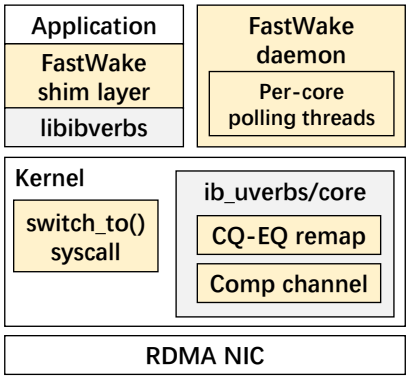


Figure 5: FastWake system architecture. Yellow components are new, and grey components are modified.

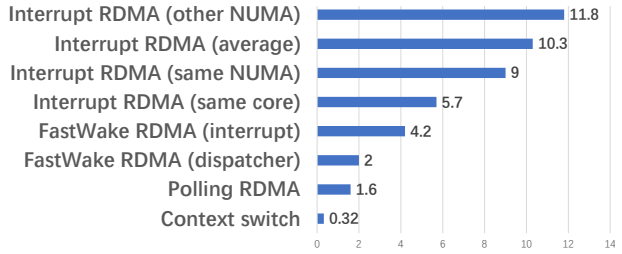
FastWake also makes an effort to wake up a thread faster. The standard way is to use the wake_up_process() kernel API which invokes the kernel scheduler to schedule the thread. To avoid the overheads, FastWake directly moves the thread from the waitqueue to the head of runqueue, and then context switches to the thread.

The application needs to call ibv_ack_cq_events() after every ibv_get_cq_event(), which takes ~0.15μs as it increments an event counter protected by a mutex. It forces ibv_destroy_cq() to wait until no unacknowledged events in order to avoid a race condition because the kernel stores CQ pointers in the pending event queue. If a CQ is destroyed and an interrupt from the NIC delivers an event to the EQ, a kernel segmentation fault would occur. FastWake totally removes the need for ibv_ack_cq_events() by storing CQ numbers instead of pointers in the pending event queue. If a CQ is destroyed and an unacknowledged event is delivered to the CQ, the kernel can find the invalid status of the CQ and ignore the event. Because CQ numbers are assigned in monotonically increasing order, destroyed CQ numbers will not be reused unless the CQ namespace is full. In the extreme case where the CQ number wraps around zero and a destroyed CQ number is reused, the thread may be mistakenly waken up, but the thread would simply go back to sleep.

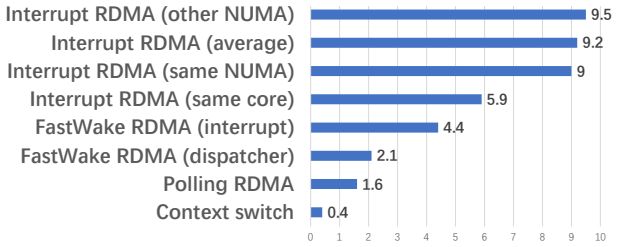
4 IMPLEMENTATION

As shown in Figure 5, FastWake is implemented in 5K lines of C code which contains modifications to the OFED [5] kernel and library, a kernel module that implements the switch_to(pid) system call, and a user-mode daemon that manages the per-core dispatcher threads. To minimize latency, we introduce a new system call rather than using *ioctl*. FastWake works with unmodified RDMA applications by introducing a shim layer above the libibverbs library that intercepts APIs regarding CQ, completion channel and event waiting.

As a side product, FastWake also supports fast inter-process communication (IPC). We introduce two system calls *hibernate()* and *wake_up_process(pid)*. A thread *pid* invokes hibernate() to sleep until another thread/process wakes it up using wake_up_process(*pid*). The hibernate() syscall simply puts the thread into interruptible waiting state. The wake_up_process(pid) syscall first checks whether the thread is in the waitqueue of the current core, and directly resumes the thread if so. Otherwise, it generates an inter-processor interrupt to the thread's current core, where the interrupt handler directly resumes the thread.



(a) X86 architecture.



(b) ARM architecture.

Figure 6: Latency (μs) of FastWake and traditional RDMA.

5 EVALUATION

We evaluate FastWake on two testbeds: an x86 testbed with dual-socket Intel Xeon Gold 6151 CPUs and an ARM testbed with dual-socket Kunpeng 920 CPUs [4]. Each server is equipped with an NVIDIA ConnectX-5 NIC [6]. The NICs of two servers are directly connected with 100 Gbps fibers. Traditional RDMA latency in polling mode is measured by perftest [8] with 64-byte RDMA Send, where `ib_send_lat` works like an RPC and reports half of the round-trip latency. For interrupt mode, we run 16 perftest [8] processes on the same core of the server to create a core sharing scenario. The perftest client is modified to connect to all the server processes and generate 64-byte RDMA Sends to a randomly selected process on the server. Then, a random process is waken up on the server and responds a message, and the client is waken up and receives the response.

Figure 6 compares the latency of FastWake and traditional RDMA. FastWake (dispatcher) corresponds to the per-core dispatcher thread approach. FastWake (interrupt) corresponds to the fast interrupt delivery approach. On both x86 and ARM, the per-core dispatcher thread approach yields minimal latency, which is only 0.4~0.5 μs higher than application polling. The additional latency is basically the context switch latency (0.3~0.4 μs) plus $< 0.1\mu$s processing time in the user-space shim layer. When all cores use this approach, the server utilizes 30% higher power compared to idle state, which agrees with the measurement of [14].

The fast interrupt delivery approach has 2.2~2.3 μs higher latency than the per-core dispatcher approach. This is because the NIC needs to first write an EQE into the event queue in host memory and then generate an interrupt to the CPU, which takes ~1.5 μs across the PCIe. Next, the interrupt handler also takes time to determine the CQ and the waiting process. Nevertheless, FastWake is 1.5 μs faster than traditional RDMA even when the interrupt and application thread are co-located on the same core. This can attribute to the full-stack optimizations that remove tasklet scheduling (0.6 μs), locking (0.3 μs), and the kernel scheduler (0.6 μs).

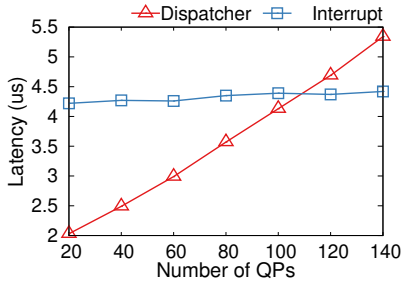


Figure 7: Latency (μs) of FastWake on x86 with different number of QPs.

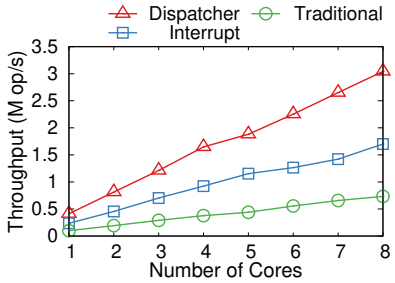


Figure 8: Throughput of FastWake and traditional interrupt-mode RDMA on x86 with different number of CPU cores.

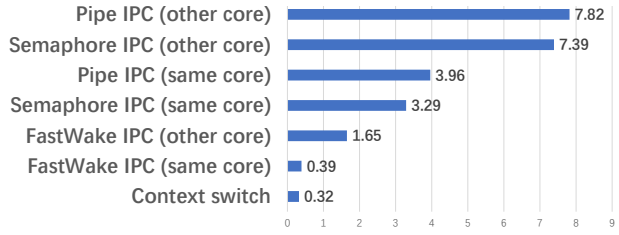
Compared to traditional RDMA without interrupt core affinity, FastWake reduces average latency by 80% on x86 and 77% on ARM. The average latency of traditional RDMA is in the middle of the latency on the same NUMA and that on another NUMA because the application runs on a random core where the chance that it runs on the same core is low.

The per-core dispatcher thread polls all CQs of the threads on the core. As Figure 7 shows, its latency increases linearly with the number of QPs. In contrast, the latency of fast interrupt delivery approach keeps constant with different number of QPs.

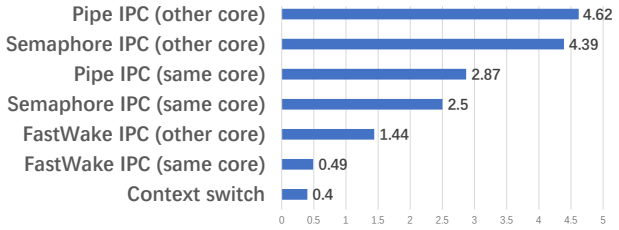
Figure 8 compares the throughput of two FastWake approaches and traditional interrupt-mode RDMA. Each core on the server runs 16 processes and the requests are randomly sent to a process. FastWake with dispatcher has 1.76x throughput of FastWake with interrupt, and 4.21x of traditional RDMA. All three approaches show 90%~93% per-core throughput with 8 cores compared to single-core performance because there is no inter-core coordination.

To benchmark the effectiveness of dynamic CQ-to-EQ remapping when the application migrates to other cores, we use *taskset* to force core migration. We find that for the first time after core migration, FastWake shows latency close to that of traditional RDMA (9.8 μs for x86 and 9.1 μs for ARM) because waking up a thread on another core takes time, and issuing a command to update the CQ-to-EQ mapping in the NIC takes an extra 1 μs. However, for subsequent requests, the latency resumes to the level before core migration because the interrupts are generated on the new core.

Finally, we compare the latency between FastWake IPC and traditional IPC mechanisms (pipe and semaphore). Pipe and semaphore are based on the virtual file system (VFS), and the wake-up process involves the kernel scheduler. So, even when the target thread runs on the same core, the latency is still high. In contrast, FastWake IPC is simply a wrapper around a context switch in the same-core case, so the standard IPC has 5x~10x latency of FastWake IPC. For inter-core thread wake-up, FastWake IPC greatly simplifies the IPI delivery path so that the standard IPC has 3x~5x latency of it.



(a) X86 architecture.



(b) ARM architecture.

Figure 9: Latency (μs) of FastWake and traditional IPC.

6 DISCUSSION

Hardware feasibility. To our knowledge, CQ-to-EQ remapping is only supported by Mellanox NICs. For other NICs, we can use an alternative approach which keeps CQ-to-EQ mapping constant and updates IRQ core affinity on the fly. Each thread is assigned to a unique EQ, i.e, IRQ number. When the thread is migrated to another core, the IRQ core affinity is updated accordingly. However, this approach can only scale to 64~128 threads because of limited IRQ numbers supported by the NIC.

Scheduling fairness. FastWake bypasses the kernel scheduler. By moving a thread to the head of runqueue, its priority is implicitly increased. However, its priority is still lower than other runqueues with higher priority, so high priority applications still take precedence.

Future work. This work only microbenchmarks latency of small messages on idle hosts. Future evaluations should measure the latency under heavy workloads and the end-to-end performance of real-world applications. Due to the lightweight nature of coroutines, its context switching overhead is lower than FastWake. We expect a comparison in both performance and programmability aspects.

7 CONCLUSION

Many applications have to use RDMA interrupt mode due to its high number of threads, leading to 4x~6x additional latency of the underlying hardware. This paper presents FastWake, a practical solution to reduce such latency for unmodified applications by redesigning the host RDMA stack. Some cores can run per-core dispatcher threads and utilize fast context switch to achieve minimal latency overhead at the cost of high CPU utilization. The other power-saving cores still cut latency by half by improving interrupt core affinity and optimizing interrupt delivery path.

REFERENCES

[1] 2020. Mellanox Adapters Programmer's Reference Manual (PRM). (2020). https://network.nvidia.com/files/doc-2020/ethernet-adapters-programming-manual.pdf.

[2] 2023. Boost Coroutine. (2023). https://www.boost.org/doc/libs/1_58_0/libs/coroutine/doc/html/coroutine/coroutine.html.

[3] 2023. DPDK: Data Plane Development Kit. (2023). https://www.dpdk.org/.

[4] 2023. Kunpeng 920 ARM-based server CPU. (2023). https://www.hisilicon.com/en/products/Kunpeng/Huawei-Kunpeng/Huawei-Kunpeng-920.

[5] 2023. Mellanox OFED. (2023). https://network.nvidia.com/products/infiniband-drivers/linux/mlnx_ofed/.

[6] 2023. NVIDIA Mellanox ConnectX-5 adapters. (2023). https://www.nvidia.com/en-us/networking/ethernet/connectx-5/.

[7] 2023. NVIDIA Mellanox ConnectX-6 adapters. (2023). https://www.nvidia.com/en-us/networking/ethernet/connectx-6/.

[8] 2023. Perftest: Infiniband Verbs Performance Test. (2023). https://github.com/linux-rdma/perftest.

[9] 2023. PHP: FastCGI Process Manager (FPM). (2023). https://www.php.net/manual/en/install.fpm.php.

[10] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. 1990. The Tera computer system. In *Proceedings of the 4th International Conference on Supercomputing*. 1–6.

[11] Hillel Avni, Alisher Aliev, Oren Amor, Aharon Avitzur, Ilan Bronshtein, Eli Ginot, Shay Goikhman, Eliezer Levy, Idan Levy, Fuyang Lu, et al. 2020. Industrial-strength OLTP using main memory and many cores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3099–3111.

[12] Michael Bächle and Paul Kirchberg. 2007. Ruby on rails. *IEEE Softw.* 24, 6 (2007), 105–108.

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

[14] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. The datacenter as a computer: Designing warehouse-scale machines. *Synthesis Lectures on Computer Architecture* 13, 3 (2018), i–189.

[15] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. 2005. *Linux device drivers*. "O'Reilly Media, Inc.".

[16] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1243–1254.

[17] Alan AA Donovan and Brian W Kernighan. 2015. *The Go programming language*. Addison-Wesley Professional.

[18] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 401–414.

[19] Miguel Grinberg. 2018. *Flask web development: developing web applications with python*. " O'Reilly Media, Inc.".

[20] Gernot Heiser and Kevin Elphinstone. 2016. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Transactions on Computer Systems (TOCS)* 34, 1 (2016), 1–29.

[21] Adrian Holovaty and Jacob Kaplan-Moss. 2009. *The definitive guide to Django: Web development done right*. Apress.

[22] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[23] Rod Johnson. 2005. J2EE development frameworks. *Computer* 38, 1 (2005), 107–110.

[24] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. 2018. Exploiting coroutines to attack the" killer nanoseconds". *Proceedings of the VLDB Endowment* 11, 11 (2018), 1702–1714.

[25] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 345–360.

[26] Anuj Kalia, Michael Kaminsky, and David G Andersen. [n. d.]. Design guidelines for high performance RDMA systems. In *USENIX Annual Technical Conference (ATC), pages=437–450, year=2016*.

[27] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 158–169.

[28] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 207–220.

[29] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101.

[30] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. 2019. SocksDirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*. 90–103.

[31] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, et al. 2020. Meltdown: Reading kernel memory from user space. *Commun. ACM* 63, 6 (2020), 46–56.

[32] Yunus Ma, Siphrey Xie, Henry Zhong, Leon Lee, and King Lv. 2022. HiEngine: How to Architect a Cloud-Native Memory-Optimized Database Engine. In *Proceedings of the 2022 International Conference on Management of Data*. 2177–2190.

[33] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. 2019. Skybridge: Fast and secure inter-process communication for microkernels. In *Proceedings of the Fourteenth EuroSys Conference*. 1–15.

[34] Bruce Momjian. 2001. *PostgreSQL: introduction and concepts*. Vol. 192. Addison-Wesley New York.

[35] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads.. In *NSDI*, Vol. 19. 361–378.

[36] Gregory F Pfister. 2001. An introduction to the infiniband architecture. *High performance mass storage and parallel I/O* 42, 617-632 (2001), 102.

[37] Luigi Rizzo. 2012. netmap: a novel framework for fast packet I/O. In *21st USENIX Security Symposium (USENIX Security)*. 101–112.

[38] Burton J Smith. 1978. A pipelined, shared resource MIMD computer. In *Proc. 1978 Int. Conf. on Parallel Processing, IEEE*. 6–8.

[39] Stefan Tilkov and Steve Vinoski. 2010. Node. js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing* 14, 6 (2010), 80–83.

[40] Robert Virding, Claes Wikström, and Mike Williams. 1996. *Concurrent programming in ERLANG*. Prentice Hall International (UK) Ltd.