TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Systems Architecture Chair of Operating Systems

Diploma Thesis

Coordinated Polling in User Space

(Daemon-Assisted Batched Polling in User Space)

Fabius Mayer-Uhma
Matriculation number: 4848386

Dresden, 27.04.2026

Supervisor
Dipl.-Inf. Jan Bierbaum

Supervising Professor
Prof. Dr.-Ing. Horst Schirmeier

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

Aufgabenstellung für die Anfertigung einer Diplomarbeit

Studiengang:	Diplom
Studienrichtung:	Informatik (2010)
Name:	Fabius Mayer-Uhma
Matrikelnummer:	4848386
Titel:	Coordinated Polling in User Space

High-performance networks (e.g., InfiniBand, Aries, RoCE, etc.) have not only been ubiquitous in HPC systems, but also gain traction in data centers. While these networks provide low latency and high bandwidth connections between different machines, they usually rely on busy polling to achieve these properties: After initiating a send or receive operation, a process will continuously query the hardware for the operation's completion and, thus, occupy a CPU without doing useful work. This behaviour, however, incurs high energy consumption — especially for long-lasting communication operations — and create serious problems with oversubscribed CPU cores.

At the other end of the spectrum lies the interrupt or event-based mode, where the network card triggers the operating system whenever a communication operation finishes. This approach allows to block waiting processes and free the CPU; either for use by other processes or to save energy. However, setting up and processing the interrupt causes significant latency overhead.

Coordinated Polling still avoids interrupts but outsources the polling for completions to a dedicated process; the waiting application processes block. Once it finds that a communication operation has finished, the polling process wakes up the application process that originally initiated this communication. Thereby, Coordinated Polling serves as a middle ground between the two extremes of process-local polling and interrupts.

The primary goal of this thesis is to implement Coordinated Polling for InfiniBand-based applications on GNU/Linux. In contrast to previous work in this area, the resulting implementation should modify neither the kernel nor the applications, such that it can be deployed transparently in an existing (HPC) system. To support MPI-based applications, a timeout mechanism will be necessary that reactivates the application process after a specified time, even if no communication is completed.

As a first step, the student should modify the RDMA core userspace libraries („IB verbs"), the main interface of InfiniBand, with a simple facility to allow a polling process to access the necessary data structures (CQs). The implementation also needs a mechanism to block and unblock processes: application processes should not run while waiting for communication, whereas the polling process should not run when there is no communication in progress. For the sake of simplicity, it is advisable to start with a single polling process per machine.

Once the base implementation is functional, the student should evaluate its performance using IB-verbs-based and MPI-based benchmarks. As time permits, the student may explore design variations and optimisations such as

- comparison to an implementation with kernel support for fast process switching,
- polling process placement (e.g., dedicated core vs. time-sharing with application processes),
- a multi-threaded polling process (e.g., one thread per core/socket/NUMA domain),
- different polling strategies (e.g., core-local vs. global).

Gutachter: Dr.-Ing. Nils Asmussen
Betreuer: Jan Bierbaum
Ausgehändigt am: 24. November 2025
Einzureichen am: 27. April 2026

Prof. Dr.-Ing. Horst Schirmeier
Betreuender Hochschullehrer

Statement of authorship

I hereby certify that I have completed this work on my own and without the help of anything other than the indicated sources and resources.

Disclosure of LLM usage

During the writing of this thesis, I used large language models for generating data visualizations and correcting spelling and grammar errors. All technical content is entirely my own work.

Dresden, 27.04.2026

Fa. Mayer-Uhma

Fabius Mayer-Uhma

Abstract

High-performance computing systems and modern data centers rely on RDMA to provide low-latency, high-bandwidth communication between nodes. Applications commonly access asynchronous network resources using either busy polling or interrupt-driven, event-based mechanisms. Busy polling dedicates a CPU core to continuously monitor completion queues and execute processing code upon RDMA operation completion. While this approach achieves sub-microsecond detection latency, it wastes CPU cycles on idle resources and does not scale under oversubscription, leading to increased latency due to core contention and higher energy consumption. In contrast, event-based approaches rely on the NIC to notify the operating system via interrupts, allowing waiting processes to block and freeing CPU cores. Although this method suits oversubscribed systems better and reduces energy usage, interrupt handling and subsequent rescheduling of user processes introduce multi-microsecond latency overhead.

Daemon-Assisted Batched Polling (DABP) occupies a middle ground between these two extremes by delegating completion queue polling to a centralized daemon. Application processes register their completion queues with the daemon and block while waiting for completions. The daemon continuously monitors registered queues and selectively wakes the corresponding application processes upon completion, avoiding process-local busy waiting.

This thesis presents the design and implementation of DABP for InfiniBand in user space on GNU/Linux without requiring kernel modifications. The implementation uses a polling daemon in combination with futex-based synchronization to block and resume application processes. For MPI workloads, integration targets the middleware layer, leaving individual applications unchanged. A fully transparent mode replaces the libibverbs polling verb directly and requires no application source changes. Experimental results show that DABP reduces average detection latency compared to interrupts from $\sim$3.7 µs to $\sim$1.8 µs for locality-aware daemon placements. A comprehensive evaluation using InfiniBand verbs microbenchmarks, MPI application benchmarks and the Valkey in-memory store examines the impact of DABP under different daemon placement strategies and process locality configurations, highlighting its benefits in oversubscribed environments and identifying opportunities for further optimization.

Table of Contents

List of Tables

List of Figures

List of Acronyms

API: application programming interface
CDF: cumulative distribution function
CPU: central processing unit
CQ: completion queue
CQE: completion queue entry
DABP: Daemon-Assisted Batched Polling
DDR4: double data rate 4
DMA: direct memory access
DPDK: Data Plane Development Kit
HPC: high-performance computing
HT: hyperthread
IPC: inter-process communication
IPI: inter-processor interrupt
LLM: large language model
MPI: Message Passing Interface
NIC: network interface card
NPB: NAS Parallel Benchmarks
NUMA: non-uniform memory access
PCI: Peripheral Component Interconnect
QP: queue pair
RAPL: Running Average Power Limit
RDMA: remote direct memory access
RoCE: RDMA over Converged Ethernet
SMT: simultaneous multithreading
UCX: Unified Communication X

1 Introduction

Modern large-scale computing systems such as HPC machines, GPU clusters and datacenters rely on low-latency interconnects to transfer data between nodes. RDMA powers this communication using specialized hardware such as InfiniBand or RoCE, with stacks operating at microsecond timescales. Applications waiting for hardware resources traditionally use one of two approaches: (1) busy-polling dedicates a CPU core to continuously checking a resource, giving very low latency but burning CPU time even when no data is available, and (2) interrupts allow hardware to notify the CPU directly, freeing it while no data is available and requiring CPU time only when an event arrives, but each notification requires kernel entry, scheduling and a context switch back to user space, adding overhead in the microsecond range [1].

Busy-polling's resource cost becomes particularly acute under oversubscription. When more processes busy-poll than CPU cores are available, processes with ready data must still compete for scheduler time: the scheduler cannot distinguish a polling process from one doing useful work, so latency degrades significantly for all busy-polling workloads.

RDMA software commonly relies on an unsatisfactory choice between these two approaches. This thesis introduces Daemon-Assisted Batched Polling (DABP), which uses a central polling daemon that batches multiple resources together and polls them centrally. Instead of driving progress themselves, applications block and transfer control to the daemon. When the daemon observes readiness, it wakes the blocked application. This achieves readiness-detection latency close to busy-polling while bounding the CPU cost to a fixed number of daemon cores, regardless of how many applications are waiting. The entire mechanism operates in user space without kernel modifications or elevated permissions.

Prior work such as FastWake [2] and SmartInterrupts [3] addresses similar problems but requires kernel modifications, ruling out deployment in restricted HPC environments. Neither fully develops batched polling as a technique in its own right. Their evaluations remain narrow: FastWake covers only microbenchmarks and SmartInterrupts only MPI. DABP fills this gap by operating entirely in user space and covering a broader spectrum of RDMA workloads.

InfiniBand dominates in HPC. Driven by the rise of LLM training, datacenters have also adopted it in large numbers. More recently, Ethernet has been gaining ground as RoCE implementations have matured, and both technologies implement RDMA. This thesis focuses on RDMA over InfiniBand and primarily targets MPI-based applications but also covers the general integration of DABP into other software. HPC workloads prioritize lowest latency, so busy-polling stacks are common. DABP can reduce energy consumption and support oversubscription while preserving that latency. In datacenters, workloads diverge: distributed LLM training uses busy-polling [4] while multi-tenant setups such as webservers and in-memory caches rely on interrupts [2]. DABP benefits both domains. For interrupt-based applications, DABP cuts average latency by up to 49.6%. For busy-polling MPI workloads using RDMA-only communication, same-core DABP introduces between 1.8% and 3.6% overhead in total runtime while keeping energy consumption close to the busy-polling baseline. Under 2x oversubscription, total runtime falls below 10% of the oversubscribed busy-polling baseline depending on the workload. Even cross-socket DABP wakes achieve lower average latency than interrupts.

RDMA applications implement waiting strategies at different levels of the software stack. DABP must therefore identify and replace the semantic wait sites, the points in program code where the application decides to wait for a future event, rather than substituting low-level primitives alone. To allow integration into existing programs, this replacement must be transparent and leave application behavior unchanged. DABP minimizes source code changes, covering a large fraction of HPC programs through modifications to Open MPI.

This thesis makes the following contributions:

1. The design of DABP, a daemon-assisted batched polling mechanism that achieves readiness-detection latency below interrupts while releasing the CPU during idle periods.
2. A transparent user space implementation requiring no Linux kernel modifications and no elevated permissions.
3. Integration into HPC and datacenter workloads by identifying semantic wait sites and replacing the existing progression mechanism with DABP.
4. An evaluation across microbenchmarks, HPC MPI workloads and an interrupt-based in-memory cache on real InfiniBand hardware.

This thesis comprises five further chapters. Chapter 2 establishes the background and surveys related work before Chapter 3 and Chapter 4 present the DABP design and its integration into existing software stacks. Chapter 5 then evaluates the system across microbenchmarks, HPC MPI workloads and a representative datacenter in-memory cache. Chapter 6 concludes the thesis with limitations and future work.

2 Background and Related Work

Batched polling is a broad concept and this work applies it to InfiniBand. The final design and its tradeoffs build on four foundational areas: the general mechanisms programs use to wait for asynchronous resources (Section 2.1), their Linux-specific implementation (Section 2.2), the InfiniBand hardware and software model (Section 2.3) and the existing RDMA application stack (Section 2.4). The chapter closes with an overview of related work. This thesis marks references to defined terminology in blue and each links back to its definition.

2.1 Synchronizing Asynchronous Resources

Computer systems commonly attach peripheral devices to the CPU. Due to the CPU being the main component for processing, information exchange between it and the hardware is essential. Many devices deliver *asynchronous* and sporadic information. Here, asynchronous means that an event may occur later and independently of the current control flow and therefore requires a mechanism of waiting and *synchronization*. Programs traditionally wait for and handle hardware events in two ways: by busy-polling or by using interrupts.

2.1.1 Polling and Interrupts

Busy polling (busy checking, spinning) allocates a CPU core to repeatedly check if hardware has new information. The CPU repeatedly reads a memory location and continues once the polled value indicates that new data has arrived. This continuous checking minimizes hardware event detection latency, which is why low-latency systems prefer it. This approach always requires a full CPU core, making it impossible to run other programs on the same core at the same time. The result is less efficiently used hardware and greater energy consumption.

The second approach is to use *interrupts*. Interrupts use a hardware signal routed through an interrupt controller to notify the CPU. An application registers interest in a specific hardware event and suspends its current computation so other applications can run. The interrupt signal takes the CPU out of its current execution context and transfers control to an interrupt handler (vector, routine). The operating system kernel defines an *interrupt handler* that processes the incoming hardware event. This approach requires CPU time only when hardware data is actually present, making concurrency possible. The result is more efficiently used hardware and less overall energy consumption.

Progression is the continued advance through program code. *An asynchronous resource* requires a special *progression mechanism* because the intent of waiting for data can be modeled by different mechanisms. A program reaches a *semantic wait site* when it decides to wait for an *event*. Busy-polling, interrupts or DABP are progression mechanisms that drive progression. Once the progression mechanism observes an asynchronous event, the resource reaches *readiness*. At this point the asynchronous data has become synchronous and synchronized with the CPU. The data can now be processed by the core.

2.1.2 User space, Kernel space, Syscalls and DMA

General-purpose operating systems enforce protection mechanisms because computers may run potentially malicious software. Figure 1 shows user space and kernel space and how they interact with RAM and other hardware. An application runs in a less privileged

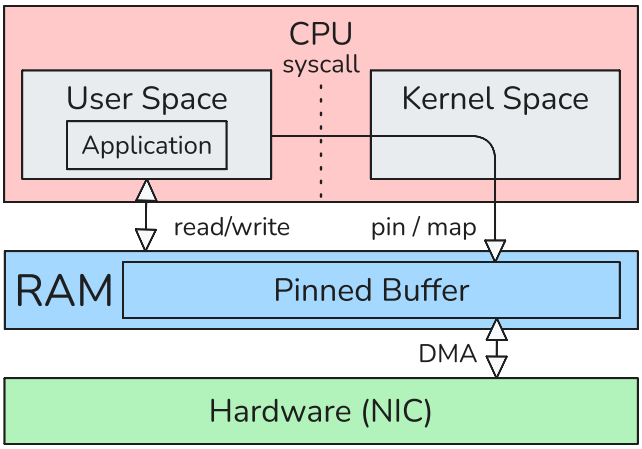


Figure 1: User space, kernel space and DMA

user space where not all hardware operations are allowed. In this mode it can only access RAM or hardware by first requesting the kernel.

A request from user space to the kernel is a syscall. The kernel space is the more privileged environment that directly controls the hardware. Kernel space can also define interrupt handlers. Kernel space can map RAM into a user space program's address space so the program can access that memory directly. After allocating RAM with the help of the kernel, the user space can now access it directly.

The kernel can allocate and pin special RAM areas. Pinning makes the memory stay at the same address and hardware can use this memory directly. A NIC can use pinned RAM to directly write to the RAM without CPU involvement and this process is *DMA*. With another syscall, the kernel can pin allocated memory and set it up for DMA so a user space application can access the NIC directly. Direct NIC access from user space removes kernel overhead when programs communicate with the NIC.

2.1.3 Concurrent Programming and Context Switches

Since DMA data can be sporadic or arrive in the future, programs must synchronize access to it. Programs can implement the busy polling approach directly in user space using atomic instructions. An atomic instruction is a CPU operation that completes as an indivisible unit: no other core or thread can observe a partially-executed state. If multiple threads contend for the resource, the CPU must wait before executing the following code.

Busy-polling is non-cooperative: the polling loop holds the CPU core without yielding to the scheduler, preventing other tasks from running on that core while the loop is active. Cooperative concurrent programming instead uses OS-defined *blocking* or *event-based* primitives that yield the CPU, allowing the scheduler to execute other tasks while a resource is awaited. Blocking describes the voluntary cooperative suspension of a process, requesting the kernel scheduler to wake it after an event.

Operating system kernels implement a process scheduler. It uses different algorithms to decide which thread gets to run next. A *context switch* occurs when the kernel saves the current thread's execution state and restores that of the next. A thread *yields* when it voluntarily switches to the scheduler. If a user application uses blocking it notifies the scheduler that it does not want to run until something happens. The wakeup event can

be linked to an interrupt handler. After the kernel processes an interrupt routine it can reschedule corresponding user code.

The interrupt-based approach has one main drawback: it includes the overhead of the kernel interrupt handler, the notification to reschedule the user thread, the scheduler deciding that the corresponding thread should run and the context switch back to user code. “Attack of the Killer Microseconds” [1] identified this overhead early. A user space busy polling approach can dispatch the processing routine immediately, thus reducing overhead to the bare minimum.

The common implementation of blocking primitives in operating systems consists of a short fastpath that starts by busy polling and falls back to actual blocking eventually, as exemplified by the Linux futex design [5]. Pusukuri et al. [6] described this as the state-of-the-art spin-then-block contention management policy. This spin-then-block strategy can be highly beneficial when programs are well synchronized with the events they await, so the event commonly occurs during the short busy-poll. However, the slow path still suffers from the full event-induced latency.

2.1.4 Semantic Wait Sites and Async Runtimes

Multiple programming languages like Python or Rust provide syntax for asynchronous events. They use the `async` keyword to define asynchronous functions and the `await` keyword to synchronize. When awaiting a resource, the program gives up control flow and transfers it to a runtime. The runtime must decide how to wake the application when it receives the asynchronous resource and which progression mechanism to use. The runtime decides between busy polling, interrupts, thread usage or process usage. This thesis refers to the intent behind an application’s `await` as a semantic wait site. Runtimes vary in their approach: most rely on event-based mechanisms, while some are closer to DABP. For example, Rust’s `Glommio` [7] is a thread-per-core runtime built on `io_uring` and uses `io_uring`-based polling to coordinate multiple event sources.

2.1.5 Oversubscription

In addition to wasting energy while polling idle hardware resources, busy-polling has a second drawback. CPUs have a bounded number of cores and modern applications such as in-memory data structure stores can have many more user applications requesting hardware resources than available CPU cores. If there are more busy-pollers than CPU cores, the system becomes oversubscribed and low latency can no longer be guaranteed for all of them. This state is called *oversubscription*.

In this scenario, some busy-pollers do not run immediately. In the worst case, the hardware resource arrives, but the user code has to wait until the next scheduling event. This event is a timed and configurable kernel event, often set to every millisecond, during which the scheduler switches between applications to allocate computation time. The scheduler cannot differentiate between busy-pollers that have data and ones that do not, so a poller with data could starve for a long time and drastically increase latency as oversubscription grows. On the other hand, oversubscription can increase system utilization for imbalanced applications, since idle cores that would otherwise go unused can absorb work from busier ones.

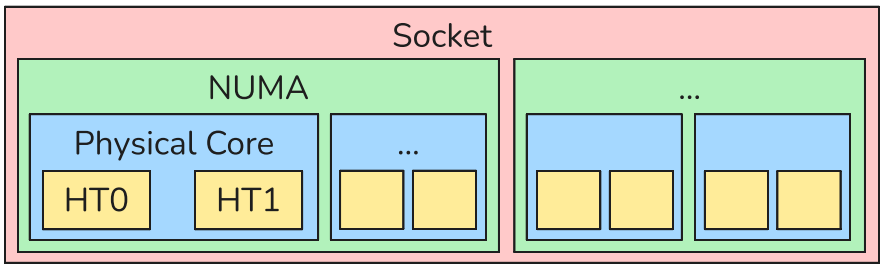


Figure 2: CPU topology hierarchy

2.1.6 CPU Topology

Modern CPU topology is relevant for the following chapters because blocking primitives are topology-dependent. Figure 2 shows the most important building blocks. A physical CPU core in blue is a full execution engine with among other things ALUs, FPUs, pipelines, registers and L1, L2 cache. Modern systems implement hyperthreading or simultaneous multithreading (SMT). With SMT each physical core exposes multiple logical cores that share most of their resources and are colored yellow. Hyperthreads have individual architectural registers, including the instruction pointer so they can execute instructions independently. Because sibling hyperthreads run on the same physical core and share core-local structures, interaction between them has lower latency than interaction across cores. Common hardware has two hyperthreads per physical core.

The next slower unit for cross-CPU communication is a NUMA domain, shown in green in the figure. A NUMA domain is a collection of physical CPU cores that share local DRAM and possibly slices of L3 cache. A CPU socket groups multiple NUMA domains.

A CPU socket, colored red in the figure, is the motherboard slot that holds one CPU. Shared resources are hardware-specific but normally there is another extra level of interconnect needed, resulting in higher latency communication between sockets.

2.2 Linux Mechanisms

The project requires close cooperation with the kernel. Most RDMA applications target Linux, so Linux is the relevant platform for the mechanisms discussed in this thesis. Additionally, Linux exposes the user space RDMA driver and many MPI and RDMA applications in publicly accessible repositories. This section covers the concrete Linux implementations of the concurrency primitives and hardware interaction introduced above.

2.2.1 Shared Memory and IPC

Linux models concurrency using tasks. A task in Linux is a schedulable execution context consisting of CPU state, stack, scheduling metadata and references to resources (heap, files, hardware). A task is the current execution point in a program and contains everything needed to resume it. A thread is a task that shares memory with other tasks. A process is one or more tasks that share an address space. Different processes do not inherently share address space unless they explicitly establish shared memory.

Shared memory is the underlying principle for the fastest possible inter-process communication (IPC). It uses zero copy and directly benefits from local cachelines. Different flags such as read-only and read-write manage access control for shared memory. The kernel more commonly allocates anonymous memory, but for named shared memory it returns a file descriptor that holds the capability to map the memory. With the right permissions and

a global constant name for the memory, another process can use it to map the memory. A safer approach is to use *Unix domain sockets*. `memfd_create` creates a shared memory handle without a global name or global visibility and the kernel cleans it up automatically when its file descriptors (FDs) close [8]. A Unix domain socket is a kernel communication channel addressed by a filesystem path and applications commonly use it to exchange data or pass shared memory handles across process boundaries.

2.2.2 Waiting and Synchronization Primitives

Shared memory provides the underlying resources to processes, but processes still must synchronize access to it to avoid race conditions. Commonly used synchronization operations are hardware atomic instructions or kernel-backed primitives. A mutex prevents simultaneous access to shared resources by blocking any task that attempts to acquire it while another holds it. Linux provides multiple syscalls for event-based blocking progression. The most common approach is `eventfd`, a file descriptor for event notification [9]. After creating the `eventfd`, the application can call `poll`, `select` or `epoll` to yield to the kernel until an event occurs, at which point the kernel returns control to the user application [10]. A low-overhead event-based sleep/wake primitive in Linux is the fast user space mutex *(futex)*. Futexes provide the same functionality as a mutex and heavily optimize the path to skip as many parts of the kernel and scheduler as possible [11], [12]. Private futexes are limited to one process, while shared futexes work across process boundaries using shared memory [13]. `futex_wait` lets a task yield to the kernel until some memory address (futex word) is changed or some timeout is met. On the other hand, the `futex_wake` syscall wakes a configurable amount of currently waiting tasks on a futex address [12]. Since Linux kernel version 5.16 (release November 2021) futex has a second revision called FUTEX2. FUTEX2 supports `futex_waitv` to yield to the kernel until one of multiple memory addresses has changed [13].

2.2.3 Scheduler and Placement

Linux has used the Completely Fair Scheduler (CFS) as its default scheduling policy since kernel 2.6.23 in 2007, allocating computation time to tasks using a range of heuristics and tunable parameters [14]. Since kernel 6.6, EEVDF (Earliest Eligible Virtual Deadline First) has replaced CFS, substituting the heuristic approach with a more algorithmic one and improving tail latency in workloads with many runnable tasks competing for CPU time [15].

Scheduling classes sit above the scheduling policy and let the kernel prioritise tasks by category before applying the policy within each class [16]. The most common classes are `SCHED_OTHER` for normal applications, `SCHED_FIFO` for real-time applications and `SCHED_IDLE` for background tasks that should only run when nothing else is runnable. Within `SCHED_OTHER`, niceness values allow users to further tune relative priority between applications.

Cgroups are a kernel mechanism for grouping processes and enforcing resource limits, including CPU share constraints [17]. *Slurm*, the standard workload manager for multi-node HPC clusters [17], [18], uses them widely. Slurm launches jobs with detailed placement configuration, including CPU pinning and governor settings. CPU affinity APIs allow users to bind a process to a specific logical CPU or set of CPUs and both Linux and Slurm expose binding constraints to pin application threads to chosen cores [16], [18], [19].

Two additional mechanisms directly influence cross-core communication latency. CPU frequency scaling governors control the processor clock speed by selecting a scaling policy [20]. *Sleep states (C-States)* are low-power idle modes the CPU enters when it has no work: deeper states consume less energy but require more cycles to exit, directly increasing the latency of the next wakeup [21]. Affinity, CPU frequency settings and C-State settings together influence the worst-case latency a blocking primitive can observe.

2.3 RDMA, InfiniBand and RoCE

Reducing the interrupt- and scheduler-induced latency overhead is most critical for low-latency hardware. This work focuses on *remote direct memory access (RDMA)* over InfiniBand hardware. InfiniBand is a high-performance interconnect standard built for the demands of large-scale distributed computing, combining ultra-low latency with high bandwidth to enable efficient communication between compute nodes. Its central mechanism is RDMA, which allows nodes to exchange data directly, bypassing the CPU and OS networking stack, thereby eliminating overhead that conventional protocols cannot avoid. As of the November 2025 TOP500 list, 276 of the world's 500 most powerful supercomputers, representing 55%, rely on InfiniBand [22], making it the dominant fabric for HPC.

Beyond traditional HPC clusters, the rise of large-scale LLM training workloads has substantially increased demand for high-performance interconnects in data centers. As in scientific supercomputing, distributed model training depends heavily on collective communication patterns such as all-reduce and remains highly sensitive to communication latency. Following NVIDIA's acquisition of Mellanox in 2020 [23], InfiniBand further strengthened its position in the emerging LLM cluster market. According to Dell'Oro [24], InfiniBand accounted for more than 80% of LLM back-end network switch sales in 2023. At the same time, Ethernet has recently gained significant momentum as RoCE-based solutions have matured. By the third quarter of 2025, Ethernet accounted for more than two thirds of LLM back-end switch sales according to Dell'Oro [24].

RDMA over Converged Ethernet *(RoCE)* extends the programming model to Ethernet-attached hardware. From the perspective of user space software, InfiniBand and RoCE are indistinguishable: user space software accesses both through the same *libibverbs* API. The underlying hardware drivers, most notably mlx4 for ConnectX-3 and mlx5 for ConnectX-4 and later generations, expose both InfiniBand and RoCE capabilities through the same user space interface [25]. For ConnectX-4 and newer hardware, RoCEv2 is fully first-class. mlx5 handles both transport modes identically at the verbs layer. A ConnectX adapter port can operate in either InfiniBand or Ethernet mode. In the latter case the driver registers RoCEv2 GIDs and the hardware itself handles UDP encapsulation, entirely transparent to the verbs layer above. As a result, DABP should function without modification on both InfiniBand- and RoCE-capable hardware.

The most prevalent asynchronous resource within RDMA is the *completion queue (CQ)*. It abstracts send and receive operations as a *completion*, also called a Completion Queue Entry. A completion queue entry represents a send or receive event and contains status, opcode, length and information such as the memory address of received message data. Figure 3 shows a diagram of a completion queue.

It is a bounded ring buffer that records the completion of data transfer operations. A *queue pair (QP)* is the hardware channel actually used to transfer data; each QP has a

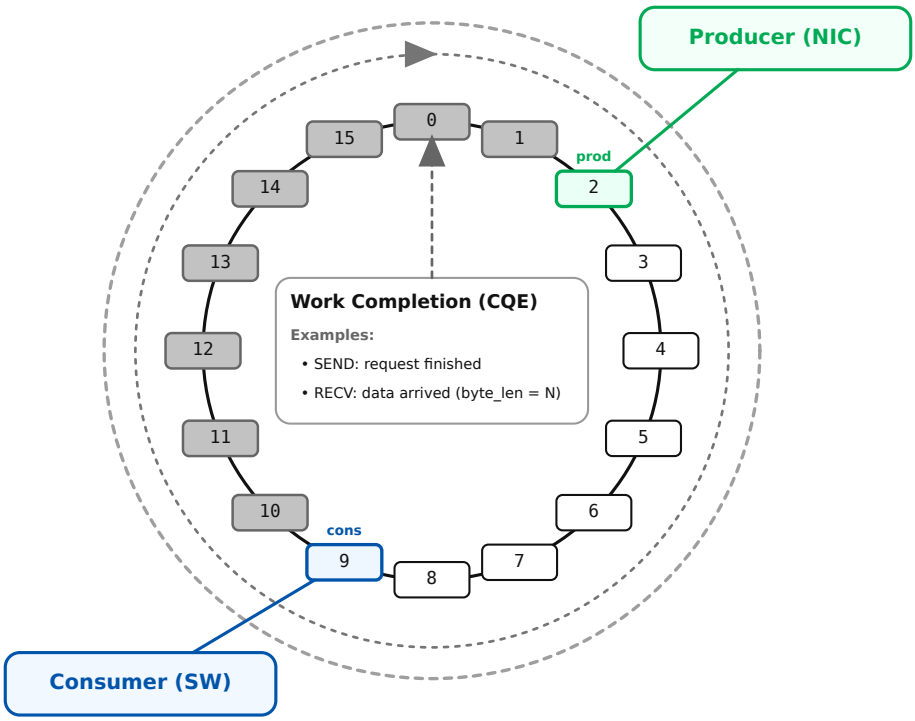


Figure 3: The Completion Queue

send queue and a receive queue and its completions are reported to the associated CQ. The producer — the NIC — writes Work Completions into the CQ and software reads elements from it. Additionally, applications can associate a completion channel with one or more completion queues and use it to register for interrupt-based notification (IRQs). The completion channel exposes an event file descriptor that user code can use to block for completion [26], [27], [28]. An application creates a CQ by issuing a request to the kernel. The kernel maps the completion queue data structure into the application's user space so accessing it does not require further kernel involvement. RDMA memory registration pins application buffers for DMA and associates them with memory regions and protection domains. Memory regions are the user-allocated receive and send buffers and protection domains can group multiple memory regions and enforce access control policies [29].

2.4 Existing application stack

Traditionally, RDMA has been used for HPC workloads. More recently, RDMA has also become increasingly relevant in datacenters, as emerging workloads such as LLMs, large-scale storage and distributed computing raise the demand for low-latency, high-throughput interconnects [30]. The existing software stack contains multiple layers and appears in Figure 4. The lowest layer is the Linux kernel. Device descriptors and memory handles reside in the Linux kernel. The `ib_core` subsystem manages devices, queue pairs, completion queues, memory regions and protection domains and owns the corresponding handles. It manages the hardware, programs hardware queues, handles interrupts and DMA, pins user memory and implements optional additional software stack layers such as IP over InfiniBand or the RDMA Connection Manager, which connect queues across multiple nodes. The layers above include the user space provider-specific drivers and abstractions in `rdma-core`. Some applications directly use `rdma-core` like Valkey and

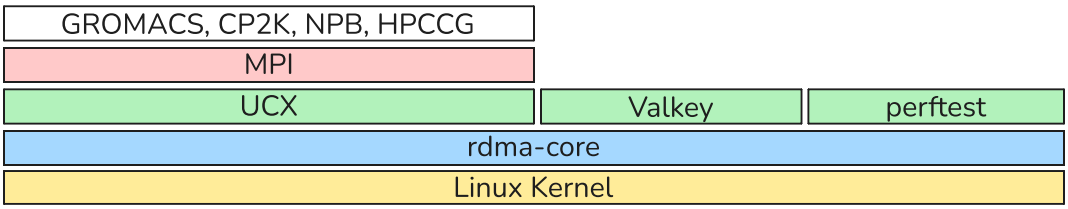


Figure 4: RDMA software stack layers and benchmark applications

the perftest suite. MPI applications like GROMACS, CP2K, NPB and HPCCG use UCX as an interface to `rdma-core`.

2.4.1 Verbs and Provider Layer

The lowest-level user space RDMA interface is `rdma-core`, which implements provider-specific code such as completion queue data structures and ring buffer allocation. It also contains `libibverbs` which generalizes over providers and defines common abstractions. Applications interact with RDMA resources through a "verb". One such verb is `ibv_poll_cq`, which reads and removes entries from the completion queue without waiting for new entries. Applications therefore use this mechanism to consume CQEs non-blockingly. `ibv_get_cq_event` provides the blocking, event-driven interface associated with completion channels [28], [31]. Completion queues can be resized at runtime which requires repinning and allocation or resizing of user buffers, but no common application uses this functionality.

A prerequisite of this work is the implementation of the *peeking* verb `ibv_peek_cq` for selected Mellanox provider families. It tests how many items are available within the completion queue without consuming entries from the queue. This verb was originally part of `ibverbs`, but the project removed it after deeming it unused. In the past there were efforts [32] to share RDMA resources between processes, but the verbs layer only implements sharing for stateless resources such as Memory Regions or Protection Domains.

2.4.2 UCX and Open MPI

Some applications call `ibverbs` directly, but in HPC the next common layer is UCX [33]. It is a high-performance communication framework that provides a unified API for low-latency, high-throughput data exchange across networks like InfiniBand, shared memory and GPUs. Like `rdma-core`, it provides primitives for one iteration of consuming busy-polling and blocking mechanisms that use event file descriptors for interrupts. UCX is the preferred modern backend for InfiniBand communication within Open MPI, one implementation of MPI, the dominant programming model for parallel computing in HPC. Applications use Open MPI to launch concurrent processes by splitting parallelizable work into multiple parts called a *rank*. Open MPI launches each rank as a separate process and often pins it to a specific CPU. Open MPI implements higher-level wait loops, progression mechanisms and waiting primitives using underlying layers such as shared memory or UCX. Applications across many domains, including computational fluid dynamics, finite element analysis and weather modelling, use these functions.

2.5 Related Work

The problem addressed by DABP belongs to the broader literature on end-system optimization for high-speed networks. Surveys of that area show that the bottleneck often

moves into the host software stack and that waiting, scheduling and I/O steering decisions can dominate latency and efficiency [34]. Multiple research efforts lay the foundation for DABP, but batched polling as a design concept is rarely mentioned directly. Many papers describe the same design space indirectly: low-latency communication, explicit progress and a runtime decision of when to poll continuously and when to sleep. Historically, von Eicken et al. in "Active Messages: A Mechanism for Integrated Communication and Computation" [35] framed communication latency as a combined systems and programming-model concern rather than as a pure device issue. "Integrating Polling, Interrupts, and Thread Management" [36] is an early direct precursor for DABP, treating waiting policy and thread scheduling as one problem instead of two unrelated layers and arguing that polling should be integrated with the thread scheduler so applications do not have to insert polling operations manually.

Münzberg's bachelor's thesis "Semantic CPU Yielding for Oversubscribed RDMA-Applications" [37] addresses the same oversubscription problem as DABP, but takes a kernel-based route. It builds on "CoRD: Converged RDMA Dataplane" [38], a dataplane that routes RDMA operations through the kernel instead of bypassing it and adds on-path logic directly into the Linux kernel RDMA driver. When a process polls a completion queue unsuccessfully, the kernel detects the stall and performs a targeted context switch to another process that has received data, without any application modifications. The requirement for CoRD's kernel modifications rules this approach out for standard HPC deployments where the environment prohibits kernel changes. DABP achieves the same goal entirely in user space.

2.5.1 Closest Predecessors

The closest matches for this thesis are "FastWake: Revisiting Host Network Stack for Interrupt-mode RDMA" [2] and "SmartInterrupts: A Node-Wide Asynchronous Message Progression Technique" [3]. FastWake describes two approaches to reducing average- and tail-latency in interrupt-based RDMA systems. One method steers hardware interrupts directly to the CPU core with the application that has requested the interrupt. FastWake modifies NIC priorities from the kernel, which makes it unsuitable for the user space requirement. The other method closer to DABP describes a per-core dispatcher thread that polls all the completion queues of the application threads on the same core. Li et al. [2] also employ a custom syscall for a kernel fast path to context switch to the thread with an incoming completion event. This syscall is a valid optimization, but can be omitted for this work. FastWake uses commodity RDMA hardware, Linux OS and unmodified applications, but requires kernel modifications, ruling it out for standard HPC deployments. DABP achieves the same goal entirely in user space, requires no kernel changes and places the poller dynamically rather than fixing one dispatcher per core. It also evaluates a broader set of benchmarks than perftest alone.

SmartInterrupts uses one or two helper processes on one system that poll completion queues and generate interrupts when events occur. The paper focuses directly on implementation within MPI. A single helper process may be associated with multiple MPI processes. They use inter-processor interrupts (IPIs) to switch back to the corresponding blocked user code, reducing the latency of RDMA hardware interrupts. IPIs are also easier to direct to specific CPU cores because the poller process knows which CPU has to wake up. This design implements interrupt steering similar to FastWake's first approach indirectly.

The paper designs benchmarks in MVAPICH, another MPI implementation, around overlapping communication with computation. They also evaluate their implementation using benchmarks from NPB. The NAS Parallel Benchmarks are a collection of benchmarking applications based on MPI. NPB benchmarks are also used by this thesis.

2.5.2 Hybrid polling and interrupt batching

Most related work on the drawbacks of polling and interrupts addresses the hybrid strategy of busy-polling briefly and falling back to blocking afterwards. Dovrolis et al. [39] propose a hybrid interrupt-polling scheme for network interfaces that dynamically switches between polling under high load and interrupts under low load, with the polling window adjusted based on measured packet arrival rates. For storage devices, Yang et al. [40] empirically show that polling delivers 2–4× lower latency than interrupt-driven completion handling, confirming that the same tradeoff applies beyond networking. DABP is a blocking approach and can therefore benefit from brief polling before it enters the slow path.

Mogul et al. [41] identify receive livelock, the condition where the system spends all its time processing interrupts and makes no application progress. They address it by batching incoming events and switching to polling under high load. NIC vendors adopted the same insight through interrupt coalescing: hardware delays raising an interrupt until either a configurable number of completions have accumulated or a timeout expires. Interrupt coalescing reduces the number of times the system must pay interrupt overhead. DABP could apply an analogous mechanism by only waking waiters once a configurable number of completion queue entries is available, reducing the frequency of wakeup primitives.

2.5.3 Reducing the downsides of busy-polling

Another nearby design point is shared-thread polling. Basu et al. [42] insert lightweight polling points into user code so that one shared thread can interleave useful work with frequent polling instead of dedicating a whole core permanently to the polling loop. The authors call these points interrupts, but in practice they are heuristically inserted jumps to custom program code injected into the binary by the compiler. This approach is relevant because it reduces wasted CPU time, but it requires compiler cooperation. The resource-checking logic of DABP could be moved into a “Compiler Interrupt”, but injecting jumps into arbitrary binaries at roughly constant time intervals is complicated. Another drawback is that applications must be compiled with the Compiler Interrupt toolchain, but source code can remain unchanged.

Relevant hardware-oriented work also explores how to make waiting cheaper without full busy-polling. Low-power wait instructions [43] and newer user-interrupt proposals such as xUI [44] aim to approach the latency of polling without dedicating a whole core to active spinning. These ideas depend on ISA or hardware changes and therefore sit outside the deployment model of this thesis. Nevertheless, special instructions could be another strategy to reduce energy consumption in DABP.

2.5.4 OS and Runtime Redesigns

Several other systems address the same latency-versus-efficiency tradeoff by redesigning the runtime or operating-system boundary more aggressively. “The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane” [45] and “ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks” [46] are

specialized operating systems (hybrid Linux + dataplane) designed to preserve low latency and high throughput under heavy load. “Shinjuku: Preemptive Scheduling for microsecond-scale Tail Latency” [47] reduces preemption overhead for microsecond-scale services, while “Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads” [48], “Caladan: Mitigating Interference at Microsecond Timescales” [49], “Arachne: Core-Aware Thread Management” [50] and “When Idling Is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone” [51] redesign core allocation, user-level scheduling, or idling policy to improve efficiency and tail latency. These systems are important context because they show how much performance can be gained once the scheduler or runtime controls placement and execution policy. That is also why they are not transparent drop-in comparisons for DABP.

3 Design

This chapter presents the final design of DABP. It begins by introducing the core mechanism and then derives the constraints that distinguish DABP from existing work. On this basis, the chapter addresses four central design questions: RDMA resource sharing, integration at the appropriate semantic wait site, synchronization between application and daemon and topology-aware daemon placement and thread count.

3.1 DABP overview

Daemon-Assisted Batched Polling (DABP) is a user space progression mechanism hybrid of busy polling and interrupts. Existing software stacks usually force an unfavorable choice between two extremes. Busy-polling gives very low readiness detection latency but continuously burns CPU time. Interrupts or event based waiting reduces CPU use but pays additional wakeup and software overhead. Asynchronous data transfer requires synchronization with hardware, yet the traditional approaches still show major disadvantages regarding latency and oversubscription. DABP combines the strengths of both traditional progression mechanisms and reduces their weaknesses.

The fundamental approach of DABP, shown in Figure 5, is to batch asynchronous resources and poll them centrally. Instead of driving progress itself while waiting for the readiness of a resource, a user application registers the resource with a global poller daemon and blocks. The daemon checks all registered CQs read-only and resumes the blocked user process when it observes readiness. Throughout this thesis, the user application is referred to as the *waiter* and the DABP daemon as the *waker*. In theory, a batched polling daemon can handle any kind of asynchronous resource. This work limits the mechanism to completion queues of RDMA because these queues are latency-critical. Fundamentally, this progression mechanism stays event based but its latency can be lower than interrupts.

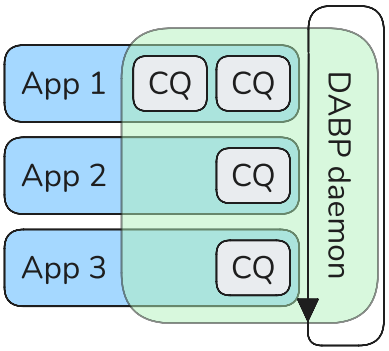


Figure 5: batched polling

The design separates the system into a slow control plane and a hot data plane. The system enters the control plane only when it registers or unregisters an asynchronous resource, so this path adds only a one-time setup cost per CQ. The data plane is the latency-critical path that the application uses while waiting for readiness. In that path the application enters a blocking wait, explicitly yields to the daemon and later resumes to consume the completion itself. Separating these two planes keeps expensive setup operations off the critical path and allows the data plane to remain as lean as possible.

DABP replaces the progression mechanisms interrupts or busy polling. To benefit both data-center and HPC workloads, DABP must replace both progression mechanisms. Busy polling always comes with drawbacks and many systems do not want to rely on a busy-poll-until-completion mechanism. Frameworks therefore commonly implement their own busy polling progression mechanism that handles multiple tasks in parallel. RDMA verbs do not expose a busy-poll-until-completion primitive, only a non-blocking consuming polling verb. To replace busy polling, this work inserts DABP directly at the framework-specific semantic wait site.

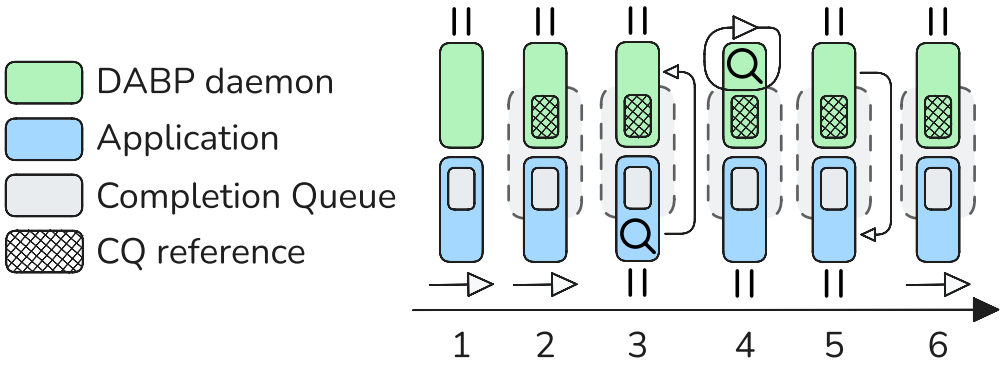


Figure 6: Baton-pass sequence between application and DABP daemon

DABP implements a two-way baton pass between two processes. Figure 6 shows this handoff in detail.

1. The application allocates a CQ resource
2. The application registers the CQ with the daemon and resumes normal execution
3. The application reaches a semantic wait site, blocks and transfers control flow to the daemon
4. The daemon busy-peeks the CQ
5. When the daemon observes readiness, it transfers control flow back to the application; if no other applications are currently waiting on asynchronous resources, it blocks
6. The application processes the completion and proceeds with normal program execution

The latency introduced by DABP depends on the locality between the user program and the daemon. The application blocks and the daemon wakes it when the completion arrives. Resuming the blocked process requires an inter-process signal and that signal is fastest when both processes run on the same CPU. Cross-core scenarios may require an IPI when the waiting CPU is idle and not already in the scheduler path. A drawback of same-core handoff is the need for one batched polling daemon per CPU core, which increases energy consumption and adds another process to each core. Because optimal strategies vary across workloads, this design keeps the placement of the DABP daemon and the number of daemon threads configurable.

The initial project name was “Coordinated Polling in User Space”, but batching asynchronous resources across applications captures the central architectural idea better. The remainder of this chapter explains the core parts of this architecture and alternatives. This includes read-only peeking, integration at semantic wait sites, explicit two-way handoff and topology-aware daemon placement.

3.2 Requirements and Constraints

Analysis of prior work and early experiments establishes a set of non-negotiable constraints for the design. The most important requirement is to keep the readiness detection latency of DABP close to busy polling and below interrupts.

At the same time, the mechanism must remain deployable in environments where modifying the Linux kernel, introducing a new runtime, or rewriting applications is not realistic. Guaranteeing transparent deployment in existing systems is key. The design should

therefore run on ordinary production-style HPC systems and datacenters. DABP must not rely on any special-purpose hardware beyond the target InfiniBand setup.

Deployment in existing software stacks imposes strict requirements on the system. The design must leave application-visible data paths and completion-consumption semantics unchanged. The application should still create resources through the existing stack, still consume completions itself and still remain correct when DABP is absent. That rules out designs that demand a new user-facing API.

Another major constraint is transparency under realistic cluster execution. HPC deployments often use restrained user permissions, pinned ranks, cgroups and tightly controlled CPU placement. The design cannot assume privileged scheduler control or custom kernel facilities.

Finally, the system must expose its important tradeoffs explicitly. DABP is not a single fixed policy but a design space for progression mechanisms with placement, blocking and scanning parameters. Those parameters should be tunable and clearly identified to avoid workload-specific overfitting. The implementation therefore must justify why concrete parameters exist at all, rather than hiding these tradeoffs behind fixed heuristics.

3.2.1 Gap to Existing Work

No existing work treats batched polling as a first-class replacement for traditional progression mechanisms. The closest predecessors each fall short on at least one of the constraints above. FastWake [2] requires changes to the kernel and limits itself to always having a poller per CPU. That design conflicts with the transparency requirements and with DABP's dynamic placement strategy. FastWake only evaluates its mechanisms using microbenchmarks and does not cover applications within HPC systems and datacenters.

SmartInterrupts [3] also takes a limited approach with only a few batched pollers per system and the design requires a custom kernel patch for IPIs. That approach focuses on MPI and therefore does not cover cloud applications. Approaches such as the user space scheduling framework Arachne [50] or kernel-bypass systems such as DPDK [52] require applications to adopt new APIs, which violates the constraints.

The focus of this work is to cover batched polling as a core mechanism and not as a byproduct. Existing work requires application or kernel changes that make these systems unsuitable for deployment in restricted-permission environments. DABP aims to provide a transparent user space layer that does not require kernel modification. For MPI-based applications it requires no source code changes, as this work places the integration inside OpenMPI and UCX. Applications outside the MPI stack require targeted source-level integration or can use a compatibility mode.

3.3 RDMA Resource Sharing and Access

With these constraints in place, the first concrete design consideration is the resource-sharing approach. RDMA uses another abstraction over send and receive buffers called the completion queue. The final design uses a global daemon process that has direct shared memory access to completion queues. Other approaches to central polling include using one batched polling thread per process, moving all RDMA resources into the DABP daemon and fully sharing CQs between processes.

3.3.1 Batched Polling Thread

The easiest prototype is to add a helper thread inside the same process and let it observe the application's CQ state directly. That variant is architecturally attractive because all relevant data structures already live in one address space and same-process wakeups can use cheaper private synchronization primitives. However, it does not solve the real coordination problem that motivates DABP. The polling thread cannot batch across multiple applications, does not align with common MPI-style process layouts and still leaves one polling helper per process rather than a shared readiness service. A major practical problem is scenarios with multiple applications per CPU, where this approach would still suffer from oversubscription. There are also many fork-based multi-process frameworks, such as MPI spawning one process per CPU, that are common in HPC development.

Another rejected family of designs keeps the polling helper but lets it consume completions directly because the upstream InfiniBand stack already implements the consuming polling verb. This exposes a semantic mismatch. The application still expects to own completion consumption, yet a consuming helper must re-publish or proxy those completions back to the user. This relay introduces additional synchronization, extra data movement and more fragile shared data structures in the path where DABP is trying to remove avoidable overhead. In practice this approach needs a separate shared lock-optimized copy of the completion queue. The key architectural lesson from this stage was that the daemon should observe readiness without consuming the completion itself. To solve this problem, this work uses a non-consuming peeking verb developed at the Chair of Operating Systems.

3.3.2 RDMA resource movement

The design requires a daemon-assisted version of batched polling. Introducing a separate process as the batched poller raises questions about who should own the RDMA resources. One approach is to move the full RDMA ownership into the daemon and proxy the entire verbs interface through it. This, however, does not work under the actual ownership constraints of RDMA software. Applications allocate ordinary memory, register that memory for DMA and then expect communication results to appear in those memory regions directly. If a separate daemon becomes the true owner of all RDMA objects, it also must become the owner of the associated communication state. That would either require retroactive sharing of already-allocated user buffers or force a copy path between daemon-owned memory and application memory. The former is not generally available in common operating systems and the latter defeats a central purpose of zero-copy RDMA.

3.3.3 Sharing stateful objects

Another approach is to share the full stateful RDMA objects themselves across processes. For stateless or simpler resources such as memory regions, export/import mechanisms exist, but the driver manages stateful objects like completion queues differently. This distinction is important because memory regions and protection-domain-style sharing mechanisms do not generalize to completion queues in a transparent way. Requiring general CQ or QP sharing across processes would pull the design toward kernel modifications that violate the transparency constraint.

The final design therefore adopts a narrower ownership split. The kernel-backed CQ object remains private to the application process. The design shares only the user space-visible

readiness state required for non-consuming observation. For the target providers, this state consists of the user space CQ ring buffer together with the read-side state needed to determine whether new completions are present. The daemon learns this information over the control plane, which exchanges the metadata needed to observe readiness on the hot path. The application still consumes completions through its original stack path, while the daemon only learns when waking the waiter is worthwhile. Once the design introduces this ownership split, the next question is where in the software stack to replace the waiting policy.

3.4 Selection of Semantic Wait Sites

DABP aims to replace the waiting policy at semantic wait sites, not one low-level read function. At first glance the obvious places to modify are the low-level verbs interfaces used for polling and interrupts. Both look attractive because they sit close to the hardware-facing abstraction and already have a clear connection to completion waiting. In practice, however, both options are poor places to express the semantic wait-site policy of DABP.

Replacing the interrupt-based blocking verb is straightforward because both it and DABP are event based approaches. This approach is relevant for interrupt-style datacenter integration but it does not cover the main busy polling progress loops used by MPI in the targeted HPC workloads. Replacing interrupts is also not just a drop-in change, because higher layers may interact with completion-channel file descriptors directly. Additionally, the interrupt path includes notify and acknowledge semantics. A notification tells the kernel to set up an event based primitive linked to an interrupt and the software must acknowledge the notification when the event arrives. Replacing the interrupt-based verb with DABP raises the question of how to handle these functions.

Conversely, replacing the consuming polling verb looks like the natural answer for busy polling, yet the polling verb itself is only one iteration of a broader policy loop. It answers the question “is something ready right now?” but not the higher-level question “should this call site keep spinning, switch to blocking, or integrate other progress sources first?”

The architecture inserts DABP at the semantic wait site rather than at the raw verbs primitive. The stack decides this wait policy progressively. `rdma-core` exposes the low-level readiness operations. UCX builds transport-level progress on top of them and MPI code decides how long to stay in a wait loop and which progress sources must remain responsive. The lower stack manages the CQ handle, so DABP needs low-level support in the verbs layer, but its decisive blocking policy belongs higher in the stack. In that sense, the design resembles runtime-managed `async/await` systems more than a pure primitive substitution: the low-level readiness primitive stays reusable, but the higher-level wait site decides the actual wait policy. C-based RDMA frameworks do not express semantic wait intent explicitly, so code analysis must identify it.

This work uses callsite profiling and flamegraph analysis of representative MPI workloads to identify the semantic wait sites, as detailed in Section 4.3.1. The result is that most investigated MPI benchmarks place the overwhelming majority of observed polling verb calls in two MPI-level callsites. This method can only suggest semantic wait sites, but code analysis reveals busy polling loops at both locations. The microbenchmark perftest and the datacenter representative Valkey both call the polling verb directly.

This evidence motivates the MPI layer as the insertion point for MPI workloads, but other stacks may require a different integration site. The architecture must therefore distinguish between low-level support and high-level usage of progression mechanisms. Direct-use stacks such as perftest or other verbs-facing applications use a lower-level DABP path, while for MPI workloads the insertion point lies above raw verbs polling at the semantic wait site.

This decision immediately creates another design consequence. In Open MPI, the progress engine handles more than just UCX. The relevant wait loops share a progression engine with other backends, so a fully blocking UCX wait cannot stall the application forever. A computation could depend on progress on other backends and unlimited blocking could stop program execution. Open MPI does not explicitly decide which backend to poll and most backends do not expose blocking mechanisms. Therefore the final design couples wait-site insertion with timeout-capped blocking and re-entry into the broader progression mechanism. Once blocking moves to the semantic wait sites, the synchronization between waiter and daemon becomes the next central design problem.

3.5 Synchronization and Scheduling

Because DABP involves switching processes, the synchronization model becomes central. The design is no longer only about who observes readiness. It is also about how control moves between the blocked application and the daemon.

One scheduling-oriented approach relies on asymmetric handoff. The observation is that the DABP daemon should not run while another process can still perform useful computation. Moving the daemon into a lower-priority scheduling class ensures that the scheduler directly returns control to the application when the daemon wakes it. This looks attractive because it keeps the daemon out of the way when the application is runnable, but it leaves the wakeup return path implicit. Waking a blocked waiter makes that waiter runnable; it does not guarantee that the scheduler will immediately transfer execution to it. That uncertainty becomes significant in the low-latency regime that DABP is trying to optimize. Scheduler yield hints reduced the resulting latency, but they did not eliminate it.

The scheduler-only design also fails a deployment requirement. Real HPC jobs frequently run under distinct cgroups and scheduler domains managed by tools such as Slurm. In that environment, assuming that a low-priority daemon will always “naturally” stay out of the way is not robust enough. Cgroups have precedence over scheduler classes. In the worst case, the daemon and application therefore compete for time slices on the same core even though the communication path itself needs only a small fraction of CPU time. The design therefore cannot rely purely on scheduler policy to enforce the intended handoff semantics.

DABP uses a baton-pass mechanism to move control flow from application to the daemon. It is an explicit blocking handoff in both directions. When the application reaches a semantic wait site, it blocks and the daemon starts polling. When the daemon observes a completion, it wakes the user application. If the application resides on the same core as the daemon, the daemon must also block immediately to avoid stealing CPU time from the application. If the daemon runs on another core, it blocks only if no other applications are currently waiting on asynchronous resources. The next section explains why topology-aware placement of DABP daemon threads is critical to this handoff pattern.

3.6 Daemon Pinning and Thread Count

Once DABP uses explicit two-way handoff, the placement between waiter and daemon in a given hardware topology becomes important because it directly influences wakeup latency. The architectural preference is clear: same-core placement offers the shortest handoff path, same-socket different-core placement is the next acceptable fallback and cross-NUMA placement is worst when low latency matters. Section 5.4 discusses the detailed measurements and the methodology behind these values. For the design, the relevant result is the ordering itself. This work focuses mainly on two cases of daemon pinning and thread counts. The first case aims to introduce as little runtime overhead as possible and therefore uses same-core handoff with one DABP daemon on each logical CPU. The second case is the other end of the spectrum, using one daemon thread to reduce overall energy consumption and to analyze topology effects on latency. The thesis evaluates placement as the main variable, while multi-daemon scaling beyond these cases remains future design space.

The ordering has different implications in different deployment domains. Pinned HPC jobs make stable placement policies realistic because each rank or thread is often already bound to a specific CPU set. This setting places the daemon with topology-aware intent and benefits from predictable locality. Datacenter-style workloads are less cooperative because oversubscription and migration between CPUs are more common. The design must tolerate cases where the runtime cannot maintain ideal same-core placement without excessive tracking overhead. Additionally, when CPUs are oversubscribed, multiple DABP daemons worsen oversubscription further, which makes a smaller thread count more attractive.

The same logic extends to the daemon thread count. A single daemon thread is an attractive setup because it maximizes batching and keeps coordination simple, but the architecture cannot assume that one poller will remain sufficient in all scenarios. If many completion queues are registered but rarely ready, the daemon may spend too much time scanning cold queues and lose the latency advantage it is meant to provide. The design therefore needs a path toward multiple daemon workers, CQ partitioning, or other scaling measures once the set of currently blocked processes grows large enough. At the same time, early measurements showed that one daemon core could still handle the tested scale without becoming saturated, so the scaling pressure is not an already observed failure mode. In contrast to a single batching daemon, same-core handoff requires one DABP daemon thread per CPU core. A drawback of employing multiple threads is that the daemon keeps each assigned CPU core at 100% utilization. This increases overall energy consumption when multiple daemon threads or application processes run across multiple CPUs.

DABP must stay flexible to any workload imbalances and could benefit from heuristics for deprioritizing unused completion queues. Due to time limitations, this work only evaluates daemon placement on cores, while keeping pinning adjustable.

4 Implementation

This chapter describes the DABP implementation. The central constraint on every design decision is a latency target in the low microseconds. The implementation uses Linux futexes as the synchronization primitive for blocking and waking both the daemon and the application, keeps the critical wakeup path minimal through methods such as a shared-memory active-wait bitmap that lets the daemon skip inactive queues and reserves unavoidable system calls for the futex handoff itself. The chapter proceeds from daemon internals to integration at each layer of the existing software stack and closes with the instrumentation that made correctness verification and semantic wait site selection tractable.

4.1 DABP Daemon Architecture

The implementation centers on a control plane and a data plane (see Section 3.1). The control plane handles registration and other infrequent lifecycle operations. The data plane covers the waiting and wakeup path that the system traverses on every blocking wait. Figure 7 shows how the final system realizes this split.

At the top in green, the control plane connects application-side registration code to the daemon's control thread. At the bottom in red, the data plane connects the waiting thread to the poller thread that performs daemon-assisted polling and wakeup. Shared memory in blue bridges both planes and provides the concrete communication channels used by the implementation.

The data plane is latency critical because it covers the transition from a semantic wait site into daemon-assisted polling and back out to the resumed waiter. The concrete work needed to enter the wait path, react to newly active resources, issue the wake and complete the two-way baton pass described in Section 3.5 determines its cost.

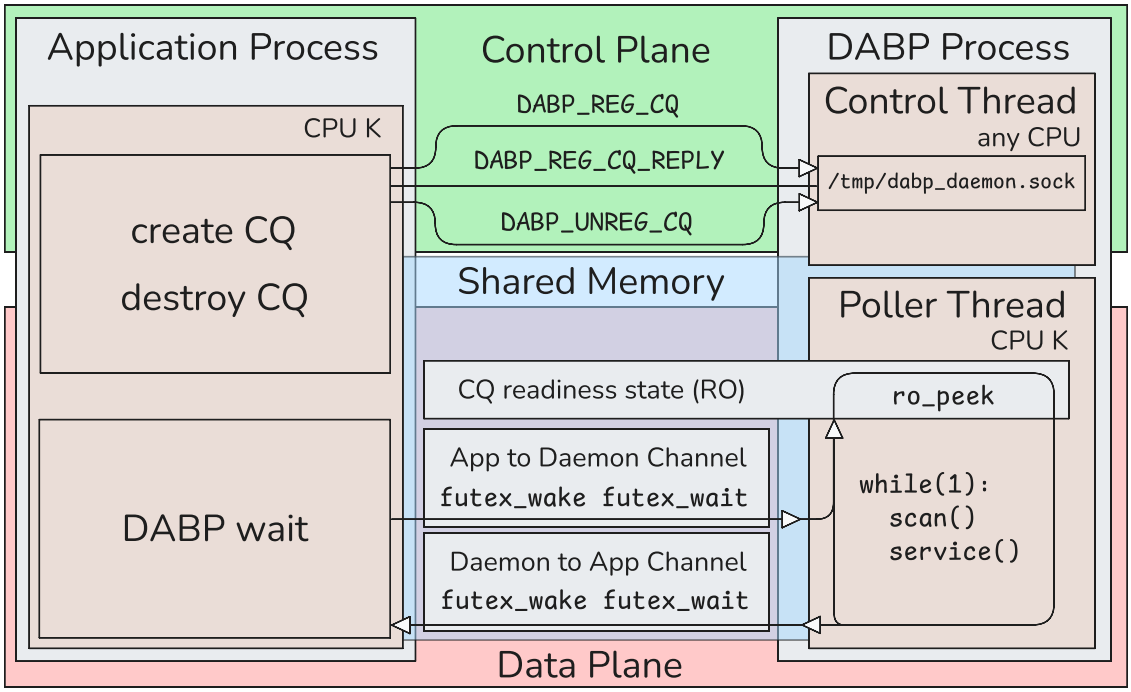


Figure 7: DABP two-plane architecture

4.1.1 Control Plane

The control plane handles all one-time per-CQ setup work so the data plane can remain lean. Its only endpoint is a unix domain socket at `/tmp/dabp_daemon.sock`, accessible to unprivileged jobs on HPC systems where system directories are not writable by users.

When the daemon starts, it spawns one control thread and one poller thread for each CPU in its affinity mask. The control thread blocks on the socket and handles all slow-path lifecycle operations. These include CQ registration, deregistration, shared memory setup and assignment of each CQ to a poller thread. After the control thread sends a successful registration reply, the registering thread is DABP-cooperating and may enter a DABP wait.

The wire protocol uses three message types. The registration request carries the registering thread's CPU hint and thread ID. The CQ ring buffer and per-CQ readiness metadata are passed as file descriptors rather than as addresses because both are backed by `memfd_create` anonymous handles that have no filesystem path and can only be transferred by passing the file descriptor itself over the unix domain socket. The ring maps into the daemon's address space as read-only, preventing accidental consumption and ensuring that inspecting completions never advances the consumer pointer, leaving consumption entirely to the application. The registration reply returns the assigned poller CPU ID and CQ slot index, along with two shared memory channel handles transferred the same way. The deregistration message carries the CPU ID and slot index and tears down the registration.

CPU assignment prefers same-CPU placement. The CPU hint carries the registering thread's pinned CPU and the control thread routes the CQ to the poller on that CPU when one exists, which gives the shortest possible wake path for a co-located waiter. If no poller runs on the hinted CPU, the control thread assigns the CQ to the poller with the fewest registered CQs. A same-NUMA fallback is not yet implemented, as it would require CPU topology probing at daemon startup and is left as future work.

The control plane assigns each CQ a fixed integer slot ID at registration time. This ID is simultaneously the CQ's index in the global registration table and its bit position in the active-wait bitmap on the assigned poller. Because the caller can only enter a DABP wait after a successful reply and the daemon commits the slot before sending that reply, the ID is stable for the entire CQ lifetime. The poller can therefore access any slot by index without acquiring a lock and the control thread can update metadata for a different slot concurrently without coordination.

4.1.2 Data Plane

The data plane is the latency-critical path traversed on every wait. Its job is to move a waiter into the daemon, allow the daemon to poll the active CQs without consuming completions and return control to the waiter with minimal scheduler overhead. Figure 7 shows this path on the bottom. The application enters a wait, uses the App-to-Daemon channel to wake the poller and receives the wake signal through the Daemon-to-App channel.

4.1.2.1 Yielding Strategy and Sleep States

The central challenge is that the daemon must poll continuously while applications are blocked but must yield as soon as an application becomes runnable on the same core or none remain blocked. A set of sleep states governs this behavior. Their transitions encode exactly when the daemon should poll and when it should yield. Figure 8 shows the state

machine with four sleep states on top and the Poll state below, each sleep state connected to Poll by a Wake transition.

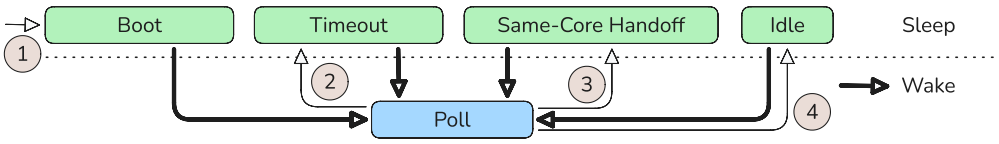


Figure 8: Daemon sleep states and Poll state

The boot state, reached via transition 1 in the figure, is entered at startup before any CQ has been registered and before the poller thread has begun its service loop. Blocking in this state avoids spinning before any work exists. The first entering waiter increments the App-to-Daemon channel futex word and moves the daemon into Poll.

The timeout state, reached via transition 2, is entered when the daemon observes that the number of blocked threads on its core is less than the number of registered threads. This means at least one application thread is runnable and should be preferred over the daemon, so the daemon sleeps. In practice this state arises when a DABP wait times out and the waiter leaves. Rather than tracking each waiter's individual deadline inside the daemon, the current implementation detects the aftermath through the blocked-count drop and yields at that point.

The same-core handoff state, reached via transition 3, is the fastest daemon path for same-core placement. The Daemon-to-App channel stores the waiter's CPU ID alongside the sequence counter, so the daemon can compare it against its own CPU immediately after issuing the wake. If they match, the daemon yields at once, giving the just-woken application thread the CPU without any further polling delay.

The idle state, reached via transition 4, is entered when all registered threads on the daemon core are blocked and no CQ slot has been marked active for `DABP_IDLE_SPINS_BEFORE_SLEEP` scan iterations, the first compile-time tunable encountered in this chapter. All `DABP_*` constants are compile-time defines rather than runtime parameters, keeping them off the timing-critical path entirely. The default of 8192 is large enough to avoid sleeping on very short inactive phases while still limiting wasted polling when no waiter is about to arrive. At that point no waiter is pending and further polling wastes CPU time, so the daemon blocks on the App-to-Daemon channel futex word until a waiter enters and wakes it.

All four sleep states share a Wake transition to Poll. An entering waiter increments the App-to-Daemon channel futex word, which moves the daemon into Poll as long as all registered threads on that core are blocked. A CQ deregistration can also trigger this transition: removing a CQ may reduce the registered count, which may bring it back down to the blocked count and require the daemon to resume polling.

On the client side, the waiting thread is sleeping on the Daemon-to-App channel. It exits that state in one of three ways: the daemon detected a completion and incremented the sequence counter, the caller's timeout expired, or a completion was already present before the client fully entered the wait and the client returns with EAGAIN.

4.1.2.2 Shared Memory Layout

The current implementation assumes that each CQ is used by exactly one application thread and the daemon at any time. This single-waiter constraint simplifies the synchronization model significantly, as it eliminates the multi-waiter races that complicate general-purpose condition variable implementations. Extending DABP to support multiple threads per CQ is left as future work.

Within this model, the Daemon-to-App channel uses a monotonically incrementing sequence counter for the futex word rather than a binary ready flag. A flag would require the waiter to reset it to zero before each new wait, because after being woken the flag is already set and `futex_wait` on the same value would return immediately. The counter avoids this: the client always snapshots the current counter value and sleeps on it and the daemon's next increment moves the value away from the snapshot without any reset step. The client snapshots the counter before setting the active-wait bit. If the daemon increments between those two steps the kernel observes the mismatch on `futex_wait` and returns immediately, so the system misses no completion. Wraparound is benign because `futex_wait` only requires that the current value differs from the snapshot, not that it is strictly greater.

Figure 9 shows the two shared-memory structures the daemon needs. The App-to-Daemon channel exists once per poller CPU, hence the alternative name per-poller channel. It stores the blocked-thread counter, which records how many cooperating threads are currently in a DABP wait. It also stores the registered-thread counter, which records the total number of DABP-cooperating threads assigned to this poller. Together these two counters implement the condition for transition 2 from Figure 8: when the registered count exceeds the blocked count, at least one application thread is still runnable and the daemon yields. The same structure contains the sleeping flag. An entering waiter reads this flag before issuing a wake syscall; if the daemon is already in Poll the flag is clear and the syscall is skipped entirely, saving a kernel entry on the common case path. The structure also contains the daemon's own futex word, on which the daemon blocks in every sleep state and which waiters increment to move it back into Poll. Finally, it contains the active-wait bitmap with one bit per registered CQ slot. The slot count `DABP_MAX_CQS_PER_CPU = 4096` is a practical upper bound chosen to cover realistic deployment sizes. Each bitmap word holds `DABP_WORD_BITS = 64` bits, matching the native machine word width so that the CTZ instruction used in the scan loop (see Section 4.1.2.5) operates on one word in a single instruction. The Daemon-to-App channel exists once per registered CQ, hence the alternative name per-CQ channel. It stores the sequence counter futex word that the daemon increments when it detects readiness and on which the waiter blocks. It also stores the waiter CPU ID and the poller CPU ID that are needed to evaluate same-CPU gating rules.

This separation lets multiple applications check the daemon's sleep state and wake it through a single shared channel, while the daemon wakes only the Daemon-to-App channel of the specific CQ that has new data rather than signaling all waiters.

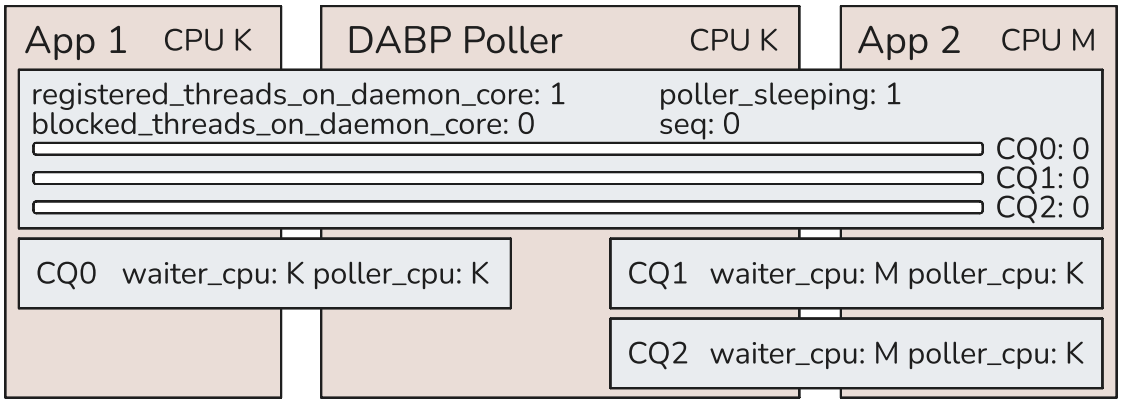


Figure 9: Shared-memory layout after CQ registration

The App-to-Daemon and Daemon-to-App channels occupy separate cache lines to prevent false sharing between them.

4.1.2.3 Entering a DABP Wait

Figure 10 shows the shared state after App 1 has entered a DABP wait on CQ 0, illustrating the same-core handoff scenario where App 1 and the daemon share the same CPU. The figure follows the setup state shown in Figure 9. CQ 0's active-wait bit is set, the blocked count equals the registered count (both 1) and `poller_sleeping` is 0 because the wake from the entering waiter has moved the daemon from Idle state to Poll. The daemon assigns CQ 1 and CQ 2 to the same poller but App 2 waits on them from CPU M. Their active-wait bits remain 0 and the daemon skips them in this scan cycle. The loop arrow in the center represents the daemon's active polling of CQ 0.

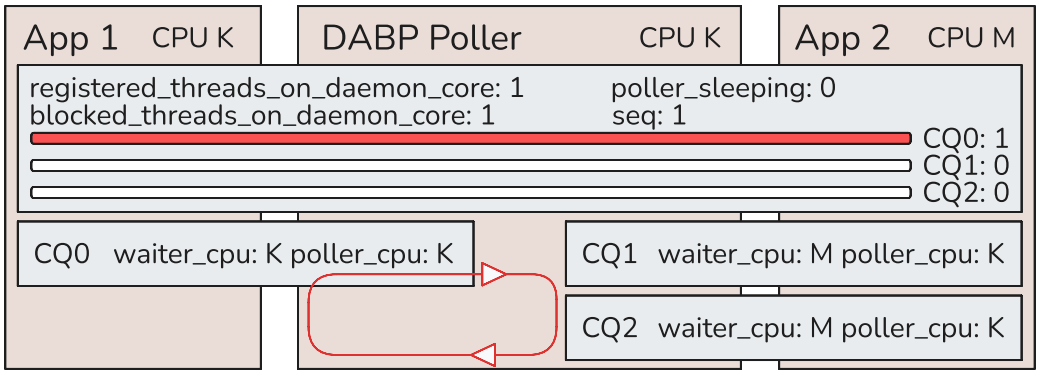


Figure 10: Shared memory state after a same-CPU waiter enters a DABP wait

To enter a wait, the client snapshots the per-CQ sequence counter, sets the active-wait bit and, if the sleeping flag is set and all registered threads on this poller are blocked, wakes the daemon into Poll. After that preparation, the client re-reads the per-CQ sequence counter. If the value already differs from the original snapshot, then the daemon detected the completion in the window between the active-wait bit set (or the daemon wake, if the daemon was sleeping) and this second counter read. In that case the client clears the active-wait bit, decrements the blocked counter on same-CPU paths and returns immediately, avoiding the futex syscall entirely.

If that fast exit does not trigger, the client issues the blocking call on the per-CQ word using the snapshot value and the caller's timeout. When control returns for any reason,

the client decrements the blocked counter on same-CPU paths and clears the active-wait bit. If the caller's timeout expired, the blocked count falls below the registered count on the next daemon iteration, driving the daemon into the Timeout state via transition 2.

The client always decrements the blocked counter and the daemon never does. This restriction arose from a double-decrement bug in an earlier version where a timeout and a daemon-detected completion coincided: the client decremented on the timeout exit path while the daemon also decremented on detecting readiness, driving the blocked count below its true value and causing the daemon to incorrectly observe the transition 2 condition from Figure 8 and yield even while waiters were still blocked.

For waiting on multiple CQs simultaneously, DABP provides a dedicated library function that passes an array of per-CQ sequence words to `futex_waitv` (see Section 2.2), which blocks until any one of them changes. One constraint arises here because `futex_wait` accepts a relative timeout and the kernel applies it after entry, whereas `futex_waitv` requires an absolute deadline. The client must therefore call `clock_gettime` before entering the wait to convert the relative timeout into an absolute deadline. On modern kernels `clock_gettime` is served by the *vDSO*, a kernel-provided shared library mapped into every process that exposes selected kernel functions without a kernel entry, but it is still an extra call on the critical path.

4.1.2.4 Idle Scheduler Class

Section 3.5 argued for explicit two-way futex handoff as the yielding mechanism rather than relying on `SCHED_IDLE` alone. Despite this, the implementation places the daemon thread in the `SCHED_IDLE` scheduler class in addition to the explicit handoff, because measurements show that `SCHED_IDLE` reduces handoff latency further even when explicit two-way handoff is already in use. Two-way handoff requires two system calls (wake followed by sleep) with a window between them in which both the daemon and the application are simultaneously runnable on the same CPU. The likely explanation is that the kernel's preference for yielding away from a `SCHED_IDLE` thread whenever a competing thread exists on the same CPU shrinks the window between the two syscalls.

`SCHED_IDLE` priority is however overridden by cgroup CPU-share constraints, which is why Section 3.5 argued against relying on it exclusively. When the daemon and the application reside in different cgroups, the scheduler allocates CPU time proportionally by cgroup weight regardless of scheduling class. A dedicated test confirmed this: a `SCHED_IDLE` process and a `SCHED_OTHER` process both pinned to the same CPU but placed in separate cgroups each received exactly 50% CPU time. Same-cgroup placement restores the intended behavior, so the current implementation benefits from it. An alternative is the kernel's cpu.idle cgroup setting, which marks an entire cgroup as idle-priority and would allow the daemon and application to reside in separate cgroups while still achieving the intended scheduling behavior. However, writing cpu.idle requires elevated privileges, making it unsuitable for unprivileged HPC deployments, so the current implementation does not use it.

4.1.2.5 Daemon Poll Loop

The daemon's inner scan loop iterates over the active-wait bitmap word by word. A single conditional branch skips zero words. For each non-zero word, the loop uses the count-trailing-zeros instruction to find the bit index of the lowest set bit, equivalently the slot

index of the next active CQ, in one hardware instruction. The Kernighan bit trick [53] then clears the lowest set bit in the local copy and the loop repeats until the word reaches zero.

Figure 11 illustrates one iteration with a 10-bit example. Bits 4, 7 and 8 are set. CTZ yields 4, identifying slot CQ_4 as the next active entry. Subtracting 1 flips bits 3 to 0. The AND with the original value clears bit 4 and leaves bits 7 and 8 for the next iteration. The operation modifies the local copy of the word only. The daemon does not touch the shared bitmap word. The daemon therefore revisits each active slot until the client clears the bit after returning from the wait.

bit index / CQ index:	9 8 7 6 5 4 3 2 1 0	
bits value:	0 1 1 0 0 1 **0 0 0 0**	CTZ = 4 → CQ_4 needs polling
bits - 1:	0 1 1 0 0 0 1 1 1 1	
bits & (bits-1):	0 1 1 0 0 0 0 0 0 0	ready for next iteration

Figure 11: CTZ iteration over the active-wait bitmap

The active-wait bitmap offers $\mathcal{O}(1)$ insertion through bitwise OR and $\mathcal{O}(1)$ removal through bitwise AND with the inverted mask. Iteration visits only active slots. Inactive words cost one zero-word load and one branch, with no additional per-slot overhead. Because the active-wait bitmap lives on the App-to-Daemon channel, any write to it — a bit set or clear by the client, or a counter update — invalidates that cache line for all readers including the daemon's scan loop. This is an inherent cost of co-locating these fields, a tradeoff the design accepts in exchange for the sequential scan access pattern it enables. If measurements reveal cache misses as a bottleneck, a future redesign could dedicate one cache line per active-wait slot, eliminating false sharing between CQ slots owned by different threads at the cost of a larger memory footprint and replacing the CTZ word scan with a SIMD-accelerated linear scan.

4.2 System Integration

The daemon and client library need entry points in the existing RDMA software stack. Figure 12 shows the standard layering before any DABP changes are applied. The Linux kernel forms the base, rdma-core sits directly above it, UCX builds on rdma-core and the evaluated MPI applications build on UCX. Valkey and perftest also rely on rdma-core, although they reach the verbs interface directly instead of passing through UCX.

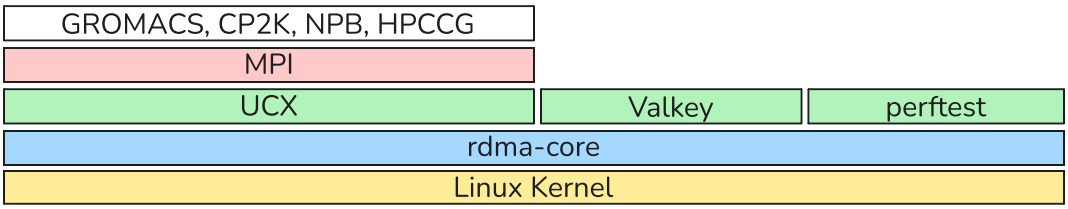


Figure 12: RDMA software stack before DABP integration

The central integration decision (see Section 3.4) is to introduce DABP only at semantic wait sites rather than use it as a blanket replacement for the standard polling verb. Across the software stack that verb serves two different purposes. Sometimes the stack uses it as a short non-blocking probe. At other times it marks the point where the software is genuinely ready to block. Replacing every invocation with a daemon-assisted wait would therefore change the meaning of many call sites that are intentionally non-blocking.

4.2.1 libibverbs and Provider Implementation

This work makes two sets of changes to rdma-core: edits to the mlx4 and mlx5 provider implementations and additions to the provider-agnostic libibverbs layer. Figure 13 shows the resulting stack.

The provider edits add a read-only readiness check that compares consumer and producer state in the CQ ring without advancing the consumer pointer, giving the daemon the non-consuming peek capability it needs. They also add `memfd_create`-based file descriptor creation so that the implementation can pass the ring mapping and per-CQ metadata to the daemon over the unix domain socket as described in Section 4.1.1. This FD handoff occurs at CQ creation time: the provider registers the queue with the daemon via the control plane protocol, stores the returned poller CPU ID and slot index in DABP-specific metadata attached to the CQ object and maps the shared-memory channels. At CQ destruction the provider sends the corresponding deregistration message to the daemon and releases those channels, keeping the daemon's view of registered queues synchronized with their lifetime without requiring explicit daemon-side lifecycle management.

Both providers also support a transparent compatibility mode enabled by setting the `DABP_WAIT_IN_POLL` environment variable. In that mode DABP replaces the standard polling verb, so applications using the plain ibverbs interface benefit from daemon-assisted waiting without any source changes. This is the configuration used for all perftest experiments in this thesis.

The implementation does not handle `ibv_resize_cq` while the daemon has a queue registered: a resize changes the ring memory layout the daemon observes through its read-only mapping and would silently corrupt that view. The implementation omits the deregister-resize-reregister path because none of the evaluated applications exercises it, leaving the behavior on a resize of a registered CQ undefined.

The libibverbs additions expose two provider-agnostic DABP verbs. `ibv_dabp_wait_cq` blocks until a single CQ has a completion available or a caller-supplied timeout expires. `ibv_dabp_wait_cq_any` waits on an array of CQs and returns as soon as any one of them becomes ready or the timeout expires. No other public ibverbs interfaces had to change.

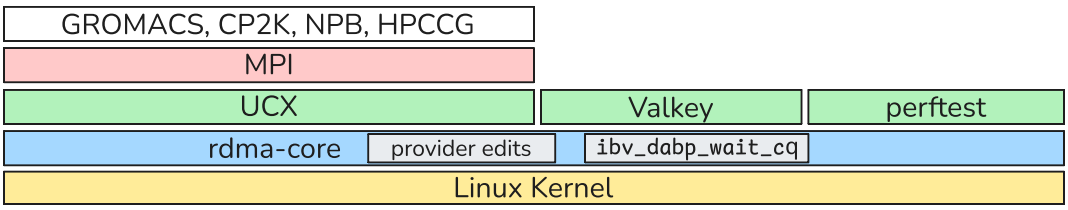


Figure 13: RDMA stack with provider-layer integration

4.2.2 UCX, OpenMPI and Valkey Integration

With the verbs layer in place, the remaining task is to invoke the new wait primitives at the upper-layer points where the software is semantically ready to block.

Because UCX exposes a similar interface to ibverbs, a single-pass poll function and an interrupt-based wait path, DABP passes through UCX the same way. The single-pass poll covers both the receive and transmit CQs, so `ucp_worker_dabp_wait`, called at the semantic wait site, calls `ibv_dabp_wait_cq_any` on both queues together to match that poll coverage.

One practical complication is that mlx5 in UCX defaults to the DEVX transport. DEVX is a UCX-internal direct device path that creates CQs through `ioctl` rather than through ibverbs, which bypasses the provider hooks added for DABP. Supporting DEVX requires a separate UCX-specific implementation and is beyond the scope of this work. For the experiments in this thesis, setting `UCX_TLS=rc_verbs` is sufficient to force mlx5 back onto the standard ibverbs path. That path is functionally equivalent for the workloads considered here. mlx4 and perftest are not affected because they do not use DEVX.

At the OpenMPI layer, callsite profiling (see Section 4.3.1) identified two wait sites that dominate polling activity: `MPI_Wait` and `MPI_Waitall`. Both first perform a short burst of roughly 256 busy-poll iterations and only then enter a timed daemon-assisted wait with a one-millisecond timeout (see Section 3.4). When that wait returns, control goes back to the existing progress loop. Because this work places the integration inside OpenMPI and UCX, none of the evaluated MPI applications required source changes.

The first version used a timeout of one second for `MPI_Wait`, which turned out to be too aggressive: a two-times oversubscribed two-node GROMACS run unexpectedly had the DABP timeout itself as the bottleneck: a single blocked `MPI_Wait` could leave the progress loop idle for as long as one second and stall the whole job. Both timeouts are now set to one millisecond.

Valkey required a similar integration at a different semantic wait site. Its original design uses AE, Valkey's internal event library, to multiplex RDMA and TCP events through event file descriptors and `epoll`. The DABP integration introduces a dedicated RDMA AE backend that calls `ibv_dabp_wait_cq` with a timeout. If no completion appears before that timeout expires, control returns to `epoll`, which continues to serve TCP and all other non-RDMA backends exactly as before. This preserves compatibility for mixed deployments while still giving RDMA completions the lower latency of daemon-assisted waiting.

Figure 14 shows the complete stack after all insertions. The Valkey AE backend and the OpenMPI `sync_wait` and `request_wait` hooks are the three semantic wait site integration points visible in the figure.

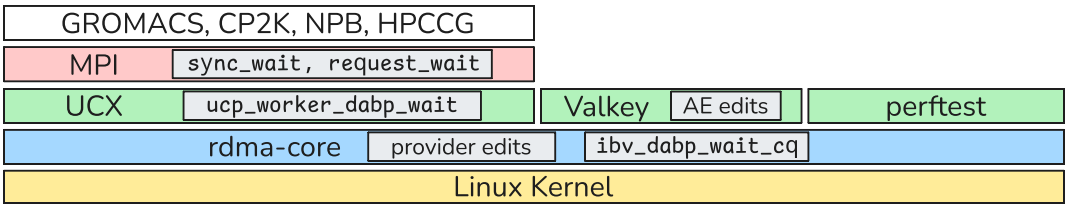


Figure 14: Complete RDMA stack with all DABP semantic wait-site insertions

4.3 Instrumentation

Both instrumentation layers described here are compile-time opt-ins guarded by pre-processor defines. They add zero overhead to a production build: all profiling hooks and debug counters compile out completely when disabled.

4.3.1 Callsite Profiling

The callsite evidence motivated the semantic wait site decisions discussed in Section 3.4. This section describes the two mechanisms used to collect it.

The first approach captures only the immediate caller of the polling verb on each invocation. The implementation accumulates the return address into an in-process per-callsite counter table. At process exit, a reporting hook emits the counters as raw addresses, resolved offline against the binary's symbol tables. This approach has no runtime dependencies, works in restricted HPC environments where sampling-based profilers are unavailable and produces reproducible results across runs. Its limitation is that it must be applied multiple times at different levels of the call stack to reconstruct the full call context.

The second approach addresses this by capturing full call-stack backtraces using libunwind. The implementation makes the polling verb a non-inline exported symbol so that the backtrace mechanism can reliably capture a backtrace on every invocation. Each call records the resulting program-counter sequence in an in-process hash table of unique call stacks with occurrence counts. At process exit, the implementation writes the table in flamegraph format, which the flamegraph tool renders as a flame graph showing the full call hierarchy rooted at the polling verb. Symbol resolution uses the dynamic linker's address-to-symbol facility, with C++ demangling applied to mangled symbols automatically.

Both approaches agree on the dominant callers: the two MPI-level wait hooks account for the large majority of polling verb calls across all tested applications. This directly motivates the two OpenMPI insertion points described in Section 4.2.2. The appendix collects the per-application execution traces for a selection of programs in Figure 26.

4.3.2 Debug Build Instrumentation

The debug build adds atomic event counters and ring-buffer traces to both shared-memory channels, recording per-daemon and per-CQ events with sequence numbers and return codes.

These traces made correctness analysis practical during development. The traces revealed the `futex_waitv` absolute-timeout constraint through an unexpectedly high timeout count in the counters, which pointed to waits returning too quickly. This turned out to be a wrong time format: the code was computing the absolute deadline incorrectly, causing `futex_waitv` to return immediately on every call.

5 Results and Evaluation

This chapter evaluates the implemented system across multiple benchmark suites, comparing latency, runtime, energy usage and throughput for busy polling, blocking and DABP-based configurations. The following questions guide the evaluation:

- Is DABP transparent enough to ensure existing behavior within multiple applications?
- How much overhead does DABP introduce to overall runtime (wall clock time), CPU-domain RAPL energy and latency compared to traditional busy-polling and interrupt-based setups?
- Which daemon placement strategies and thread counts are the best choices?
- How does DABP hold up in the presence of oversubscription?

The chapter starts by describing the chosen benchmarks in Section 5.1, the evaluation platforms in Section 5.2 and the measurement methodology in Section 5.3. The chapter then presents results of a self-written `futex_wake` latency benchmark in Section 5.4. The results have affected the DABP design in Section 3.6. Section 5.5 then evaluates DABP latency on InfiniBand communication through additional microbenchmarks.

Section 5.6 includes a large amount of different MPI-based benchmarks to cover functionality within HPC. These benchmarks cover DABP overhead both with and without oversubscription and evaluate scaling across multiple nodes. Section 5.7 evaluates a representative application for a multi-tenant datacenter setup. The chapter concludes by reviewing threats to validity in Section 5.8.

Table 1 shows an overview of important terms used in this chapter.

5.1 Chosen Benchmarks and Scenarios

The chosen microbenchmark for direct InfiniBand latency is the perftest suite [54]. The perftest suite is a collection of microbenchmarks to measure the performance of RDMA networks. Perftest uses the RDMA verbs directly to evaluate data transfer between a client and a server. This thesis uses `ib_send_lat`, a one-way send-latency microbenchmark.

MPI benchmarks contain programs across multiple domains. Each suite contributes candidates whose unmodified baseline runtime falls between one and two minutes. This range is kept small because each benchmark runs across multiple scenarios and total measurement time grows quickly. Beyond that runtime filter, this work applied no deeper workload-selection criteria and the final subset is effectively arbitrary.

The HPC benchmark tier includes GROMACS [55], CP2K [56], HPCCG [57] and NAS Parallel Benchmarks (NPB) [58]. GROMACS is a C++ software suite for high-performance molecular dynamics simulations. The chosen workloads are `BTC`, `CMET` and `MEM`, covering a binding

Term	Meaning
Baseline	Reference setup with unmodified source code
NUMA domain	Group of CPUs with faster local memory access and lower internal communication cost
Unpinned process	The scheduler can and will move the process to arbitrary CPU cores
K-times oversubscription	K processes pinned to the same CPU core

Table 1: Scenario and topology refresher

free-energy input, a protein-ligand equilibration input and a membrane-protein system from the official benchmark set. Large-CPU-count runs use the `RIB` input (2-million-atom ribosome-in-water simulation) from the official Max Planck benchmark set [59]. CP2K is a quantum-chemistry and solid-state physics package, written in Fortran, for atomistic simulations of solid, liquid, molecular, periodic, material, crystal and biological systems. The chosen input datasets include `H2O-64` and `H2O-GGA` from the official CP2K v2024.1 benchmark inputs [60], representing a 64-water-molecule DFT benchmark and a water benchmark with a GGA-based electronic-structure setup. HPCCG is a C++ benchmark based on the conjugate gradient method, an iterative algorithm for solving certain systems of linear equations numerically. Input parameters only include three integers that define the local three-dimensional grid dimensions. NAS Parallel Benchmarks (NPB) are a suite of programs developed by the NASA Advanced Supercomputing Division. They contain workloads ranging from embarrassingly parallel programs to communication- and memory-bandwidth-bound tests. The chosen testsets are class `C` (medium-sized), `BT`, `LU` and `SP`. These benchmarks are directly part of NPB. All benchmarks target parallel supercomputer performance evaluation.

The datacenter representative benchmark is Valkey [61]. Valkey is an in-memory NoSQL data store and the Redis [62] fork that remained under a permissive license following Redis's license change [63]. Valkey acts as a high-performance database, cache and message broker. The project contains `valkey-benchmark`, a benchmarking tool for measuring different characteristics of the server. The chosen workloads are `GET`, `SET`, `PING_BULK` and `PING_INLINE` because they require as little CPU computation as possible and are limited by communication throughput. `GET` reads one key, `SET` writes one key, `PING_BULK` measures pipelined `PING` requests and `PING_INLINE` measures the inline form of the same command.

5.1.1 Scenarios

A focus of this chapter is to evaluate different daemon placement strategies and daemon thread counts. These scenarios depend on the CPU topology. To analyse the placement effects, each application is pinned to exact CPU cores unless stated otherwise. Table 2 summarizes the recurring scenario abbreviations used throughout the result tables. The table defines the placement between the DABP daemon and the measured application. The same logical core scenario spawns one daemon thread per benchmark CPU, while all other DABP scenarios use exactly one daemon thread. This contrast captures the two ends of the thread count spectrum discussed in Section 3.6.

5.2 Evaluation Platforms

Every benchmark is run on two platforms. The first platform is Taurus, the remnant of an old HPC system of TU Dresden. The second platform is barnard, an HPC cluster of TU Dresden. Both systems use slurm for node allocation.

Taurus is a cluster of up to four identical machines. Each machine has an Intel Xeon CPU E5-2680 v3 running at a maximum CPU frequency of 2.50 GHz. There are two NUMA sockets, 12 cores per socket and 2 threads per core resulting in 48 logical cores. Each system has 64 GB of DDR4 RAM installed. The machines connect to both Ethernet and InfiniBand using a Mellanox Technologies MT27500 Family [ConnectX-3] network card. Debian 13 (kernel 6.12.63+deb13-amd64) runs on each machine.

Abbrev.	Scenario	Topology and placement
`Poll`	Busy polling	Pinned baseline, no DABP daemon
`Intr`	Interrupts	Pinned baseline, no DABP daemon
`Intr-P`	Pinned interrupts	Valkey only; pinned, no DABP daemon
`Intr-U`	Unpinned interrupts	Valkey only; scheduler may migrate the process
`SC`	Same logical core	Daemon and application share one logical CPU
`HT`	Hyperthread sibling	Same physical core, different logical CPUs
`SN`	Same NUMA	Different cores in the same NUMA domain
`XN`	Cross NUMA	Different NUMA domains on the same socket
`XS`	Cross socket	Different sockets

Table 2: Scenario abbreviations used throughout the evaluation chapter

Barnard is a general-purpose CPU cluster of 720 nodes by Atos/Eviden at TU Dresden's Center for Information Services and High Performance Computing, intended for highly parallel, data-intensive and compute-intensive HPC applications [64]. Each machine has an Intel Xeon CPU Platinum 8470 running at a maximum CPU frequency of 2.00 GHz. There are two sockets, 52 cores per socket and 2 threads per core resulting in 208 logical cores. The platform exposes four NUMA domains per socket, resulting in eight NUMA domains per system. Each system has 512 GB of RAM installed. The nodes connect to both Ethernet and InfiniBand using a Mellanox Technologies MT28908 Family [ConnectX-6] network card. Red Hat Enterprise Linux 9.6 (kernel 5.14.0-570.49.1.el9_6.x86_64) runs on each node.

5.3 Measurement Methodology

The configuration ensures reproducibility for each benchmark. The Nix package manager [65] orchestrates the build process. Nix stores build outputs in unique paths derived from their complete dependency graph, which makes builds reproducible and allows multiple package variants to coexist safely. The build process containerizes the resulting binaries and distributes them to all nodes so benchmark runs are reproducible too.

UCX and Open MPI are configured to always use RDMA for cross rank communication. This affects the benchmark by disabling shared memory, which is the default backend and faster than RDMA. The configuration is necessary because this work does not implement batched polling for shared memory. Shared memory is a non-blocking backend within Open MPI and using it as the main communication method requires constant polling. The DABP implementation assumes that RDMA is the main communication path and serves other backends only after a 1 ms timeout. To make measurements comparable, the configuration disables shared memory for the baseline as well. This influences baseline benchmarks negatively.

Intel Running Average Power Limit (RAPL) counters [66] are model-specific registers that expose accumulated energy usage for hardware power domains such as the CPU package, cores and DRAM. For every node used in relevant benchmarks, the measurement reads these counters monotonically for all CPU packages. After the run the system collects and sums all values across nodes.

Every benchmark allocates nodes exclusively. This ensures that other users' workloads do not influence measurements.

5.4 Futex Wake-Latency Microbenchmark

The microbenchmark evidence for futex wake latency influenced the development of DABP. The goal of this setup is to measure the optimal futex-based context switch strategy. Figure 15 shows the exact control flow. The benchmark uses the two-way futex handoff approach to synchronize the two processes. Depending on the scenario, the benchmark pins the processes to specific CPU cores. Each iteration performs the same handoff between waiter and waker processes.

The measured hot path includes the `futex_wake` syscall and the `futex_wait` syscall that yields back through the scheduler. The analysis calculates latency by recording timestamps in the waker before calling `futex_wake` and in the waiter after returning from `futex_wait`. The timestamps are determined using clock_gettime(CLOCK_MONOTONIC, ...), which on Linux is usually served through the vDSO, avoiding a full kernel entry. The vDSO reads kernel-maintained timekeeping data and computes a monotonic nanosecond timestamp from the active hardware clock source. A common source is the CPU Time Stamp Counter (TSC) [67], a register that increments every cycle or at a constant rate, which the kernel scales into time units.

The benchmark logic counts only iterations where the futex explicitly woke the waiter. This avoids spurious fast-path wakes that occur if the waker modifies the futex word before the waiter even starts waiting, resulting in `EAGAIN`. The benchmark runs for 30 s for each scenario and discards the first 10 000 iterations for cache warm-up.

Figure 16 shows the cumulative distribution function of the result latencies. Same logical CPU handoff has the lowest latency, followed by the HT-sibling, same NUMA, same socket and cross-socket placement. The hyperthread-sibling scenario shows a broader latency distribution. The waker's presence on the sibling hyperthread shares physical execution resources with the waiter. Because the scheduler sees two separate logical CPUs, it may not deschedule the waker as quickly as in the same-core case, which increases waiter latency. Latency samples for the same-core scenario appear in one small group directly

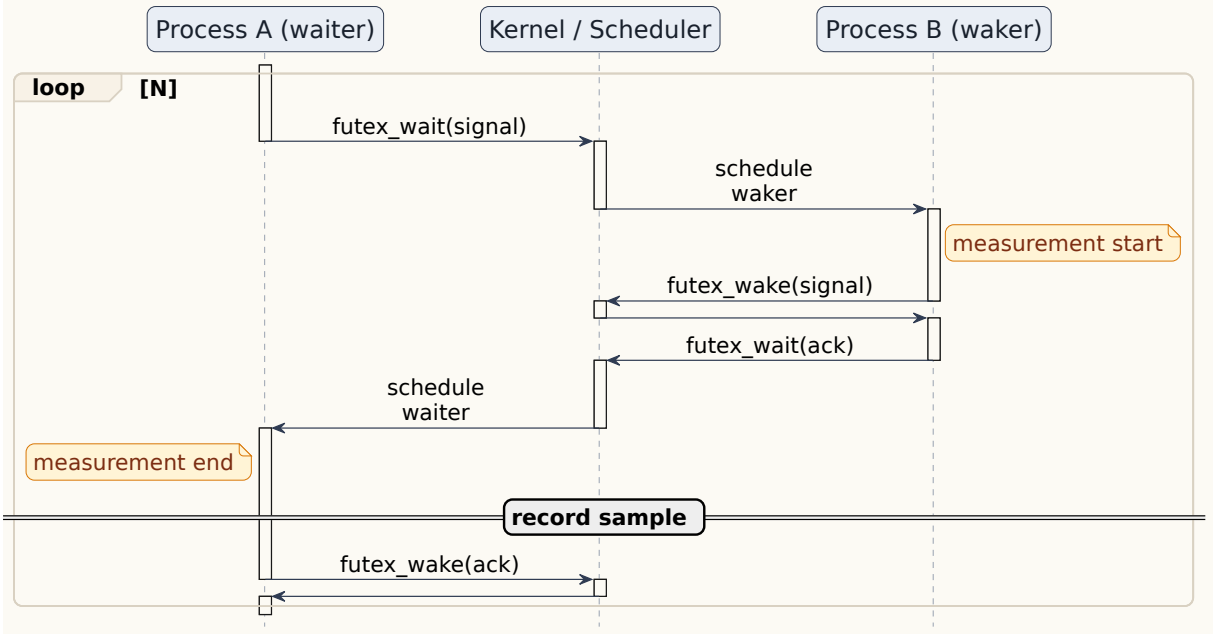


Figure 15: Sequence diagram of the futex wake-latency benchmark

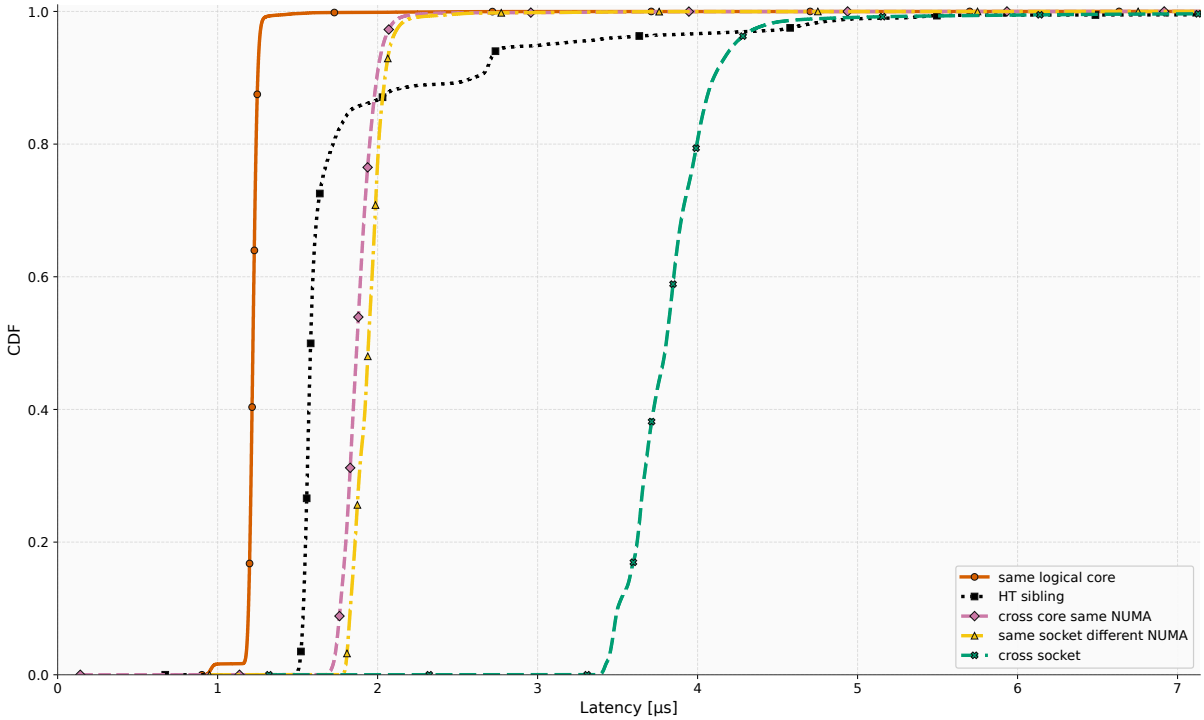


Figure 16: CDF of futex wake latency by placement regime (Taurus comparison in Appendix, p. 63)

at 1 µs and the majority at about 1.2 µs. The first group could represent the theoretical fastest path within the scheduler. Only a few samples reach this minimum because the scheduling path depends on many factors. C states likely also contribute to this variability.

Table 3 shows the main characteristics of the graph in microseconds. Average latency is up to three times higher when using cross-core communication.

The Linux scheduler takes many different paths depending on the current scheduling state and depends on numerous factors. One explanation for the latency distribution is that cross core communication gets slower depending on CPU distance in the topology.

The scheduler path depends on whether cores are idle, which idle state they occupy and how quickly they re-enter the scheduler. Futex wakeups use the normal Linux task wakeup path: `futex_wake` marks the waiter as runnable and the scheduler then dispatches the waiter. Linux idle threads avoid an interrupt only in a special polling-idle case. Otherwise an idle CPU sleeps until an interrupt arrives [68]. If the target CPU core is fully idle and not polling, waking the waiter therefore requires a reschedule IPI.

5.5 Perftest RDMA Microbenchmark

The second low-level latency benchmark uses perftest. The benchmark reports half the round-trip latency between a server and a client node as a one-way estimate. The measurements replace the non-blocking poll verb call with the blocking DABP verb. The perftest binary itself is unmodified; only the RDMA verb for completion polling changes. The measurements are the basis for detailed cumulative distribution function (CDF) plots, histograms and a percentile summary.

The benchmark runs across two nodes because the additional latency of the cross-node communication visualizes degradation in the same-core handoff path better. The

Scenario	min (µs)	p50 (µs)	p90 (µs)	p99 (µs)	p99.9 (µs)	avg (µs)
same logical core	0.901	1.22	1.25	1.29	2.147	1.221
HT sibling	0.672	1.581	2.599	5.224	7.215	1.821
cross core same NUMA	0.141	1.871	1.994	2.135	3.417	1.881
same socket different NUMA	0.949	1.942	2.046	2.216	3.361	1.945
cross socket	1.321	3.804	4.101	4.848	9.511	3.835

Table 3: Percentile summary of futex wake latency by placement regime

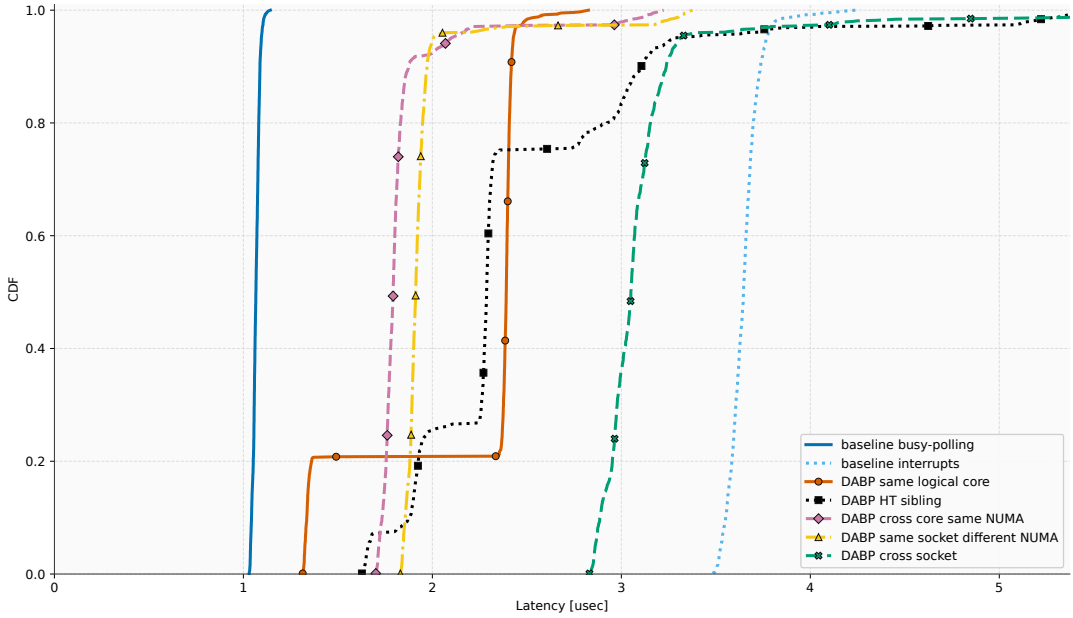


Figure 17: CDF of perftest latency (Taurus comparison in Appendix, p. 65)

benchmark pins both client and server to a CPU in the NUMA domain that holds the PCI connection to the NIC. A pre-run topology probe determines placement and all common scenarios from Table 2 use CPU pinning.

Figure 17 shows the CDF including both traditional busy-polling and interrupt-based approaches. The figure also plots the three common DABP scheduling and pinning strategies. DABP lies between these two approaches in all placement scenarios. DABP latency sits closer to the busy-polling baseline than to interrupts in all scenarios except cross-socket.

Figure 18 shows the distribution of measured latencies grouped into histograms. The traditional busy-polling approach has the majority of samples at 1 µs. The interrupt-based approach has a much higher latency, between 3 µs and 4 µs. The values are much more scattered for interrupts, indicating higher tail latencies. Within DABP, a common latency pattern emerges. The cross-core cases including same NUMA, different NUMA and different socket distribute their latencies around one value. The special cases of same logical core and HT sibling both show a bipartite pattern, likely reflecting different scheduling paths. A better-synchronized connection or a more optimized DABP implementation could reach the fast path more consistently. Another explanation is CPU sleep states. A

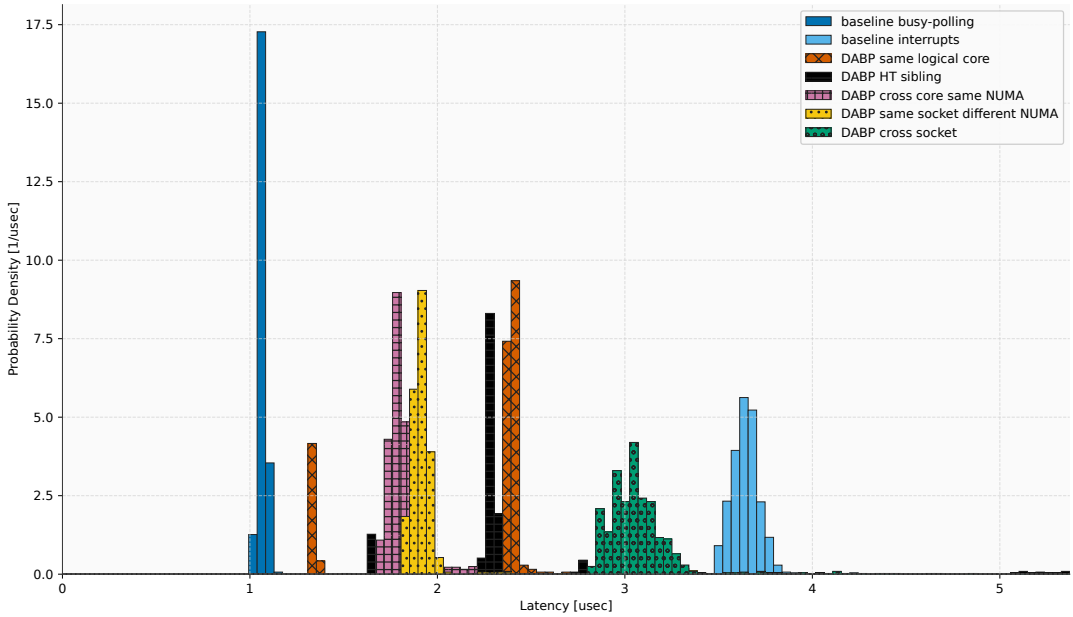


Figure 18: Histogram of perftest latency (Taurus comparison in Appendix, p. 67)

normal HPC setup cannot control sleep states because modifying them requires elevated permissions.

Metric	Poll (µs)	Intr (µs)	SC (µs)	HT (µs)	SN (µs)	XN (µs)	XS (µs)
min	1	2.92	1.1	1.17	1.24	1.12	1.08
p50	1.066	3.646	2.391	2.287	1.792	1.912	3.05
p90	1.088	3.737	2.418	3.106	1.875	1.972	3.238
p99	1.11	3.96	2.58	5.36	3.14	3.32	5.53
max	12.31	55.63	13.43	22.43	19.52	78.48	29.61
avg	1.07	3.65	2.16	2.45	1.84	1.95	3.1

Table 4: Perftest latency summary (Taurus comparison in Appendix, p. 69)

Table 4 shows values for all scenarios and highlights that, in contrast to Section 5.4, DABP does not achieve the best average latency for same logical core handoff. Cross-core handoff within the same NUMA domain or the same socket results in better average latencies, confirming that both are valid placement strategies.

Table 5 shows an overview of measured perftest values as percentages of the baseline busy-polling approach. On average, DABP reaches 172% to 289.7% of the baseline, while the traditional interrupt-based approach reaches up to 341.1%. P99 tail latency improves in some cases, but DABP is fundamentally scheduler-dependent and cannot guarantee consistent tail latency reduction.

Metric	Poll (%)	Intr (%)	SC (%)	HT (%)	SN (%)	XN (%)	XS (%)
Average	100	341.1	201.9	229	172	182.2	289.7
P99	100	356.8	232.4	482.9	282.9	299.1	498.2

Table 5: Perftest summary relative to busy polling

5.6 HPC Application Benchmarks

To evaluate HPC workloads, this section reviews different MPI-based applications. The setup still follows the strict CPU pinning of the common scenarios from Table 2. Unlike the microbenchmarks, these measurements use multiple CPUs for the benchmark application. To compare placement strategies cleanly, every benchmark uses at most one NUMA domain worth of CPUs minus one. Leaving one CPU free within the domain allows the same-NUMA daemon placement without overlapping the benchmark cores, while still using enough CPUs to generate meaningful communication load. All placement strategies use the same number of CPUs to keep scenarios comparable. The total CPUs are split symmetrically across both nodes. The concrete total CPU counts used for each benchmark in the standard two-node setup appear in Table 6.

All scenarios use exactly one DABP daemon except the same logical core setup. For the same logical core placement the system spawns one DABP daemon thread for each CPU the benchmark uses and pins it to that core. This evaluation excludes HT sibling measurements because they would require additional logic within DABP. Each measurement is run three times and plots include error bars to visualize fluctuation across runs. To increase InfiniBand communication and get closer to a setup that does not disable shared memory, each benchmark runs on two nodes.

5.6.1 Overhead without Oversubscription

Figure 19 shows the runtime for each benchmark in each scenario. DABP overhead stays close to the baseline and the figure highlights the advantage of the same logical core handoff scenario. The same logical core placement is often significantly closer to the busy-polling baseline, for example in NPB's BT. When the daemon shares the logical core with the benchmark, the CPU enters the daemon and continues polling instead of going idle. This can avoid the CPU falling into deeper sleep states and thus reduce overhead.

Table 7 shows that DABP same-core handoff introduces 1.8% to 3.6% overhead in total runtime over the optimal baseline busy-polling setup. Cross-core same-NUMA handoff stays below ten percent overhead in seven of nine cases. For all placements, DABP's overhead stays below ten percent in 29 of 36 cases.

Figure 20 shows the RAPL measurements for each benchmark in each scenario. The RAPL counters on barnard are inaccessible at the time of writing, so these diagrams only include measurements from Taurus.

Benchmark	Total CPUs
NPB BT	16
CP2K H2O-64	20
CP2K H2O-GGA	20
GROMACS BTC	20
GROMACS CMET	20
GROMACS MEM	20
HPCCG	8
NPB LU	16
NPB SP	16

Table 6: MPI total CPUs used per benchmark in the standard two-node setup

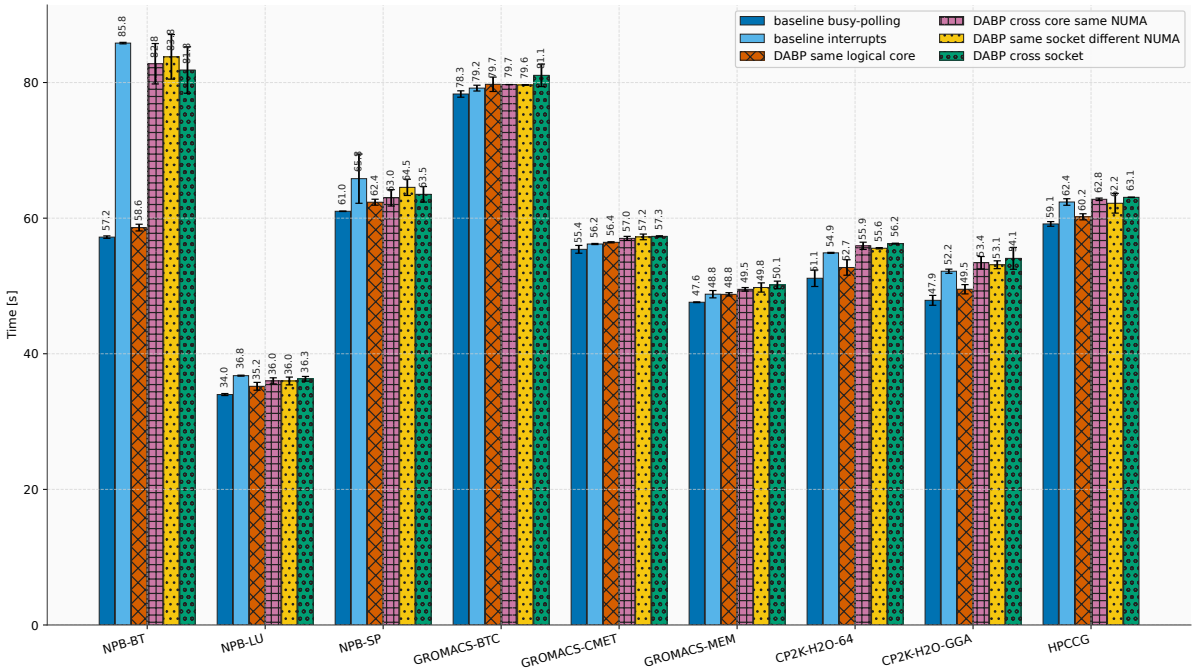


Figure 19: Runtime of MPI applications across daemon placement scenarios, two-node setup (Taurus comparison in Appendix, p. 70)

Overall CPU-domain RAPL energy stays close to the baseline busy-polling setup for most setups. Inspecting CPU usage during the benchmarks reveals that blocking approaches other than the same logical core handoff scenario reduce CPU usage. Early measurements of only one node showed that total CPU-domain RAPL energy can fall below the busy-polling baseline for all blocking approaches. After correcting the measurement and summing across all nodes, blocking approaches no longer undercut the baseline in total CPU-domain RAPL energy. Average power consumption is still slightly lower than for busy polling in many interrupt-based, same logical core and especially same-NUMA DABP measurements, which matches the observation that these scenarios keep CPUs slightly less busy. This diagram highlights that a cross-socket DABP handoff can significantly increase measured CPU-domain RAPL energy. The daemon may prevent the other socket from entering sleep states, which could explain this increase.

Benchmark	Poll (%)	Intr (%)	SC (%)	SN (%)	XN (%)	XS (%)
NPB-BT	100	150	102.5	144.7	146.5	143.1
NPB-LU	100	108.2	103.6	105.9	105.9	106.8
NPB-SP	100	107.9	102.2	103.2	105.8	104.1
GROMACS-BTC	100	101.1	101.8	101.8	101.7	103.5
GROMACS-CMET	100	101.4	101.9	102.9	103.3	103.5
GROMACS-MEM	100	102.5	102.4	104	104.6	105.4
CP2K-H2O-64	100	107.3	103.1	109.4	108.7	110
CP2K-H2O-GGA	100	109	103.4	111.6	111	112.9
HPCCG	100	105.5	101.8	106.1	105.2	106.6

Table 7: Total runtime of MPI benchmarks relative to busy polling

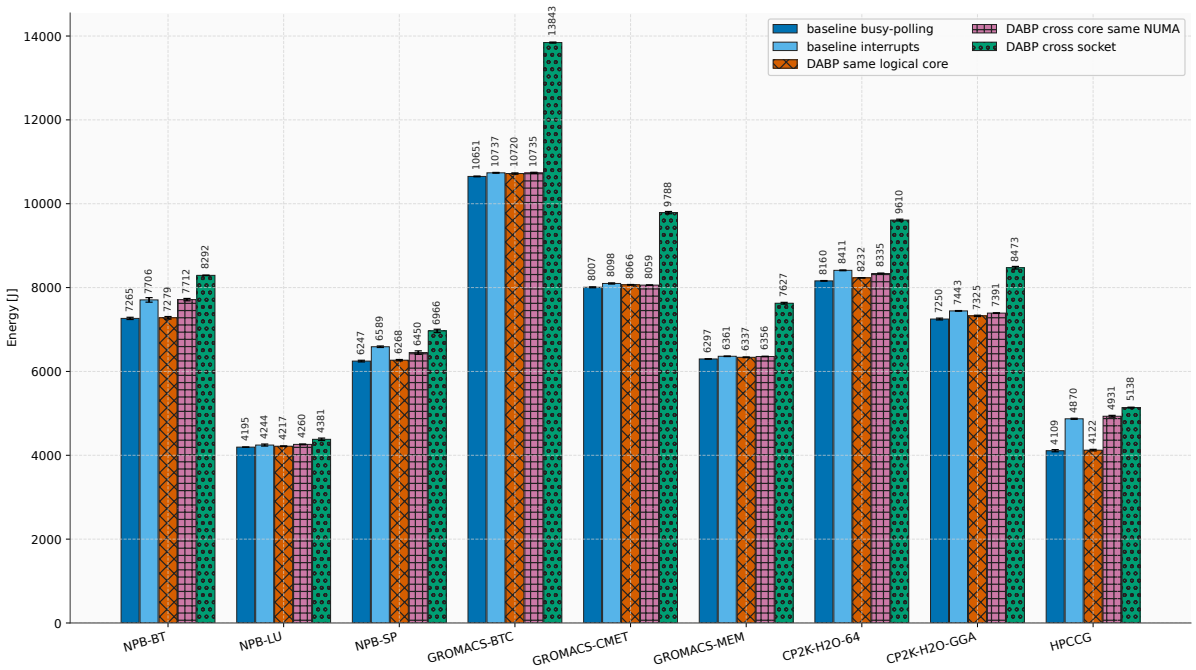


Figure 20: CPU-domain RAPL energy of MPI applications across daemon placement scenarios

Table 8 verifies that same logical core DABP introduces at most 1% overhead in measured CPU-domain RAPL energy. For 18 of 27 cases the measured CPU-domain RAPL energy overhead remains below ten percent. Optimal scheduling policies are workload-dependent. A same-core DABP policy always measures at least as much CPU-domain RAPL energy as the busy-polling baseline. If an MPI process blocks with DABP in this setup, it yields directly to the polling daemon, keeping the CPU at 100%.

5.6.2 Behavior under Oversubscription

The following benchmarks measure key figures under 2-times oversubscription. Each benchmark is run with double process count on the same CPU cores. CPU pinning stays the same as in the measurements without oversubscription, including for the interrupt-based baseline. Additional processes are pinned to the originally used cores, so if a benchmark had one process on CPU K, it now has two pinned to that core.

Benchmark	Poll (%)	Intr (%)	SC (%)	SN (%)	XS (%)
NPB-BT	100	106.1	100.2	106.2	114.1
NPB-LU	100	101.2	100.5	101.6	104.4
NPB-SP	100	105.5	100.3	103.3	111.5
GROMACS-BTC	100	100.8	100.6	100.8	130
GROMACS-CMET	100	101.1	100.7	100.6	122.2
GROMACS-MEM	100	101	100.6	100.9	121.1
CP2K-H2O-64	100	103.1	100.9	102.2	117.8
CP2K-H2O-GGA	100	102.7	101	101.9	116.9
HPCCG	100	118.5	100.3	120	125

Table 8: MPI CPU-domain RAPL energy relative to busy polling

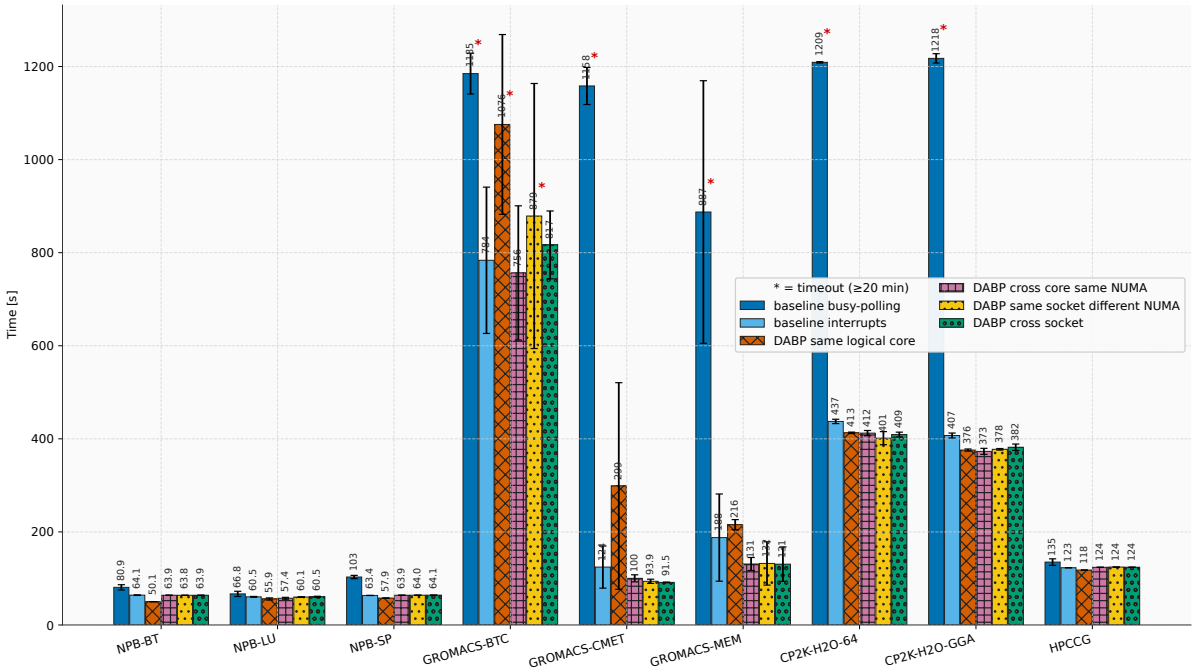


Figure 21: MPI runtime under 2-times oversubscription (Taurus comparison in Appendix, p. 72)

A key advantage of DABP over busy-polling is that it degrades less under oversubscription. When a thread busy-polls and another thread receives an RDMA completion, the waiting thread depends entirely on the Linux scheduler to get CPU time. The scheduler's context-switch interval is configurable but in practice far exceeds what low-latency RDMA communication requires.

Figure 21 shows the same measurements as Figure 19 under oversubscription. The baseline busy-polling approach degrades heavily and total runtime increases more than tenfold for certain workloads. The benchmark strategy therefore used a 20-minute timeout per run. A red star marks any bar where at least one repetition exceeded this timeout. This appears mostly in baseline busy-polling scenarios.

All blocking approaches perform considerably better or equally, depending on the benchmark. All three DABP placement policies undercut the traditional interrupt-based approach on average. Table 9 shows that the blocking approaches can be less than 10% of the busy-polling setup for GROMACS `CMET`. DABP results in better overall runtime for most workloads, but GROMACS `BTC` is an outlier. In this benchmark, the same logical core and same socket different NUMA handoff scenarios timed out in some cases, which is not observed on Taurus. Some benchmarks are less susceptible to degradation under oversubscription than others. Blocking scenarios for NPB and HPCCG yield only about ten percent improvement over the busy-polling baseline.

Table 10 shows the overall runtime as a percentage of the non-oversubscribed busy-polling values. Some workloads are more susceptible to degradation under oversubscription such as CP2K's `H2O-GGA` with up to 2 542.7% of the original runtime. Without the timeout, runtimes would be even longer. The blocking approaches do not reach the original runtime, which is plausible because the oversubscribed scenario doubles the process count and therefore also increases the total communication volume. Because the

Benchmark	Poll (%)	Intr (%)	SC (%)	SN (%)	XN (%)	XS (%)
NPB-BT	100	79.3	61.9	79	78.9	79
NPB-LU	100	90.5	83.6	85.9	89.9	90.6
NPB-SP	100	61.5	56.1	62	62.1	62.2
GROMACS-BTC	100	66.1	90.8	63.8	74.2	68.9
GROMACS-CMET	100	10.7	25.8	8.7	8.1	7.9
GROMACS-MEM	100	21.2	24.3	14.8	14.9	14.7
CP2K-H2O-64	100	36.2	34.2	34.1	33.2	33.8
CP2K-H2O-GGA	100	33.4	30.9	30.6	31	31.4
HPCCG	100	91	87.3	91.7	92.1	91.8

Table 9: MPI runtime under 2-times oversubscription, relative to busy polling

Benchmark	Poll (%)	Intr (%)	SC (%)	SN (%)	XN (%)	XS (%)
NPB-BT	141.4	74.7	85.4	77.2	76.2	78.1
NPB-LU	196.6	164.5	158.8	159.4	166.9	166.7
NPB-SP	168.9	96.4	92.8	101.5	99.2	100.9
GROMACS-BTC	1 513	989.7	1 348.7	949.2	1 103.4	1 007.5
GROMACS-CMET	2 090.6	221.4	529.2	175.8	164	159.7
GROMACS-MEM	1 864.4	384.9	442	264.7	265.6	260.6
CP2K-H2O-64	2 364.3	797	783.1	737.3	722.5	727.4
CP2K-H2O-GGA	2 542.7	780.4	759.1	698.3	710.5	706.4
HPCCG	228.5	197.2	196	197.4	200.2	196.7

Table 10: MPI runtime under 2-times oversubscription, relative to the non-oversubscribed case

setup pins all additional processes to the same cores as the originals, the scheduler has no free cores to migrate work to and cannot compensate for unbalanced communication patterns.

Figure 22 shows the same measurements as Figure 20 for oversubscription. The figure shows the same pattern as Figure 21 and confirms that a cross-socket DABP handoff can introduce a large increase in measured CPU-domain RAPL energy. Table 11 summarizes the diagrams using percentages, showing the same picture as the runtime analysis under oversubscription. Oversubscription affects workloads differently but blocking approaches remain superior.

Table 12 compares the values to the non-oversubscribed setup. For the blocking approaches, the measured CPU-domain RAPL energy lies between 87% and 351% of the non-oversubscribed application.

5.6.3 MPI Scaling Across Multiple Nodes

A core drawback of the evaluation is that most experiments use only a limited number of CPUs within a node. This section therefore adds a larger MPI benchmark that scales across multiple nodes and total rank counts. CPU pinning is still fixed but the only included DABP scenario is same logical core handoff. The evaluation uses this scenario because it has proven to achieve the lowest overhead on average and does not require CPU topology

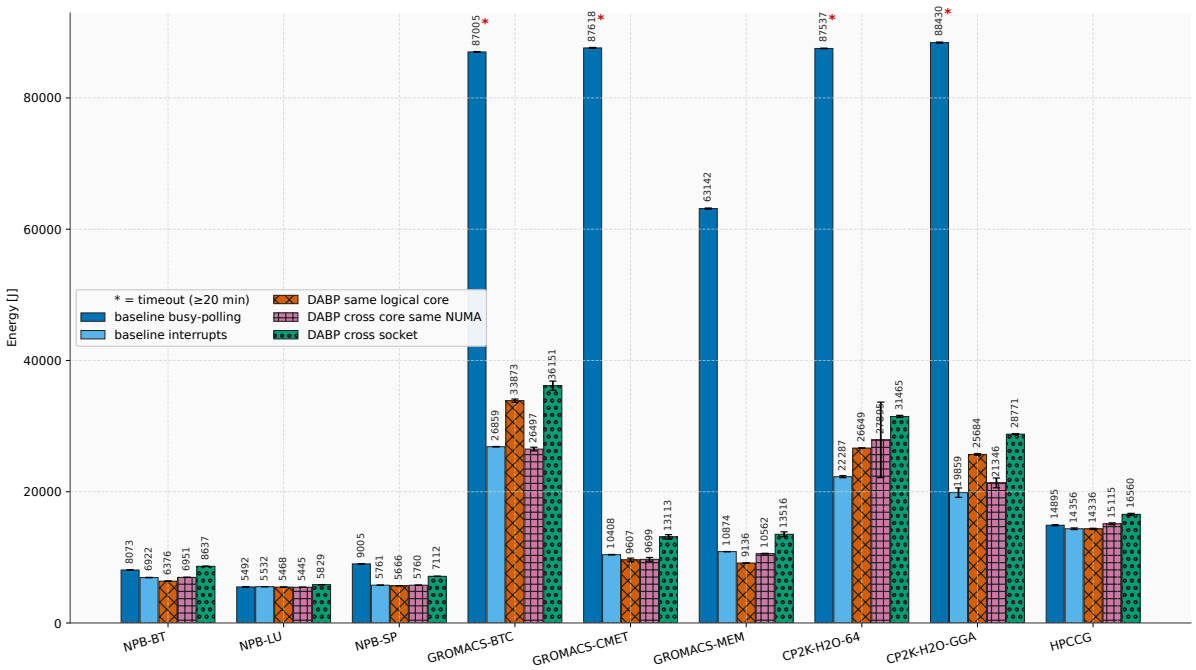


Figure 22: MPI CPU-domain RAPL energy under 2-times oversubscription

Benchmark	Poll (%)	Intr (%)	SC (%)	SN (%)	XS (%)
NPB-BT	100	85.7	79	86.1	107
NPB-LU	100	100.7	99.5	99.1	106.1
NPB-SP	100	64	62.9	64	79
GROMACS-BTC	100	30.9	38.9	30.5	41.6
GROMACS-CMET	100	11.9	11	11.1	15
GROMACS-MEM	100	17.2	14.5	16.7	21.4
CP2K-H2O-64	100	25.5	30.4	31.9	35.9
CP2K-H2O-GGA	100	22.5	29	24.1	32.5
HPCCG	100	96.4	96.2	101.5	111.2

Table 11: MPI CPU-domain RAPL energy under 2-times oversubscription, relative to busy polling

Benchmark	Poll (%)	Intr (%)	SC (%)	SN (%)	XS (%)
NPB-BT	111.1	89.8	87.6	90.1	104.2
NPB-LU	130.9	130.3	129.7	127.8	133.1
NPB-SP	144.2	87.4	90.4	89.3	102.1
GROMACS-BTC	816.9	250.2	316	246.8	261.1
GROMACS-CMET	1 094.3	128.5	119.1	120.4	134
GROMACS-MEM	1 002.7	170.9	144.2	166.2	177.2
CP2K-H2O-64	1 072.8	265	323.7	334.7	327.4
CP2K-H2O-GGA	1 219.8	266.8	350.7	288.8	339.5
HPCCG	362.5	294.8	347.8	306.5	322.3

Table 12: MPI CPU-domain RAPL energy under 2-times oversubscription, relative to the non-oversubscribed case

probing. The daemon runs on the same cores as the benchmark, eliminating topology considerations such as how to place the daemon within the NUMA topology.

Figure 23 shows the overall runtime for a 2-node setup using 200 CPUs, a 5-node setup using 400 CPUs and a 10-node setup using 400 CPUs total. The figure clearly shows that the baseline interrupt approach introduces overhead. Table 13 shows the results as percent of the baseline busy-polling approach. For 200 CPUs, the overhead of interrupts is an additional 46.7% total runtime. DABP can reduce this overhead to 19.7% which is less than half the overhead. This corresponds to the microbenchmark latency that was about half for DABP same core compared to interrupts. The same pattern holds for the 5- and 10-node setups with twice as many CPUs. DABP reduces the overhead of interrupts, but the reduction is smaller for the 400 CPUs case than for 200 CPUs.

This benchmark is already communication-bound, which limits how much additional scaling can reduce runtime. Doubling the CPU count only reduces total baseline runtime from 100.3 s to 86.1 s and 85.4 s. The benchmark does not scale well, which could affect the DABP results too.

A final observation is that the 5-node and 10-node measurements are very close. This shows that total runtime depends much more on the total CPU count than on the number of nodes. For this benchmark, adding more nodes does not affect runtime when the total CPU count stays constant.

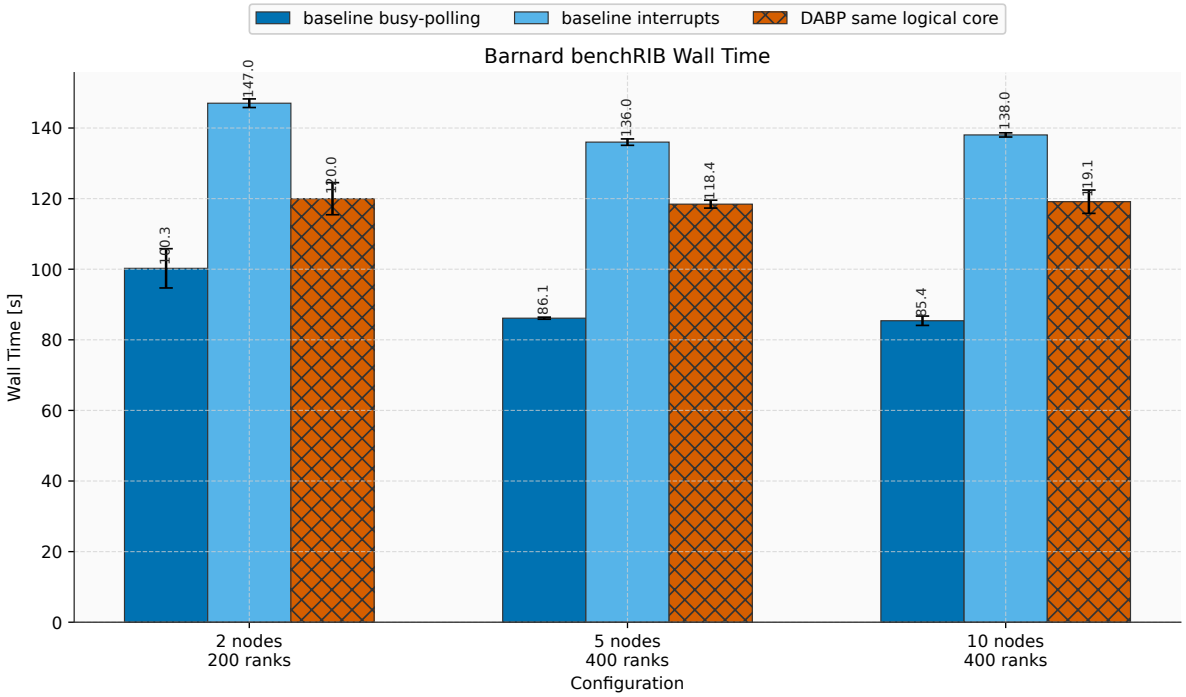


Figure 23: MPI runtime for the larger Barnard benchmark set

Nodes	Ranks	Poll (%)	Intr (%)	SC (%)
2	200	100	146.7	119.7
5	400	100	157.9	137.5
10	400	100	161.6	139.5

Table 13: Barnard runtime relative to busy polling for the larger benchmark set

5.7 Datacenter Workloads: Valkey

Currently, fully implemented open-source datacenter benchmark applications using native RDMA are rare. This section evaluates Valkey as the datacenter representative workload. This benchmark uses a two-node setup with one node as the server and one client node. Application placement follows the common scenarios from Table 2. The evaluation also includes unpinned cases because datacenter workloads are rarely pinned to specific cores. There are no other applications running while conducting the benchmarks. Each benchmark scenario runs for 400 000 iterations. Multi-tenant setups expect oversubscription, which means Valkey does not use busy-polling at all. The difference to the other benchmarks is that the baseline is now interrupts. All DABP scenarios use exactly one daemon thread per node. A drawback of the Valkey benchmarks is that the tool rounds all measurements to full microseconds and reports them in milliseconds, limiting precision for sub-10 µs latencies.

Figure 24 shows median and tail latencies of four Valkey benchmarks. In this scenario, Valkey acts as an in-memory key-value store supporting read and write operations. `GET`, `SET` and `PING` depend most on communication latency and require the least server-side computation. DABP consistently reduces latency from 0.031 ms to 0.007 ms compared to interrupts. DABP reduces tail latencies to ∼23% of the unpinned baseline as shown in Table 14. This is a direct consequence of replacing interrupt-based RDMA communication with batched polling and additionally replacing Linux event file descriptors with futexes.

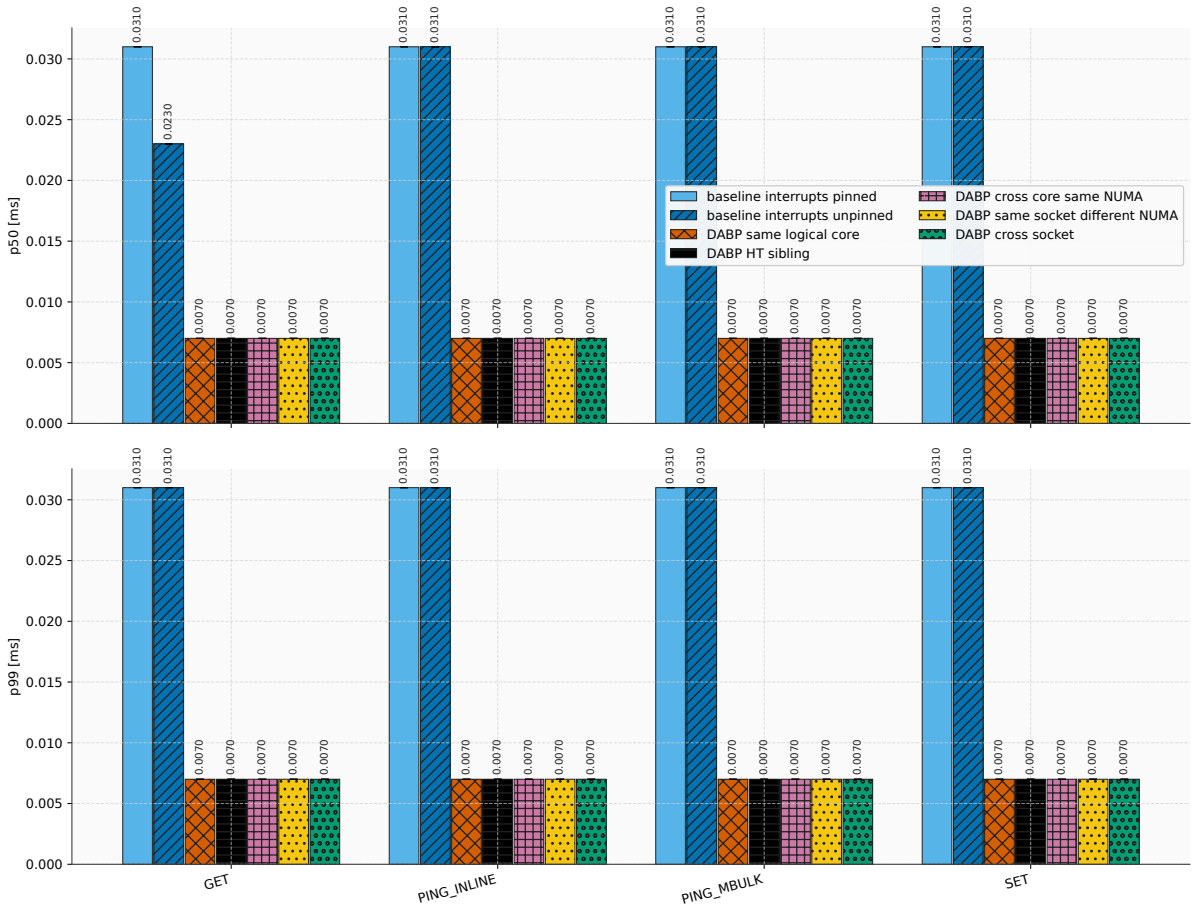


Figure 24: Valkey p50 and p99 latency (Taurus comparison in Appendix, p. 74)

Operation	Intr-P (%)	Intr-U (%)	SC (%)	HT (%)	SN (%)	XN (%)	XS (%)
P50 latency							
GET	134.8	100	30.4	30.4	30.4	30.4	30.4
PING inline	100	100	22.6	22.6	22.6	22.6	22.6
PING mbulk	100	100	22.6	22.6	22.6	22.6	22.6
SET	100	100	22.6	22.6	22.6	22.6	22.6
P99 latency							
GET	100	100	22.6	22.6	22.6	22.6	22.6
PING inline	100	100	22.6	22.6	22.6	22.6	22.6
PING mbulk	100	100	22.6	22.6	22.6	22.6	22.6
SET	100	100	22.6	22.6	22.6	22.6	22.6

Table 14: Valkey p50 and p99 latency relative to the unpinned interrupt baseline

Table 14 shows that Valkey exhibits little sensitivity to CPU pinning in these measurements. With DABP, median (P50) latency falls to between ∼23% and ∼30%, and P99 latency falls to ∼23%.

Figure 25 shows that the latency reduction does not directly translate into throughput gains. DABP pushes Valkey closer to the point where InfiniBand becomes the bottleneck and throughput depends heavily on the DABP placement strategy. Table 15 shows the measured values as percentages of the baseline unpinned approach. Cross-socket scenarios improve throughput to at most 110.3% and reduce throughput for GET by 0.6%. Same logical core DABP brings big throughput improvements up to 373.2%. The hyperthread sibling handoff approach is the second best strategy, offering up to 263.3% of the baseline throughput.

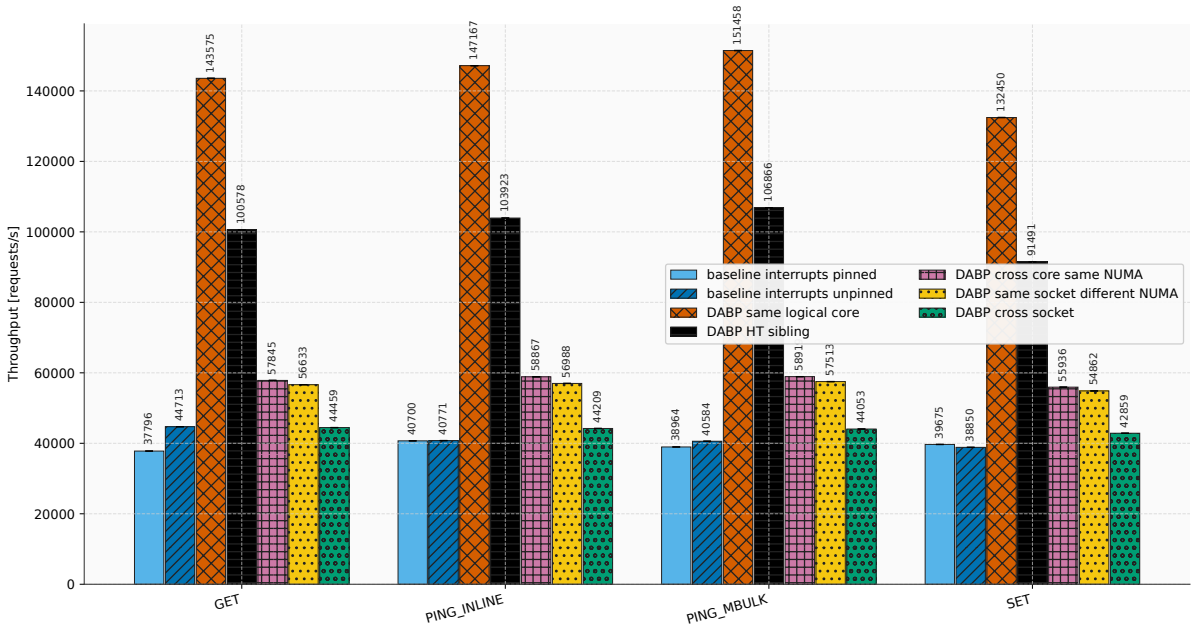


Figure 25: Valkey request throughput across daemon placement scenarios (Taurus comparison in Appendix, p. 76)

Operation	Intr-P (%)	Intr-U (%)	SC (%)	HT (%)	SN (%)	XN (%)	XS (%)
GET	84.5	100	321.1	224.9	129.4	126.7	99.4
PING inline	99.8	100	361	254.9	144.4	139.8	108.4
PING mbulk	96	100	373.2	263.3	145.2	141.7	108.5
SET	102.1	100	340.9	235.5	144	141.2	110.3

Table 15: Valkey throughput relative to the unpinned interrupt baseline

5.8 Threats to Validity

Broader threats to validity include application selection bias. The chosen programs have different sensitivities to RDMA latency, but whether they cover the full spectrum is beyond this work's scope. Hardware selection poses another broad threat to validity. The chapter evaluates all benchmarks on two machines, but other hardware could yield different results. Another potential problem is the setup used for MPI benchmarks. The results chapter focuses on different scheduling policies such as cross-NUMA or same-core, pinning benchmarks to a limited same-NUMA set of CPU cores. Results differ when all available resources are used and Figure 23 visualizes this.

To avoid overfitting to the current software and hardware setup, DABP variables from Chapter 4 remain at their default, arbitrary values. These "arbitrary values" could already be good fits for this system but not for others. Future research and implementations must sweep the parameter space to avoid local minima for only certain hardware and software. An automatic application- and hardware-specific tuning at runtime could be an even better approach.

Many placement strategies were covered in this chapter, but the same-physical-core different-hyperthread case was not included in the MPI benchmarks. Adding it to the daemon introduces another layer of complexity, so this work defers the feature, and the MPI results may miss interesting cases involving SMT siblings.

One limitation is that the Valkey throughput gains and latency reductions depend on replacing the event-file-descriptor-based progression engine with futexes, a fundamental change that requires modifying the AE library at the source level.

6 Conclusion and Outlook

This chapter summarizes the problem addressed by this thesis (Section 6.1), presents the main findings (Section 6.2), discusses limitations (Section 6.3) and outlines future work (Section 6.4).

6.1 Summary of Problem and Approach

HPC clusters and datacenters depend on RDMA networks to move data between nodes with minimal delay. Applications must wait for hardware events and react to them as quickly as possible. Busy-polling delivers sub-microsecond detection latency but burns a full CPU core continuously and collapses under oversubscription, when more processes compete for cores than are available. Interrupt-based waiting yields the CPU while waiting but introduces kernel interrupt handling, scheduling and context switch overhead that pushes average latency into the multi-microsecond range. Neither approach is satisfactory for workloads that demand both low latency and efficient resource use.

Prior work such as FastWake [2] and SmartInterrupts [3] addressed this tradeoff but required Linux kernel modifications, ruling out deployment in HPC environments where kernel replacement is impractical. FastWake limited its evaluation to microbenchmarks and SmartInterrupts to MPI workloads. Neither made centralized completion queue polling an explicit design goal.

DABP addresses this gap entirely in user space. A central daemon maps the CQ of each registered application read-only, detects new completions without consuming them and wakes blocked applications via futexes when a completion is observed. Applications block at semantic wait sites, points where the program commits to waiting for an event that will arrive independently of current control flow, transferring control to the daemon until it observes readiness.

This thesis implements DABP and integrates it via two approaches. The first replaces the busy-polling verb in libibverbs directly, requiring no application source changes. The second makes targeted source changes at semantic wait sites within Open MPI, UCX and Valkey. Integrating at the middleware level means that any application built on Open MPI benefits without requiring individual application changes. For software that manages non-RDMA backends alongside RDMA, DABP caps its blocking with a configurable timeout, allowing the application to service other backends before re-entering the wait. The evaluation covers microbenchmarks, HPC MPI workloads and Valkey, measured on two InfiniBand clusters at TU Dresden.

6.2 Main Findings

The central question of this thesis is whether a user-space implementation can achieve latencies competitive with busy-polling without exceeding the interrupt baseline. The evaluation confirms that DABP can be implemented entirely in user space. The perftest microbenchmark shows that futex-based wakeups place average detection latency below the interrupt baseline across all tested daemon placements. The progression mechanism remains event-based, so the application thread yields the CPU while waiting and concurrent progress in other processes remains possible.

Table 16 shows microbenchmark results from the perftest suite. Even the worst DABP placement (XS, cross-socket) reaches 289.7% of the busy-poll average, well below inter-

Metric	Poll (%)	Intr (%)	SC (%)	HT (%)	SN (%)	XN (%)	XS (%)
Average	100.0	341.1	201.9	229.0	172.0	182.2	289.7

Table 16: Perftest send latency relative to busy-polling baseline. Scenario abbreviations follow Table 2.

rupts at 341.1%. The best placement (SN, same NUMA) reaches 172.0% on average. For MPI workloads with shared-memory transport disabled (see Section 5.3), same-core DABP adds 1.8% to 3.6% overhead over the busy-polling baseline across the benchmark suite and across all placements. 29 of 36 measured cases remain below 10% additional runtime (see Table 7 in Section 5.6). At larger scale, a two-node run at 200 CPUs shows that DABP (SC) adds 19.7% overhead over the busy-poll baseline, compared to 46.7% for interrupts (see Table 13). For interrupt-based workloads, Valkey median latency with DABP falls to ∼23%–30% of the unpinned interrupt baseline and P99 to ∼23%. Throughput depends strongly on placement: SC reaches 373% of the unpinned interrupt baseline, HT sibling up to 263%, while cross-socket placement stays near parity at 99.4%–110.3% (see Table 15).

Daemon placement and thread count both have a measurable effect on latency and energy usage. Same-NUMA placement (SN) achieves the lowest average latency in the perftest microbenchmarks, while same-core placement (SC) remains especially efficient for MPI workloads. This difference shows that low-level futex handoff measurements explain part of the placement behavior, but full-application results also depend on scheduler effects and integration details. Same-core DABP measures at most 1% additional CPU-domain RAPL energy over busy-polling. Both approaches keep the CPU at 100% utilization, so the energy increase reflects the longer runtime, with 18 of 27 measured cases staying within 10%. Cross-socket placement (XS) is consistently the weakest strategy: it raises tail latency and can noticeably increase total CPU-domain RAPL energy.

Under oversubscription, the qualitative result is clearer than the exact ranking between placements. Busy-polling degrades severely once multiple processes compete for the same cores. CP2K `H2O-GGA`, for example, inflates to up to 2542.7% of the non-oversubscribed busy-poll runtime. Blocking approaches, including DABP, remain much more stable and can drop below 10% of the oversubscribed busy-poll runtime, as seen for GROMACS `CMET`. In the evaluated workloads, DABP generally outperforms the interrupt baseline on average and avoids the extreme runtime inflation seen with busy-polling.

6.3 Limitations

Readers should treat the current implementation of DABP as a research prototype with a deliberately narrow scope. Its strongest path is same-core or locality-aware polling on systems where thread placement is controlled and the provider exposes the CQ layout needed for read-only peeking.

Several operational boundaries follow from that focus. The daemon socket and shared-memory transport target an experimental environment and still need stronger access-control treatment for use in a broader multi-tenant setting. The implementation does not yet support daemon restart, crash recovery or cleanup of abandoned registrations.

The implementation also draws a clear provider boundary. This work targets `mlx4` and `mlx5`, matching the hardware focus of the thesis. Vendor specificity is a real drawback but not fatal for the evaluation target. Feasibility analysis during implementation suggests

that other common vendors require only provider-specific engineering work rather than fundamental changes. For Intel Ethernet, the basic CQ base and ring state are also userspace-managed, although extended CQE handling and resize-list transitions require careful integration. For Broadcom, CQ ring memory and phase-bit validity checks are likewise userspace-visible, but resize and deferred-doorbell behavior would make the integration more involved.

DABP is not a drop-in replacement in applications that manage multiple communication backends within a shared progression mechanism. Because DABP is neither a file-descriptor event source nor a classical busy-poll loop, it cannot be substituted for either without disrupting the other backends. The transparent libibverbs path avoids source changes, but it suits only applications in which RDMA alone dominates the waiting behavior. Perftest is the only evaluated program in this category.

For mixed-backend software, the main difficulty is integrating DABP without stalling progress in the other backends. Open MPI illustrates this on the busy-polling side: Open MPI's non-blocking progress engine does not expose a clean blocking model, so the current DABP integration relies on timeouts that limit, but do not remove, unnecessary waiting (see Section 4.2.2). The current system also does not implement the Open MPI shared-memory backend for DABP, which is why all MPI benchmarks in this thesis disable shared-memory communication (see Section 5.3).

Valkey illustrates the mixed-backend constraint on the interrupt-driven side. Valkey waits on event file descriptors via a blocking syscall, while DABP uses futexes. Futexes cannot transparently integrate into an event-file-descriptor-based waiting path, so the application requires source-level changes.

A final inherent constraint is that the daemon scans completion queues sequentially. Once the time to iterate all registered CQs exceeds the latency of an interrupt-based notification, DABP becomes suboptimal. The next section discusses how to address these limitations.

6.4 Future Work

Several directions exist for further improving and extending DABP, organized below by scope from internal mechanisms to broader ecosystem and hardware targets.

6.4.1 Algorithmic improvements to DABP

This subsection covers scalability to many completion queues, adaptive placement under process migration and idle periods, deeper latency analysis, formal verification of the baton-pass algorithm and a direct yield primitive.

This work does not analyze the scalability of DABP with large numbers of completion queues. A future project could determine how many completion queues can be polled sequentially before DABP becomes slower than interrupts. Once polling exceeds that threshold, the system could spawn additional daemon processes or switch to interrupt-based waiting. Heuristics to deprioritize cold completion queues could further reduce unnecessary polls.

Within datacenters, direct CPU pinning is rare, which can lead to process migration between cores. An algorithm that detects such migrations and moves completion queues to a daemon on the same NUMA domain or the same core could recover lost placement locality.

A related direction is dynamic placement based on observed idle behavior. When an application enters a sustained idle period with no competing workload on the core, the daemon could shift to a cross-core or cross-NUMA placement to reduce energy consumption, accepting slightly higher wakeup latency in exchange.

Beyond scaling and placement, this thesis evaluates DABP latency primarily through benchmarks, which leaves several analysis methods unexplored. A cache-miss analysis and profiling of scheduler decisions could identify additional bottlenecks. A complete end-to-end micro-mechanism analysis could decompose the user-to-daemon handoff, daemon detection and daemon-to-user handoff stages individually. Each stage offers room for targeted optimizations and carefully placed hardware instructions, which compound at the latency scale DABP operates in.

The baton-pass algorithm, used to coordinate the waiter and daemon, had to handle error classes such as missed wakeups. Formal verification of the current implementation could confirm its correctness and reveal further logical improvements. Better data structures for the hot-bit transfer between client and daemon are one concrete direction.

Early futex-handoff latency benchmarks showed a measurable improvement when using `SCHED_FIFO`, though doing so requires elevated permissions and can starve other processes. A related and more invasive direction is the implementation of a direct yield primitive similar to the one proposed in FastWake [2], which would allow yielding and resuming another process in a single syscall. This approach started as an initial task in this work but time constraints led to setting it aside, given that fully optimized user-space DABP already achieves very low latency. A reference implementation would serve as a lower-bound comparison and would reduce dependence on the Linux scheduler, lowering tail latencies.

6.4.2 Changes to the RDMA ecosystem

The integration work exposed that Linux currently provides no mechanism to multiplex futex-based and event-file-descriptor-based waiting in a single syscall, which limits how transparently DABP can integrate into interrupt-based applications. Extending the kernel polling interface to allow futex addresses alongside event file descriptors in a single wait call would let mixed-backend applications adopt DABP without restructuring their waiting logic. A complementary kernel extension would be a batch futex-wake primitive: when multiple completion queues become ready in the same daemon iteration, the current Linux futex API forces the daemon to issue one syscall per queue instead of one syscall for all of them. A single syscall that accepts a list of futex addresses and wakes one waiter on each would allow the daemon to deliver all pending notifications in one kernel entry, reducing the per-CQ overhead.

Open MPI currently exposes only non-blocking progress backends. An MPI implementation that provides blocking variants would integrate more naturally with DABP and avoid the timeout-driven workaround and its side effects. Extending DABP to cover the shared-memory backend in Open MPI would remove the restriction that forces all benchmarks in this thesis to disable same-node shared-memory communication.

Broader provider support is a prerequisite for wider adoption. Feasibility analysis and implementation for vendors beyond Mellanox, including verification that the completion queue resize verb is correctly handled, would make DABP portable to a larger share of

deployed hardware. In multi-tenant datacenter environments, the shared daemon access model also warrants a security review. While process isolation covers most concerns, a malicious process could potentially use daemon timing behavior as a side channel to infer the communication state of another application. DABP also lacks evaluation on RoCE, which has gained significant traction in datacenters. RoCE reaches sub-5 µs latencies comparable to InfiniBand, so reducing interrupt overhead through DABP could offer similar gains in that environment. Recent work has begun offloading completion queue polling directly to the GPU in LLM training workloads, removing the CPU proxy thread that currently drives RDMA progress [69]. This eliminates the CPU-side busy-polling overhead that DABP targets, so the current design does not directly apply. It does, however, motivate an analogous batching mechanism at the GPU level, where the same tradeoffs between polling latency and resource cost arise.

6.4.3 DABP and SIMD CPU instructions

Completion queues have a very predictable memory layout, which makes them a candidate for parallel scanning using SIMD instructions. The readiness check for a single CQ comprises four steps: (1) read the consumer index of the CQ, (2) read the last byte of the CQE at that index, (3) compare that byte to the consumer index with the lowest bit flipped and (4) treat a match as a new completion event. Because AVX-512 gather instructions support 64-bit per-lane indices with an independent TLB lookup per lane [67], each lane can address a CQ on a separate memory page. A single gather instruction then retrieves the relevant byte from each of the K queues in parallel, and a comparison instruction identifies all queues with new data simultaneously. This approach increases the number of completion queues the daemon can scan within the time budget of an interrupt-based notification, directly addressing the sequential scaling limitation identified in this thesis.

6.4.4 DABP for other targets

This thesis focuses on InfiniBand completion queues, but DABP is not specific to that resource. Any asynchronous resource with a user-space-visible readiness indicator is a candidate. Future applications could include low-latency hardware such as GPUs or storage devices, cross-core communication over shared memory pipelines, or faster dispatch for event-based primitives such as event file descriptors and mutexes.

6.5 Closing Statement

This thesis introduced DABP as a daemon-assisted batched polling mechanism that bridges the gap between busy-polling and interrupt-based waiting for RDMA applications. It presented the design of DABP, a user-space implementation that requires no Linux kernel modifications or elevated permissions and integrated it into HPC and datacenter workloads through semantic wait sites. The evaluation showed that DABP can reduce interrupt-related latency, keep overhead low for the evaluated MPI workloads and avoid the severe degradation of busy-polling under oversubscription. These results establish DABP as a viable third option in the waiting-policy design space and motivate further work on blocking progression mechanisms for RDMA and other asynchronous resources.

Bibliography

[1] L. A. Barroso, M. Marty, D. A. Patterson, and P. Ranganathan, “Attack of the Killer Microseconds,” *Communications of the ACM*, vol. 60, no. 4, pp. 48–54, Mar. 2017, doi: 10.1145/3015146.

[2] B. Li, Z. Xiang, X. Wang, H. Ruan, J. Zhou, and K. Tan, “FastWake: Revisiting Host Network Stack for Interrupt-mode RDMA,” in *Proceedings of the 7th Asia-Pacific Workshop on Networking*, in APNet '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1–7. doi: 10.1145/3600061.3600063.

[3] K. Kumar, J. A. Zounmevo, and A. Afsahi, “SmartInterrupts: A Node-Wide Asynchronous Message Progression Technique,” in *Proceedings of the 28th European MPI Users' Group Meeting*, in EuroMPI '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://www.queensu.ca/academia/afsahi/pprl/papers/EuroMPI-2021.pdf

[4] A. Gangidi *et al.*, “RDMA over Ethernet for Distributed AI Training at Meta Scale,” in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024. doi: 10.1145/3651890.3672233.

[5] H. Franke, R. Russell, and M. Kirkwood, “Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux,” in *Proceedings of the Ottawa Linux Symposium*, 2002, pp. 479–495. [Online]. Available: https://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf

[6] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan, “No More Backstabbing... A Faithful Scheduling Policy for Multithreaded Programs,” in *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, in PACT '11. Galveston Island, TX, USA: IEEE, 2011, pp. 12–21. doi: 10.1109/PACT.2011.8.

[7] G. Costa, “Introducing Glommio, a Thread-per-Core Crate for Rust and Linux.” Accessed: Apr. 18, 2026. [Online]. Available: https://www.datadoghq.com/blog/engineering/introducing-glommio/

[8] Linux man-pages project, “memfd_create(2) - Linux manual page.” Accessed: Mar. 31, 2026. [Online]. Available: https://man7.org/linux/man-pages/man2/memfd_create.2.html

[9] Linux man-pages project, “eventfd(2) - Linux manual page.” Accessed: Mar. 31, 2026. [Online]. Available: https://man7.org/linux/man-pages/man2/eventfd.2.html

[10] Linux man-pages project, “epoll(7) - Linux manual page.” Accessed: Mar. 31, 2026. [Online]. Available: https://man7.org/linux/man-pages/man7/epoll.7.html

[11] Linux man-pages project, “futex(7) - Linux manual page.” Accessed: Mar. 31, 2026. [Online]. Available: https://man7.org/linux/man-pages/man7/futex.7.html

[12] Linux man-pages project, “futex(2) - Linux manual page.” Accessed: Mar. 31, 2026. [Online]. Available: https://man7.org/linux/man-pages/man2/futex.2.html

[13] The Linux Kernel documentation, “futex2.” Accessed: Mar. 31, 2026. [Online]. Available: https://docs.kernel.org/6.11/userspace-api/futex2.html

[14] The Linux Kernel documentation, “CFS Scheduler.” Accessed: Apr. 16, 2026. [Online]. Available: https://docs.kernel.org/scheduler/sched-design-CFS.html

[15] The Linux Kernel documentation, “EEVDF Scheduler.” Accessed: Apr. 16, 2026. [Online]. Available: https://docs.kernel.org/scheduler/sched-eevdf.html

[16] Linux man-pages project, “sched(7) - Linux manual page.” Accessed: Mar. 31, 2026. [Online]. Available: https://man7.org/linux/man-pages/man7/sched.7.html

[17] Linux man-pages project, “cgroups(7) - Linux manual page.” Accessed: Mar. 31, 2026. [Online]. Available: https://man7.org/linux/man-pages/man7/cgroups.7.html

[18] SchedMD, “Resource Binding.” Accessed: Mar. 31, 2026. [Online]. Available: https://slurm.schedmd.com/resource_binding.html

[19] Linux man-pages project, “sched_setaffinity(2) - Linux manual page.” Accessed: Mar. 31, 2026. [Online]. Available: https://man7.org/linux/man-pages/man2/sched_setaffinity.2.html

[20] The Linux Kernel documentation, “CPU Performance Scaling.” Accessed: Apr. 16, 2026. [Online]. Available: https://docs.kernel.org/admin-guide/pm/cpufreq.html

[21] The Linux Kernel documentation, “CPU Idle Time Management.” Accessed: Apr. 16, 2026. [Online]. Available: https://docs.kernel.org/admin-guide/pm/cpuidle.html

[22] InfiniBand Trade Association, “InfiniBand Extends Leadership in November 2025 TOP500 List; Connects 7 of the Top 10 Green500 Supercomputers.” Accessed: Apr. 13, 2026. [Online]. Available: https://www.infinibandta.org/infiniband-extends-leadership-in-november-2025-top500-list-connects-7-of-the-top-10-green500-supercomputers/

[23] NVIDIA Corporation, “NVIDIA Completes Acquisition of Mellanox, Creating Major Force Driving Next-Gen Data Centers.” Accessed: Apr. 16, 2026. [Online]. Available: https://nvidianews.nvidia.com/news/nvidia-completes-acquisition-of-mellanox-creating-major-force-driving-next-gen-data-centers

[24] Dell'Oro Group, “Ethernet More Than Doubles Size of InfiniBand as the Leading Fabric for AI Scale-Out Networks in 2025.” Accessed: Apr. 13, 2026. [Online]. Available: https://www.delloro.com/news/ethernet-more-than-doubles-size-of-infiniband-as-the-leading-fabric-for-ai-scale-out-networks-in-2025/

[25] linux-rdma, “RDMA Core Userspace Libraries and Daemons.” Accessed: Apr. 13, 2026. [Online]. Available: https://github.com/linux-rdma/rdma-core

[26] Linux man-pages project, “ibv_create_cq, ibv_resize_cq, ibv_destroy_cq(3) - Libibverbs Programmer's Manual.” Accessed: Mar. 31, 2026. [Online]. Available: https://man7.org/linux/man-pages/man3/ibv_destroy_cq.3.html

[27] Linux man-pages project, “ibv_req_notify_cq(3) - Libibverbs Programmer's Manual.” Accessed: Mar. 31, 2026. [Online]. Available: https://man7.org/linux/man-pages/man3/ibv_req_notify_cq.3.html

[28] Linux man-pages project, “ibv_get_cq_event(3) - Libibverbs Programmer's Manual.” Accessed: Mar. 31, 2026. [Online]. Available: https://man7.org/linux/man-pages/man3/ibv_get_cq_event.3.html

[29] A. Kalia, M. Kaminsky, and D. G. Andersen, “Design Guidelines for High Performance RDMA Systems,” in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO: USENIX Association, June 2016, pp. 437–450. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[30] R. Mittal *et al.*, “Revisiting Network Support for RDMA,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, in SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 313–326. doi: 10.1145/3230543.3230557.

[31] Linux man-pages project, “ibv_poll_cq(3) - Libibverbs Programmer's Manual.” Accessed: Mar. 31, 2026. [Online]. Available: https://man7.org/linux/man-pages/man3/ibv_poll_cq.3.html

[32] A. Rosenbaum, “Multiprocess Sharing of RDMA Resources,” in *14th Annual OpenFabrics Alliance Workshop*, Monterey, CA, USA, Mar. 2018. [Online]. Available: https://openfabrics.org/images/2018workshop/presentations/103_ARosenbaum_Multi-ProcessSharing.pdf

[33] P. Shamis *et al.*, “UCX: An Open Source Framework for HPC Network APIs and Beyond,” in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, IEEE, 2015, pp. 40–43. doi: 10.1109/HOTI.2015.13.

[34] N. Hanford, V. Ahuja, M. K. Farrens, B. Tierney, and D. Ghosal, “A Survey of End-System Optimizations for High-Speed Networks,” *ACM Computing Surveys*, vol. 51, no. 3, July 2018, doi: 10.1145/3184899.

[35] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, “Active Messages: A Mechanism for Integrated Communication and Computation,” in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, in ISCA '92. New York, NY, USA: Association for Computing Machinery, 1992, pp. 256–266. doi: 10.1145/139669.140382.

[36] K. Langendoen, J. Romein, R. Bhoedjang, and H. E. Bal, “Integrating Polling, Interrupts, and Thread Management,” in *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, Annapolis, MD, USA: IEEE, Oct. 1996, pp. 13–22. doi: 10.1109/FMPC.1996.558057.

[37] E. Münzberg, “Semantic CPU Yielding for Oversubscribed RDMA-Applications,” Bachelor's thesis, Technische Universität Dresden, Dresden, Germany, 2023.

[38] M. Planeta, J. Bierbaum, M. Roitzsch, and H. Härtig, “CoRD: Converged RDMA Dataplane,” in *Proceedings of the 2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE Computer Society, 2025, pp. 1074–1090. doi: 10.1109/IPDPS64566.2025.00099.

[39] C. Dovrolis, B. Thayer, and P. Ramanathan, “HIP: Hybrid Interrupt-Polling for the Network Interface,” *ACM SIGOPS Operating Systems Review*, vol. 35, no. 4, pp. 50–60, 2001, doi: 10.1145/506084.506089.

[40] J. Yang, D. B. Minturn, and F. Hady, “When Poll Is Better than Interrupt,” in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST 12)*, Berkeley,

CA: USENIX Association, Feb. 2012. [Online]. Available: https://www.usenix.org/conference/fast12/when-poll-better-interrupt

[41] J. C. Mogul and K. K. Ramakrishnan, "Eliminating Receive Livelock in an Interrupt-Driven Kernel," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 217–252, 1997, doi: 10.1145/263326.263335.

[42] N. Basu, C. Montanari, and J. Eriksson, "Frequent Background Polling on a Shared Thread, Using Light-Weight Compiler Interrupts," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, in PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1249–1263. doi: 10.1145/3453483.3454107.

[43] R. Zhang, T. Kim, D. Weber, and M. Schwarz, "(M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels," in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023, pp. 7267–7284. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-ruiyi

[44] B. Aydogmus *et al.*, "Extended User Interrupts (xUI): Fast and Flexible Notification without Polling," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, in ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 373–389. doi: 10.1145/3676641.3716028.

[45] A. Belay *et al.*, "The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane," *ACM Transactions on Computer Systems*, vol. 34, no. 4, pp. 1–39, Dec. 2016, doi: 10.1145/2997641.

[46] G. Prekas, M. Kogias, and E. Bugnion, "ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks," in *Proceedings of the 26th Symposium on Operating Systems Principles*, in SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 325–341. doi: 10.1145/3132747.3132780.

[47] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency", in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA: USENIX Association, Feb. 2019, pp. 345–360. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/kaffes

[48] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA: USENIX Association, Feb. 2019, pp. 361–378. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[49] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating Interference at Microsecond Timescales," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, Berkeley, CA: USENIX Association, Nov. 2020, pp. 281–297. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/fried

[50] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout, "Arachne: Core-Aware Thread Management," in *13th USENIX Symposium on Operating Systems Design and Implemen-*

tation (OSDI 18), Carlsbad, CA: USENIX Association, Oct. 2018, pp. 145–160. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/qin

[51] H. M. Demoulin *et al.*, "When Idling Is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, in SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 621–637. doi: 10.1145/3477132.3483571.

[52] Data Plane Development Kit Project, "Programmer's Guide, Version 25.03.0." 2025. Accessed: Apr. 16, 2026. [Online]. Available: https://doc.dpdk.org/guides-25.03/prog_guide/index.html

[53] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Prentice Hall, 1988.

[54] linux-rdma, "Infiniband Verbs Performance Tests." Accessed: Apr. 18, 2026. [Online]. Available: https://github.com/linux-rdma/perftest

[55] S. Pronk *et al.*, "GROMACS 4.5: a high-throughput and highly parallel open source molecular simulation toolkit," *Bioinformatics*, vol. 29, no. 7, pp. 845–854, 2013, doi: 10.1093/bioinformatics/btt055.

[56] T. D. Kühne *et al.*, "CP2K: An electronic structure and molecular dynamics software package — Quickstep: Efficient and accurate electronic structure calculations," *The Journal of Chemical Physics*, vol. 152, no. 19, p. 194103, 2020, doi: 10.1063/5.0007045.

[57] Exascale Computing Project, "HPCCG - ECP Proxy Applications." Accessed: Apr. 18, 2026. [Online]. Available: https://proxyapps.exascaleproject.org/app/hpccg/

[58] D. H. Bailey *et al.*, "The NAS Parallel Benchmarks," *The International Journal of Supercomputer Applications*, vol. 5, no. 3, pp. 63–73, 1991, doi: 10.1177/109434209100500306.

[59] Department of Theoretical and Computational Biophysics, Max Planck Institute for Multidisciplinary Sciences, "A free GROMACS benchmark set." Accessed: Apr. 18, 2026. [Online]. Available: https://www.mpinat.mpg.de/grubmueller/bench

[60] CP2K developers group, "CP2K benchmark inputs in the v2024.1 source distribution." Accessed: Apr. 18, 2026. [Online]. Available: https://github.com/cp2k/cp2k/tree/v2024.1/benchmarks

[61] valkey-io, "valkey." Accessed: Apr. 18, 2026. [Online]. Available: https://github.com/valkey-io/valkey

[62] Redis Ltd., "Introduction to Redis." Accessed: Apr. 18, 2026. [Online]. Available: https://redis.io/about/

[63] Redis Ltd., "Redis Adopts Dual Source-Available Licensing." Accessed: Apr. 18, 2026. [Online]. Available: https://redis.io/blog/redis-adopts-dual-source-available-licensing/

[64] Center for Information Services and High Performance Computing (ZIH), TU Dresden, "Barnard: High-Performance Computing System Overview." Accessed: Apr. 18, 2026. [Online]. Available: https://tu-dresden.de/zih/hochleistungsrechnen/hpc?set_language=en

[65] E. Dolstra, M. de Jonge, and E. Visser, “Nix: A Safe and Policy-Free System for Software Deployment,” in *Proceedings of the 18th Large Installation System Administration Conference (LISA '04)*, 2004, pp. 79–92. [Online]. Available: https://nixos.org/~eelco/pubs/nspfssd-lisa2004-final.pdf

[66] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, “RAPL in Action: Experiences in Using RAPL for Power Measurements,” *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 3, no. 2, pp. 1–26, Mar. 2018, doi: 10.1145/3177754.

[67] Intel Corporation, “Intel 64 and IA-32 Architectures Software Developer's Manual, Rev. 091.” 2026. Accessed: Apr. 18, 2026. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[68] The Linux Kernel documentation, “CPU Scheduler implementation hints for architecture-specific code.” Accessed: Apr. 18, 2026. [Online]. Available: https://docs.kernel.org/scheduler/sched-arch.html

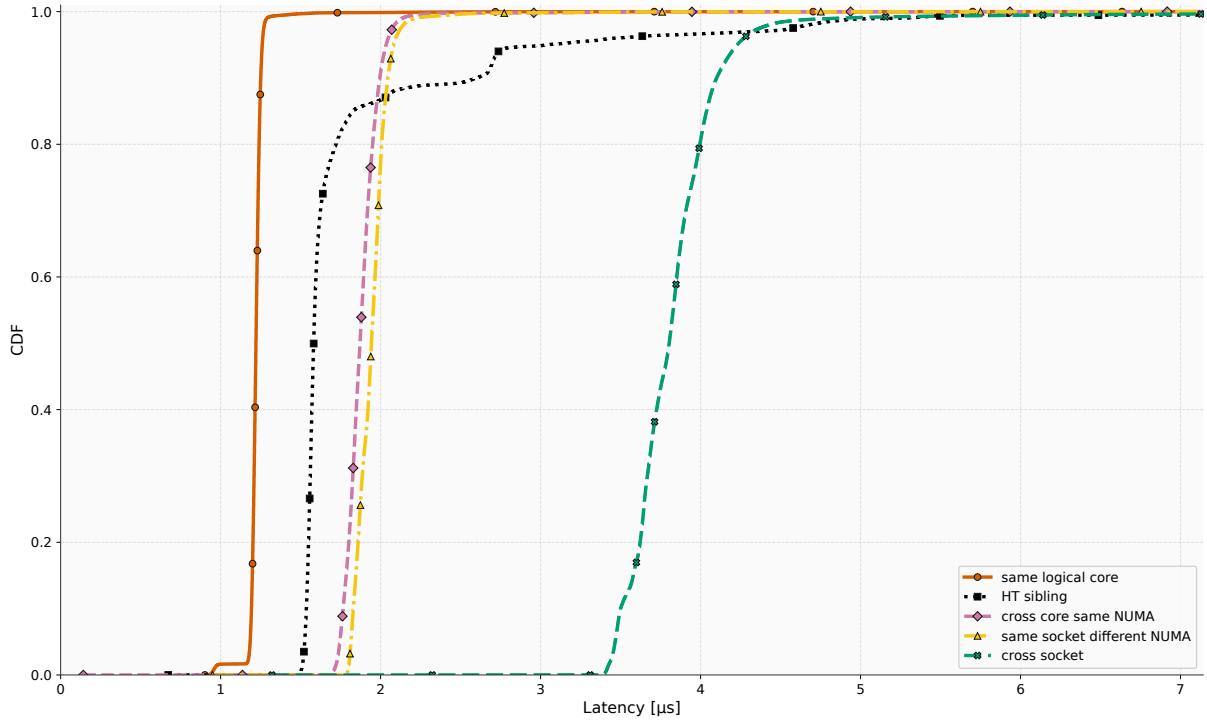
[69] K. Hamidouche *et al.*, “GPU-Initiated Networking for NCCL,” *arXiv preprint arXiv:2511.15076*, 2025, Accessed: Apr. 26, 2026. [Online]. Available: https://arxiv.org/abs/2511.15076

Appendix

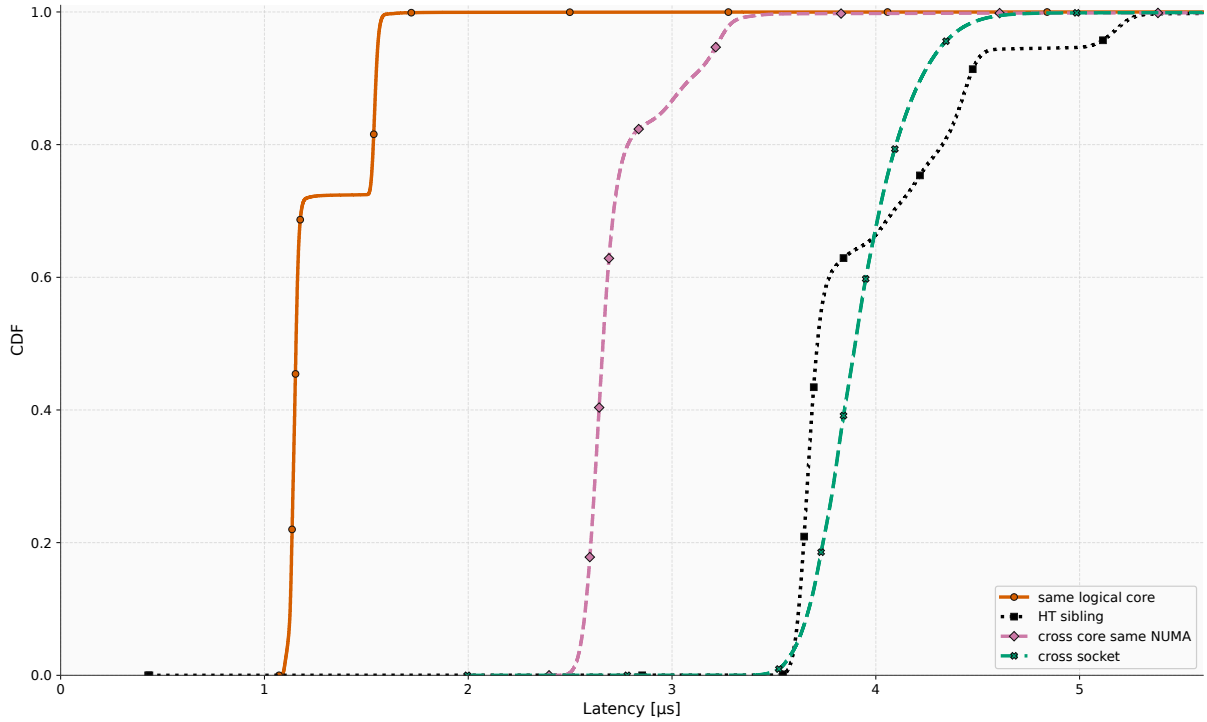
Futex Wake-Latency Comparison

Back to Figure 16

Barnard



Taurus



Barnard

Place	p50	p99	avg
same logical core	1.22 µs	1.29 µs	1.221 µs
HT sibling	1.581 µs	5.224 µs	1.821 µs
cross core same NUMA	1.871 µs	2.135 µs	1.881 µs
same socket different NUMA	1.942 µs	2.216 µs	1.945 µs
cross socket	3.804 µs	4.848 µs	3.835 µs

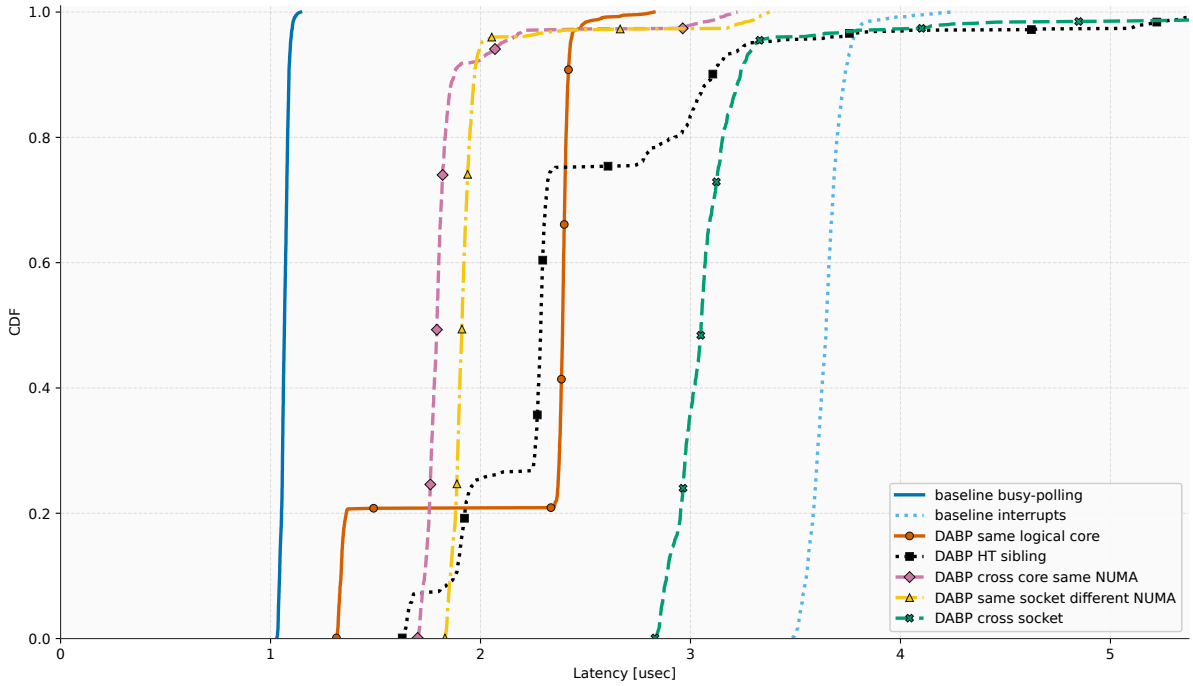
Taurus

Place	p50	p99	avg
same logical core	1.155 µs	1.575 µs	1.257 µs
HT sibling	3.713 µs	5.25 µs	3.945 µs
cross core same NUMA	2.66 µs	3.33 µs	2.747 µs
cross socket	3.898 µs	4.532 µs	3.928 µs

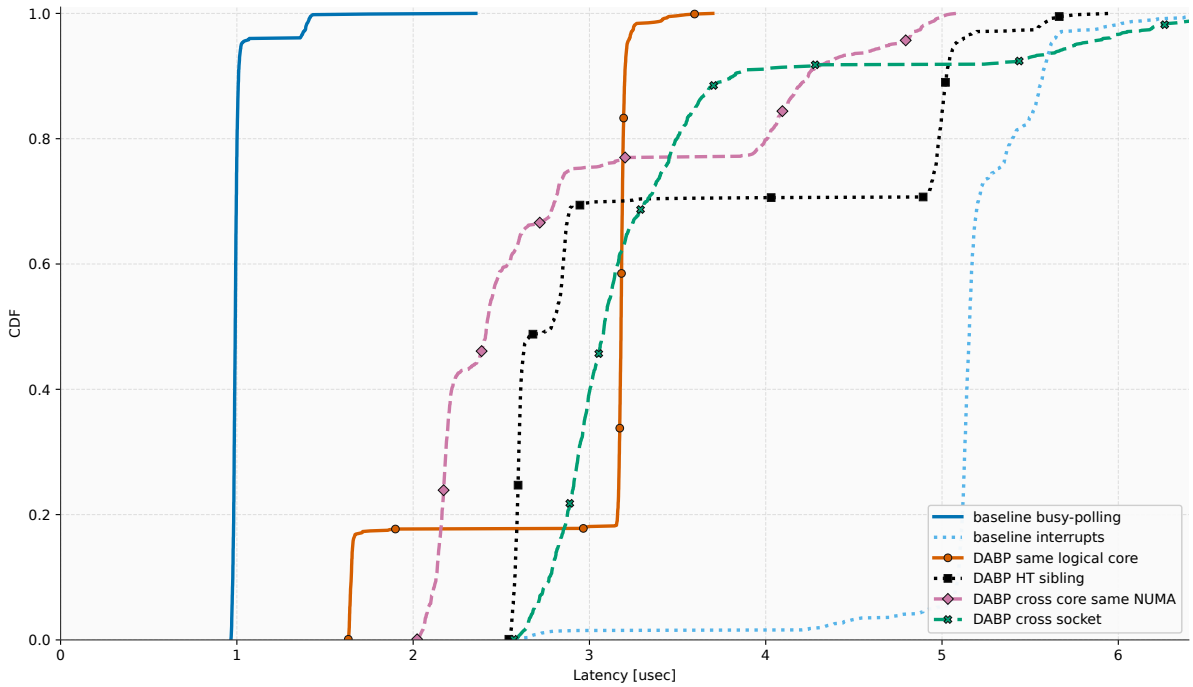
Perftest CDF Comparison

Back to Figure 17

Barnard



Taurus



Barnard

Scen.	p50	avg	p99
Poll	1.066 µs	1.07 µs	1.11 µs
Intr	3.646 µs	3.65 µs	3.96 µs
SC	2.391 µs	2.16 µs	2.58 µs
HT	2.287 µs	2.45 µs	5.36 µs
SN	1.792 µs	1.84 µs	3.14 µs
DABP same socket different NUMA	1.912 µs	1.95 µs	3.32 µs
XS	3.05 µs	3.1 µs	5.53 µs

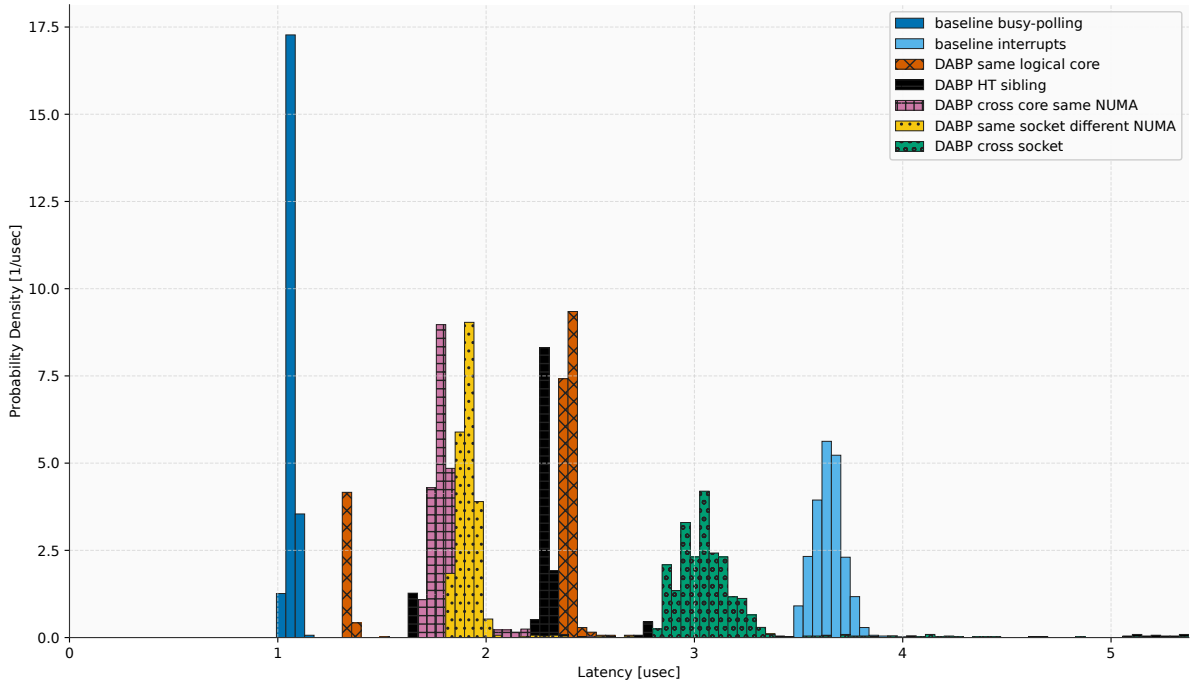
Taurus

Scen.	p50	avg	p99
Poll	0.991 µs	1.01 µs	1.41 µs
Intr	5.156 µs	5.22 µs	6.24 µs
SC	3.179 µs	2.9 µs	3.42 µs
HT	2.785 µs	3.4 µs	5.61 µs
SN	2.416 µs	2.83 µs	4.97 µs
XS	3.083 µs	3.35 µs	6.46 µs

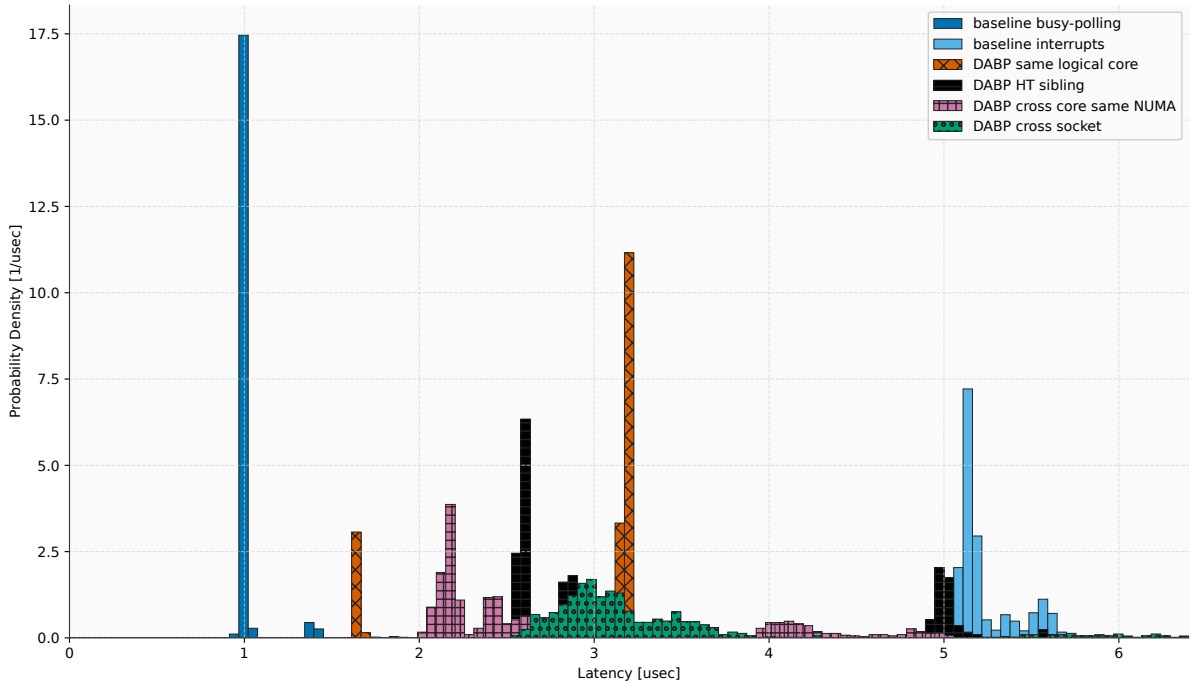
Perftest Histogram Comparison

Back to Figure 18

Barnard



Taurus



Barnard

Scen.	p50	avg	p99
Poll	1.066 µs	1.07 µs	1.11 µs
Intr	3.646 µs	3.65 µs	3.96 µs
SC	2.391 µs	2.16 µs	2.58 µs
HT	2.287 µs	2.45 µs	5.36 µs
SN	1.792 µs	1.84 µs	3.14 µs
DABP same socket different NUMA	1.912 µs	1.95 µs	3.32 µs
XS	3.05 µs	3.1 µs	5.53 µs

Taurus

Scen.	p50	avg	p99
Poll	0.991 µs	1.01 µs	1.41 µs
Intr	5.156 µs	5.22 µs	6.24 µs
SC	3.179 µs	2.9 µs	3.42 µs
HT	2.785 µs	3.4 µs	5.61 µs
SN	2.416 µs	2.83 µs	4.97 µs
XS	3.083 µs	3.35 µs	6.46 µs

Perftest Summary Comparison

Back to Table 4 | Back to Table 5

Barnard summary

Scen.	p50	avg	p99
Poll	1.066 µs	1.07 µs	1.11 µs
Intr	3.646 µs	3.65 µs	3.96 µs
SC	2.391 µs	2.16 µs	2.58 µs
HT	2.287 µs	2.45 µs	5.36 µs
SN	1.792 µs	1.84 µs	3.14 µs
DABP same socket different NUMA	1.912 µs	1.95 µs	3.32 µs
XS	3.05 µs	3.1 µs	5.53 µs

Taurus summary

Scen.	p50	avg	p99
Poll	0.991 µs	1.01 µs	1.41 µs
Intr	5.156 µs	5.22 µs	6.24 µs
SC	3.179 µs	2.9 µs	3.42 µs
HT	2.785 µs	3.4 µs	5.61 µs
SN	2.416 µs	2.83 µs	4.97 µs
XS	3.083 µs	3.35 µs	6.46 µs

Barnard relative to busy polling

Scen.	avg	p99
Poll	100%	100%
Intr	341.1%	356.8%
SC	201.9%	232.4%
HT	229%	482.9%
SN	172%	282.9%
XN	182.2%	299.1%
XS	289.7%	498.2%

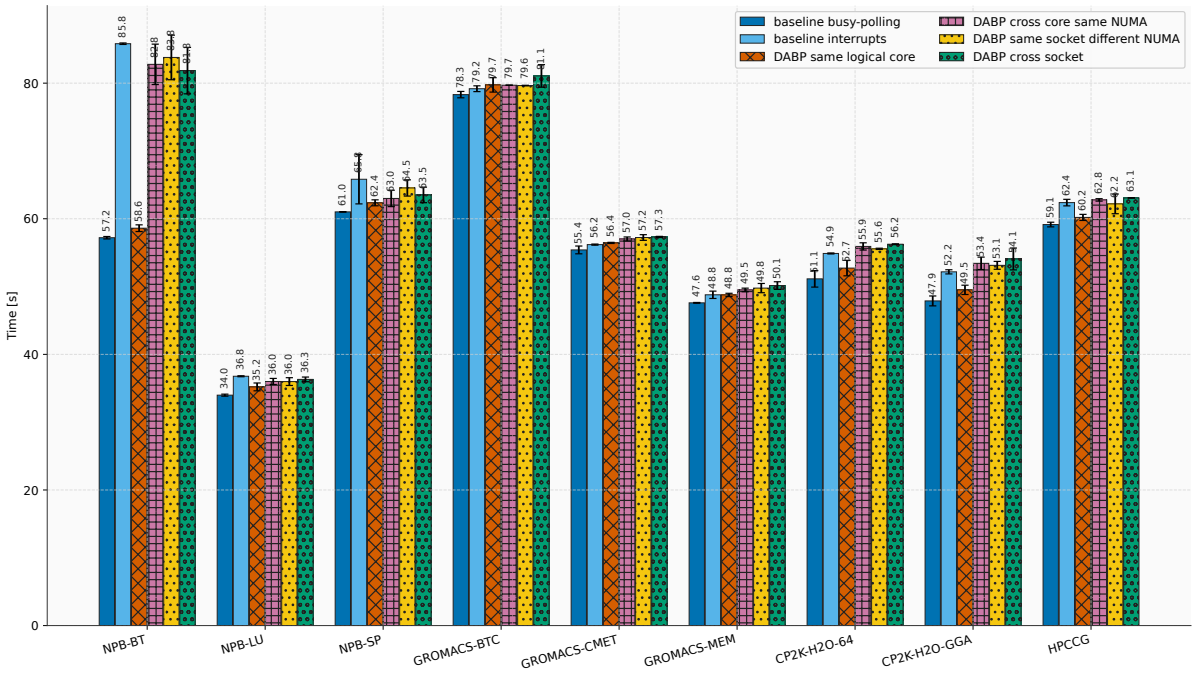
Taurus relative to busy polling

Scen.	avg	p99
Poll	100%	100%
Intr	516.8%	442.6%
SC	287.1%	242.6%
HT	336.6%	397.9%
SN	280.2%	352.5%
XS	331.7%	458.2%

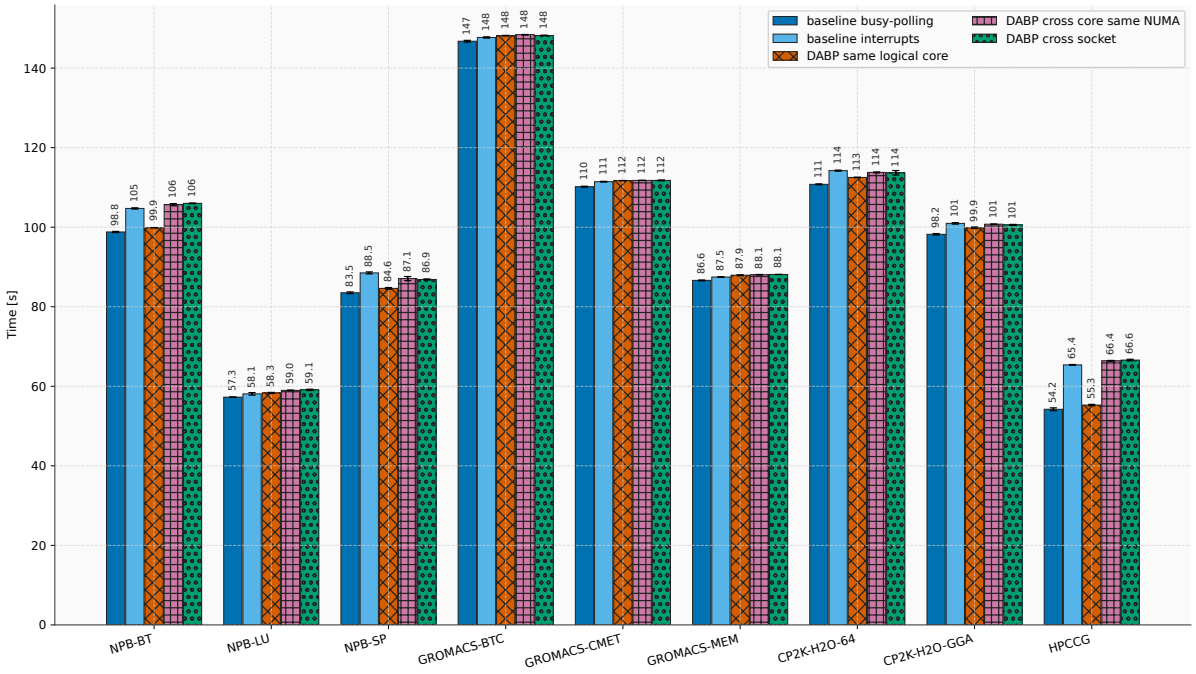
MPI Runtime Comparison

Back to Figure 19

Barnard



Taurus



Barnard

Bench	Poll	Intr	SC	SN	XN	XS
NPB-BT	100%	150%	102.5%	144.7%	146.5%	143.1%
NPB-LU	100%	108.2%	103.6%	105.9%	105.9%	106.8%
NPB-SP	100%	107.9%	102.2%	103.2%	105.8%	104.1%
GROMACS-BTC	100%	101.1%	101.8%	101.8%	101.7%	103.5%
GROMACS-CMET	100%	101.4%	101.9%	102.9%	103.3%	103.5%
GROMACS-MEM	100%	102.5%	102.4%	104%	104.6%	105.4%
CP2K-H2O-64	100%	107.3%	103.1%	109.4%	108.7%	110%
CP2K-H2O-GGA	100%	109%	103.4%	111.6%	111%	112.9%
HPCCG	100%	105.5%	101.8%	106.1%	105.2%	106.6%

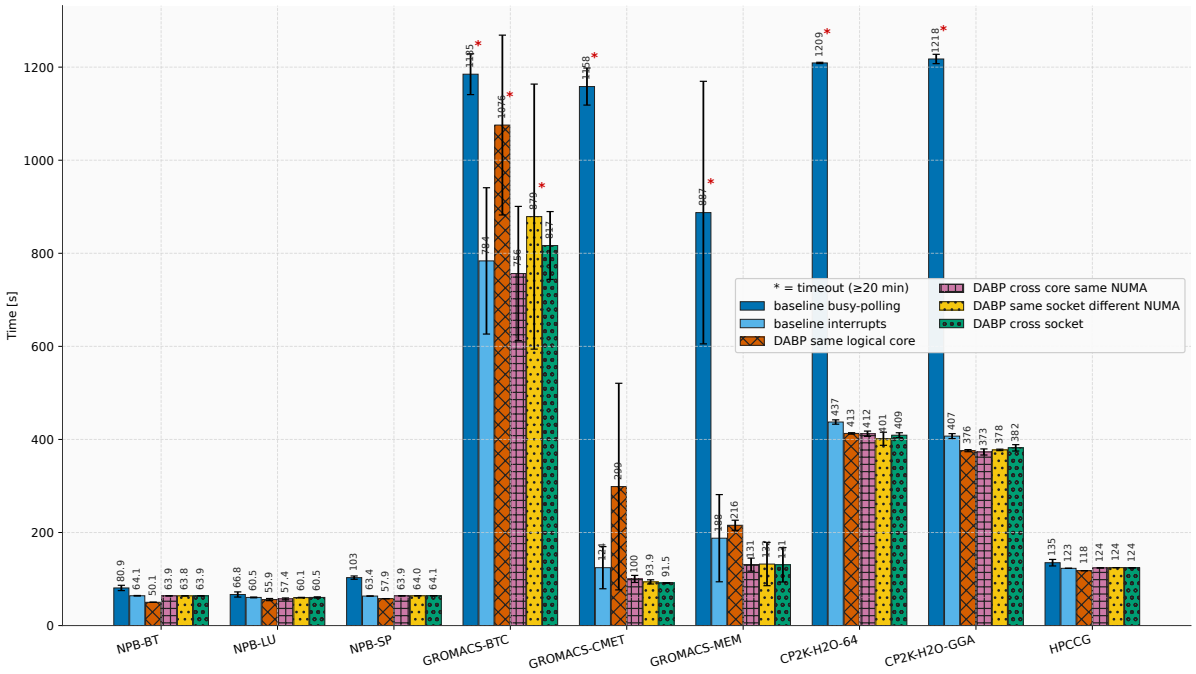
Taurus

Bench	Poll	Intr	SC	SN	XS
NPB-BT	100%	106%	101.1%	106.9%	107.3%
NPB-LU	100%	101.4%	101.8%	103%	103.2%
NPB-SP	100%	106%	101.3%	104.3%	104%
GROMACS-BTC	100%	100.6%	101%	101.1%	101%
GROMACS-CMET	100%	101.1%	101.4%	101.4%	101.4%
GROMACS-MEM	100%	101%	101.5%	101.6%	101.7%
CP2K-H2O-64	100%	103.1%	101.6%	102.7%	102.6%
CP2K-H2O-GGA	100%	102.8%	101.7%	102.6%	102.4%
HPCCG	100%	120.6%	101.9%	122.5%	122.8%

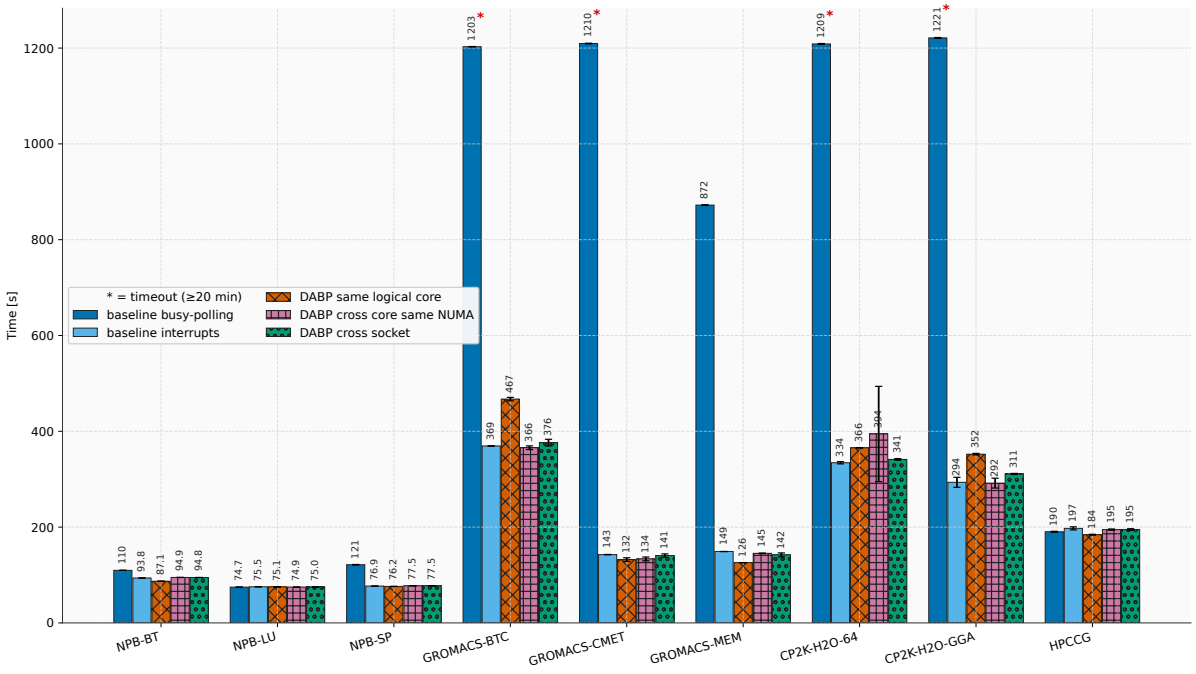
MPI Runtime 2x Oversubscription Comparison

Back to Figure 21

Barnard



Taurus



Barnard

Bench	Poll	Intr	SC	SN	XN	XS
NPB-BT	100%	79.3%	61.9%	79%	78.9%	79%
NPB-LU	100%	90.5%	83.6%	85.9%	89.9%	90.6%
NPB-SP	100%	61.5%	56.1%	62%	62.1%	62.2%
GROMACS-BTC	100%	66.1%	90.8%	63.8%	74.2%	68.9%
GROMACS-CMET	100%	10.7%	25.8%	8.7%	8.1%	7.9%
GROMACS-MEM	100%	21.2%	24.3%	14.8%	14.9%	14.7%
CP2K-H2O-64	100%	36.2%	34.2%	34.1%	33.2%	33.8%
CP2K-H2O-GGA	100%	33.4%	30.9%	30.6%	31%	31.4%
HPCCG	100%	91%	87.3%	91.7%	92.1%	91.8%

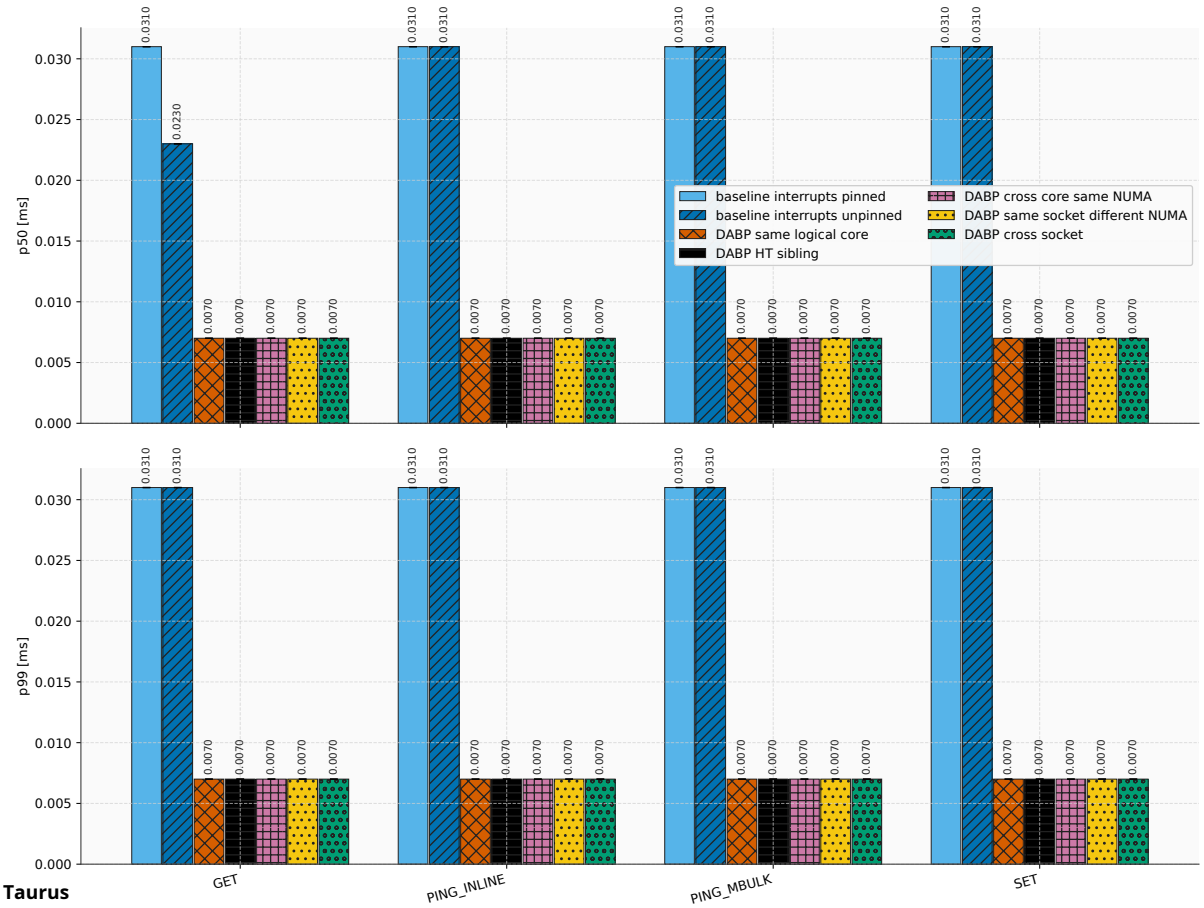
Taurus

Bench	Poll	Intr	SC	SN	XS
NPB-BT	100%	85.5%	79.3%	86.5%	86.4%
NPB-LU	100%	101%	100.5%	100.2%	100.4%
NPB-SP	100%	63.3%	62.7%	63.8%	63.8%
GROMACS-BTC	100%	30.7%	38.8%	30.4%	31.3%
GROMACS-CMET	100%	11.8%	10.9%	11%	11.6%
GROMACS-MEM	100%	17.1%	14.4%	16.7%	16.3%
CP2K-H2O-64	100%	27.7%	30.2%	32.6%	28.2%
CP2K-H2O-GGA	100%	24%	28.8%	23.9%	25.5%
HPCCG	100%	103.7%	96.7%	102.3%	102.5%

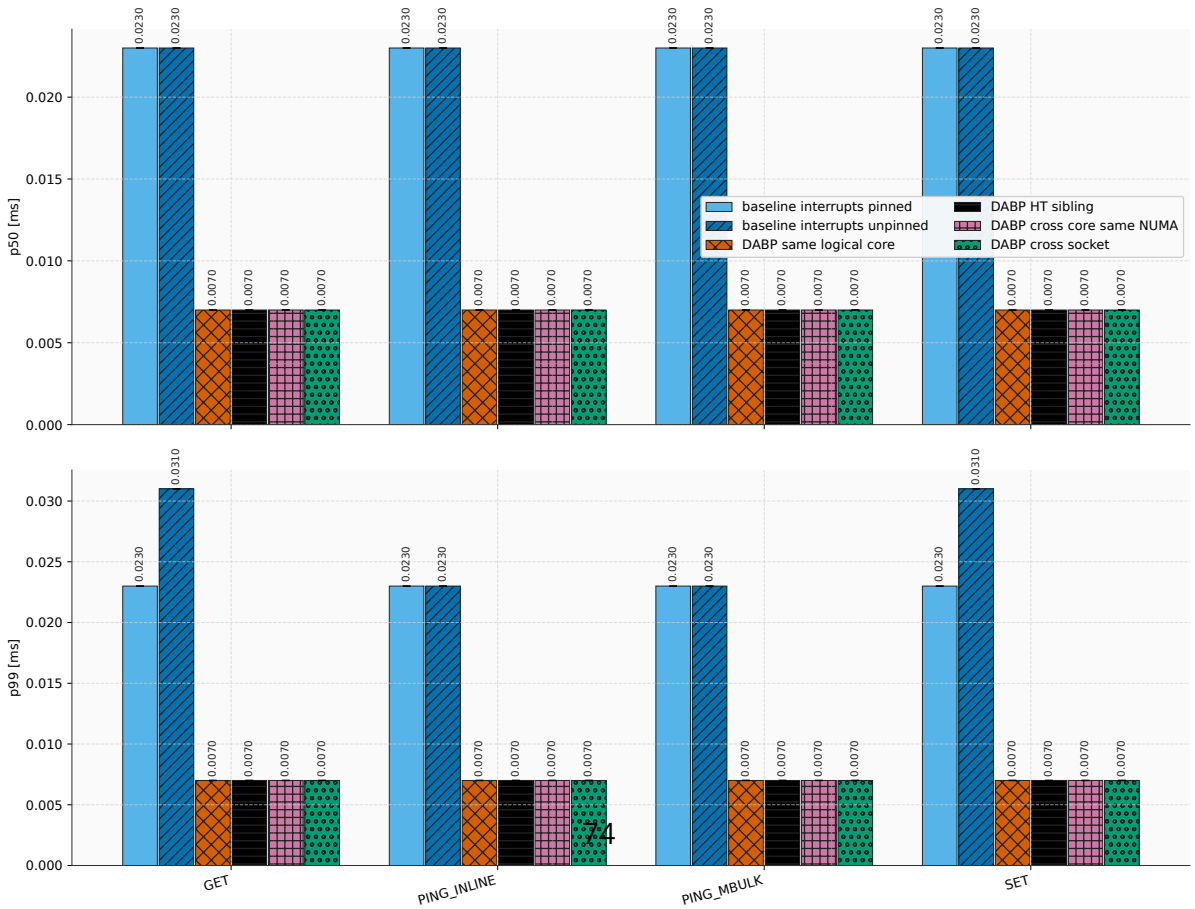
Valkey Latency Comparison

Back to Figure 24

Barnard



Taurus



Barnard

Op	p50 Blu	p50 SC	p99 Blu	p99 SC
GET	100%	30.4%	100%	22.6%
PI	100%	22.6%	100%	22.6%
PM	100%	22.6%	100%	22.6%
SET	100%	22.6%	100%	22.6%

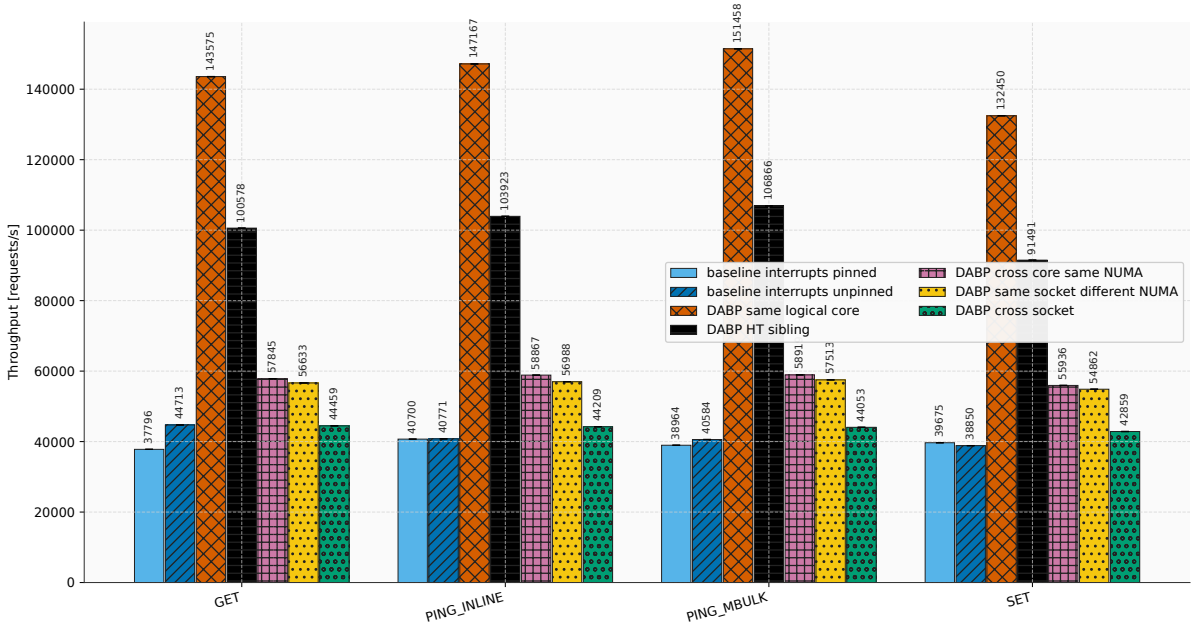
Taurus

Op	p50 Blu	p50 SC	p99 Blu	p99 SC
GET	100%	30.4%	100%	22.6%
PI	100%	30.4%	100%	30.4%
PM	100%	30.4%	100%	30.4%
SET	100%	30.4%	100%	22.6%

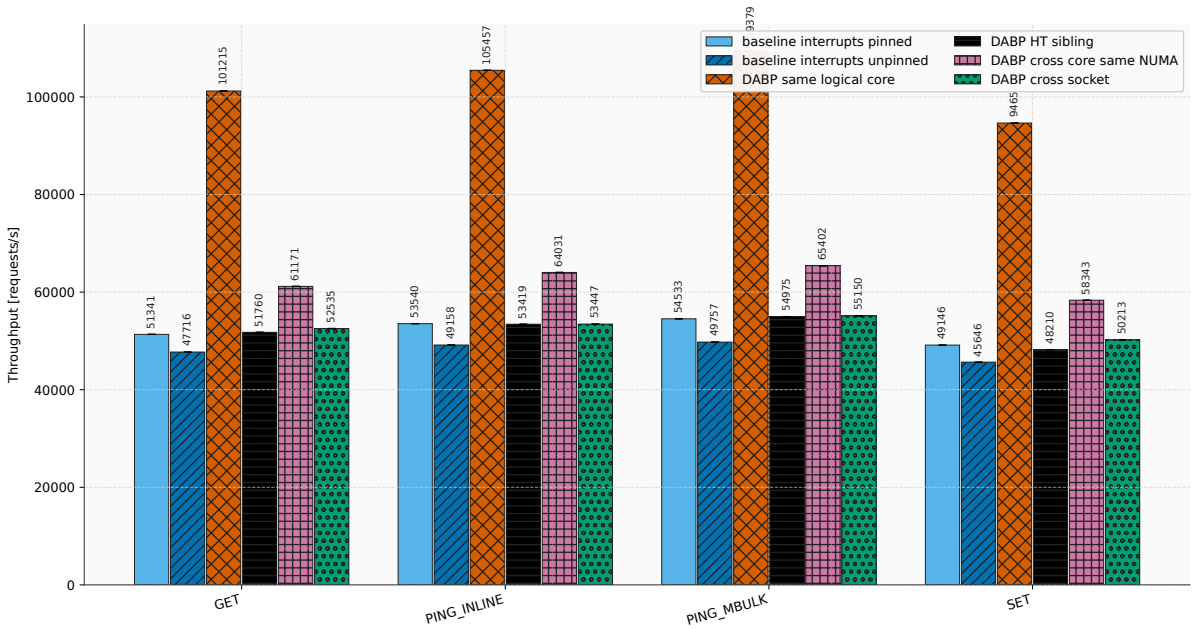
Valkey Throughput Comparison

Back to Figure 25

Barnard



Taurus



Barnard

Op	Blu	SC	SN	XS
GET	100%	321.1%	129.4%	99.4%
PI	100%	361%	144.4%	108.4%
PM	100%	373.2%	145.2%	108.5%
SET	100%	340.9%	144%	110.3%

Taurus

Op	Blu	SC	SN	XS
GET	100%	212.1%	128.2%	110.1%
PI	100%	214.5%	130.3%	108.7%
PM	100%	219.8%	131.4%	110.8%
SET	100%	207.4%	127.8%	110%

Execution Traces

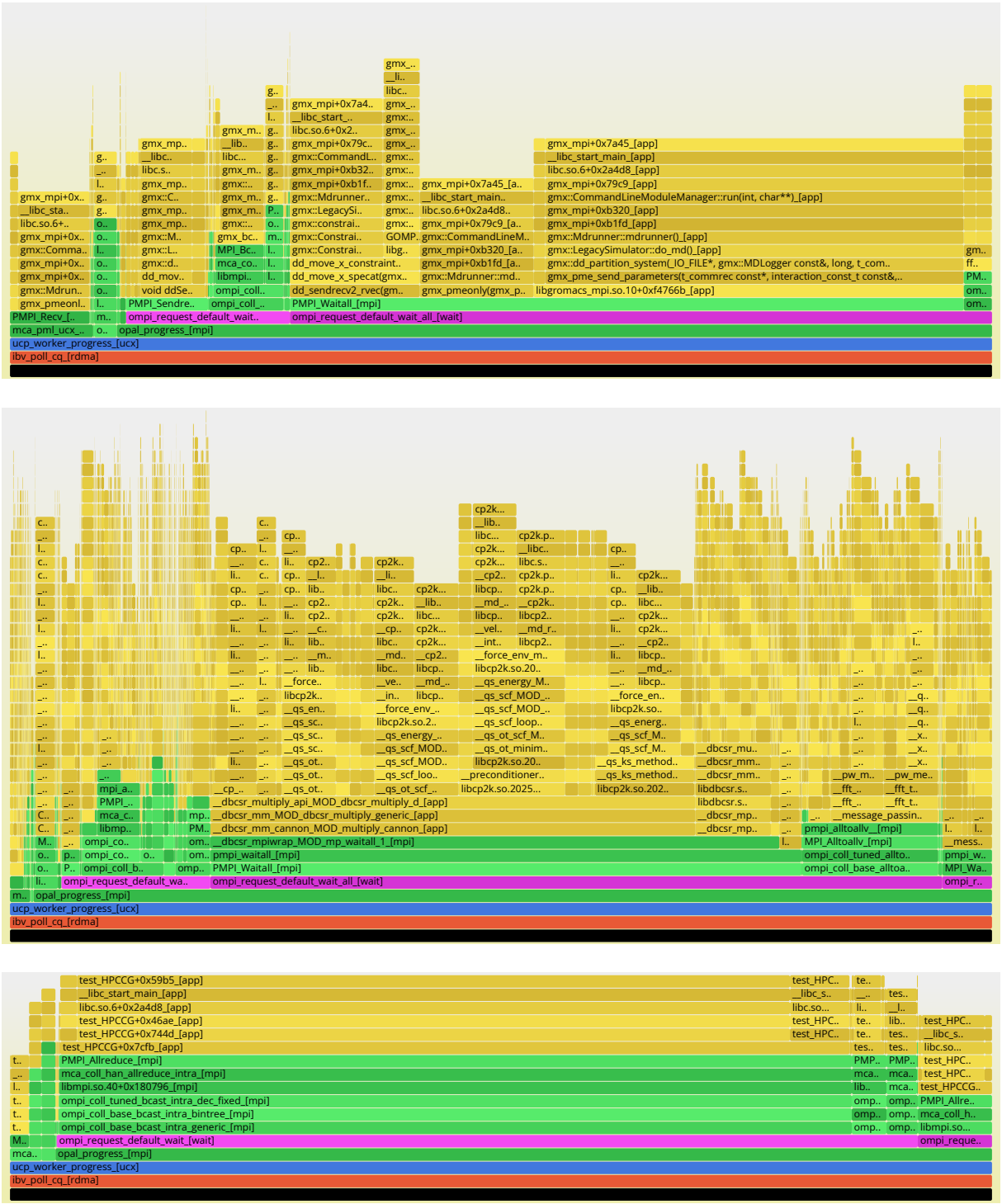


Figure 26: Execution traces for all benchmark workloads (1/2: GROMACS, CP2K, HPCCG). Each trace shows the layer-by-layer time breakdown for the baseline and DABP configurations.

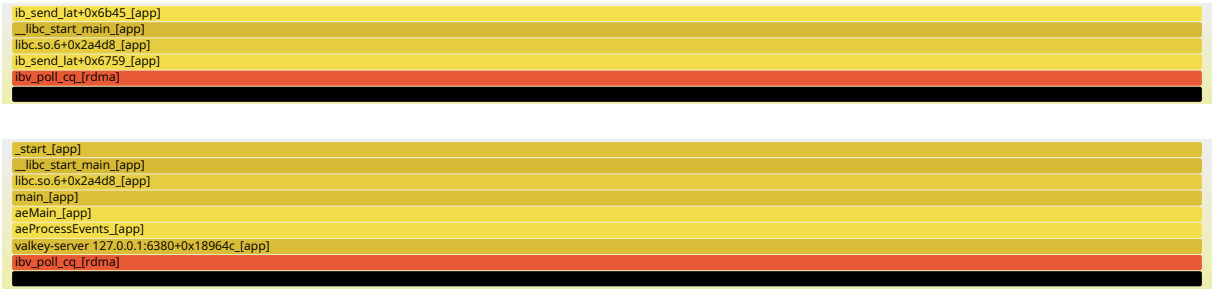


Figure 27: Execution traces (2/2: perftest, Valkey).