Hermes: Accelerating Page Migration and HPC Data Transfers with NoC-attached engines

Francesco Sgherzi
francesco.sgherzi@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

Juan Miguel de Haro Ruiz
juan.deharoruiz@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

Jordi Fornt Mas
jordi.fornt@openchip.com
OpenChip
Barcelona, Spain

Pouya Esmaili Dokht
pouya.esmaili@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

Marco Siracusa
marco.siracusa@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

Ivan Fernandez
ivan.fernandez@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

Miquel Moretó
miquel.moreto@upc.edu
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
Barcelona, Spain

Adrià Armejach
adria.armejach@upc.edu
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
Barcelona, Spain

Abstract

The growing adoption of Heterogeneous Memory Systems, combining DDR, HBM and CXL-attached memory, has made page migration across NUMA nodes an ever growing performance bottleneck. At the same time, High Performance Computing (HPC) workloads relying on MPI communications for data movement, can spend a significant fraction of their execution time in data movement primitives, straining the CPU's ROB and pipeline, trashing cache content without performing useful work. To address these concerns, this paper introduces Hermes, a lightweight NoC-attached engine that accelerates data movement primitives by the preexisting Arm memory operation (MOPS) instructions. Hermes initiates transfers in as few as three instructions, eliminating ROB pressure, and branch mispredictions of software implementations. Hermes requires no programmer intervention, as modern compilers already emit accelerated instructions as substitute for `memcpy`, `memmove` or `memset` libcalls. We implement Hermes in SystemVerilog and synthetize the design in 12nm, demonstrating an area overhead of 1.6% and a maximum power overhead of 1.15% relative to Arm Neoverse N1 cores. Using the gem5 simulator we show that Hermes achieves up to a 9.4× speedup (median 1.28×) in page movement across Heterogeneous NUMA nodes, accelerates data movement MPI operations up to 6.6× (median 1.56×) while consistently improving throughput in individual transfers above 256 B.

CCS Concepts

• **Computer systems organization** → **Multicore architectures**.



This work is licensed under a Creative Commons Attribution 4.0 International License.
PACT '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846871

Keywords

data movement, NoC-accelerators, NUMA, gem5

ACM Reference Format:
Francesco Sgherzi, Juan Miguel de Haro Ruiz, Jordi Fornt Mas, Pouya Esmaili Dokht, Marco Siracusa, Ivan Fernandez, Miquel Moretó, and Adrià Armejach. 2026. Hermes: Accelerating Page Migration and HPC Data Transfers with NoC-attached engines. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '26), October 19-22, 2026, Chicago, IL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3838684.3846871

1 Introduction

The rising employment of Heterogeneous Memory Systems, combining DDR, HBM and Memory-Over-Compute Express Link (CXL), is introducing new challenges in the handling of memory tiers modern server scenarios. While these systems aim at achieving either higher memory bandwidths (DDR-HBM systems [23, 54]) or higher memory capacities (DDR-CXL-attached memory [3, 46]), they introduce a critical bottleneck in the form of page migration across (possibly heterogeneous) Non-Uniform Memory Access (NUMA) nodes. As workloads are increasingly deployed on tiered memory, pages must be moved between devices with vastly different latency and bandwidth characteristics. This concern, however, extends beyond memory tiering. Server applications consistently dedicate between 4%-10% (Figure 1) of their cycles to data movement primitives, with warehouse-scale computers spending a minimum of 5% of their cycles in data transfers [19]. Furthermore, in HPC environments, MPI data transfer operations further compound this issue, as inter-process communication heavily relies on `memcpy` operations that occupy the cores without performing useful computation.

Despite considerable research efforts [10, 18, 20], software implementations of data movement operations still remain, to a great extent, inefficient. Such operations frequently do not exceed 5% (Figure 2) of the system's available bandwidth for transfer sizes that are common in modern workloads, as they saturate core resources

long before hitting a memory bottleneck. In this, hardware accelerators such as the Intel Data Streaming Accelerator (DSA) [23] featured in Sapphire Rapids Xeon processors, attempt to offload such operations to dedicated engines. However, they rely on Direct Memory Access (DMA) mechanisms that require construction and submission of 64 B job descriptors, hence incurring in significant startup and wind-down overheads. As we show in Section 5.2, this overhead renders Intel DSA not applicable to the common use case of small/medium transfers.

In this paper, we propose Hermes, a Network On Chip (NoC)-attached engine for data movement primitives. Unlike DMA-based approach which require the construction of multi-field job descriptors, Hermes requires as little as three instructions to initiate the transfer, and engages directly with the NoC to perform the transfer without requiring further CPU interaction. Hermes leverages the `cpy` and `set` family of instruction, already defined in the Armv8.8 and Armv9.3 ISAs MOPS extension, which modern compilers already emit as drop in replacements for `memcpy`, `memmove`, and `memset`, requiring no code modification. The Hermes engine further supports non-temporal (i.e., cache bypassing) transfers via the Direct Memory Transfer (DMT) mechanism present in any AMBA 5 Coherent Bus Interface (CHI)-compliant NoCs, allowing big transfer to not pollute caches with unneeded data. We further propose asynchronous variants of these instructions, allowing the CPU to either enter a lower power state or to perform other computations.

In summary, we make the following contributions:

- We propose Hermes, a NoC-attached engine for data movement primitives. Hermes leverages preexisting Arm instructions to transparently accelerate `memcpy`s, `memmove`s, `memset`s as well as page movement operations in kernel space.
- We implement Hermes in SystemVerilog and synthesize in 12 nm, demonstrating an area overhead of 1.6% and a maximum power increase of 1.15% relative to an Arm Neoverse N1 CPU.
- We evaluate Hermes on heterogeneous NUMA systems and show that it achieves up to a 9.4× speedup (median 1.28×) in page movement and up to 6.6× speedup (median 1.56×) in MPI operations, considering outperforming state-of-the-art DMA accelerators on the same workloads.
- We characterize the raw throughput of the engine, demonstrating consistent speedup for transfers above 256 B, reaching up to 16× on individual `memcpy` operations, while achieving up to 38% of the peak bandwidth of our simulated system, while software implementations do not exceed 2.8%.

2 Background and Motivation

2.1 Data Movement in Userspace

`memcpy` is the main primitive for copying and moving blocks of memory from one location to another, while `memset` is the main primitive for initializing a region of memory to a given value. Within the `C` standard library, there exist several implementations of such primitives, tailored to leverage specific architectural features. For instance, in `glibc`, the Arm version of `memcpy` has four vector-optimized versions, with the x86_64 version having six [10], to account for the different vector instruction sets supported. Regardless of the implementation the core structure (Algorithm 1),

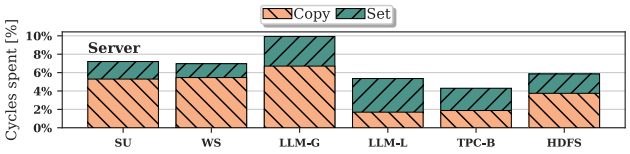


Figure 1: Percentage of cycles spent in data movement, for mobile and server applications.

is shared among all of them. An initial transfer (hereafter referred to as *prologue* - (1)) is performed to align source and destination addresses to word or cache-line size, depending on whether scalar or vector instructions are used. Then, a loop (hereafter referred to as *main loop* - (2)) is performed *iters* times, where *iters* is the ratio between the size to be transferred and the amount of data the instruction used in the loop can transfer. In the loop, scalar or vector instructions are employed, depending on the architecture. Finally, a last transfer (hereafter referred to as *epilogue* - (3)) is performed to complete the missing bytes left out by the rounding down. We note that dividing the computation into these three sections is crucial, as it enables the use of more vectorized data transfer instructions, which require alignment. This structure introduces

Algorithm 1 `memcpy` base algorithm

```
function MEMCPY(destination, source, n)
    N_unaligned ← n mod alignment
    iters ← ⌊n / t_size⌋
    N_remaining ← n − N_unaligned − iters × t_size
    transfer N_unaligned bytes                    ▷ prologue (1)
    for i ∈ [0..iters) do                         ▷ main (2)
        transfer t_size bytes
    end for
    transfer N_remaining bytes                    ▷ epilogue (3)
    return destination
end function
```

several performance pitfalls. Firstly, the *main loop* executes a considerable amount of instructions, thus putting continuous strain on the ROB [19]. This issue is even more pronounced if vector instructions are not used. Secondly, most operations in the *main loop* are loads and stores; hence, the overhead from cache misses and possible evictions on writes considerably hinders performance [18]. This is further exacerbated in small transfer scenarios (Figure 2), where prefetchers are unable to operate at their full potential. Thirdly, caches' Miss Status Holding Registers (MSHRs) are quickly filled by the continuous load requests. Orthogonally, since the main body of data movement primitives contains a loop, the overhead of branch misprediction (Algorithm 1 - (2)) becomes relevant for small data transfers [37, 40]. For instance, a small but common transfer of a 64 B cacheline would still enter a transfer loop, but immediately cause a mispredict in the outcome of the branch and trigger a pipeline flush. These four contributing factors are at the root of the performance pitfalls of data movement primitives, as they either (1) encounter expensive pipeline flushes due to mispredictions on

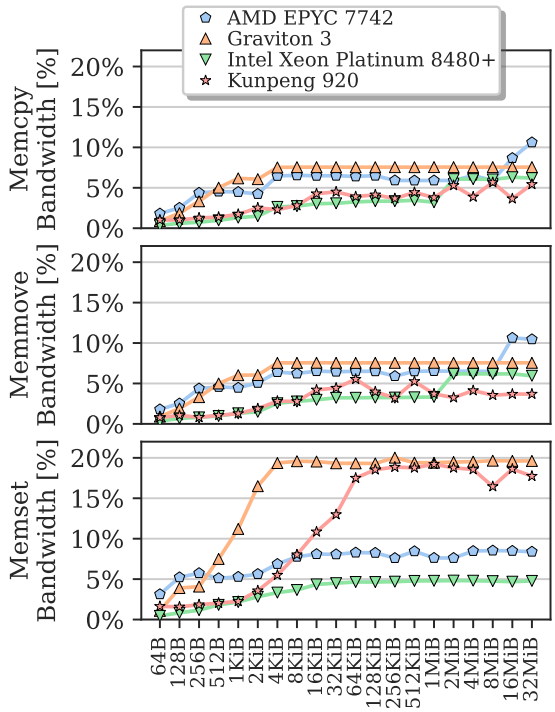


Figure 2: Percentage of peak bandwidth achieved in data transfer at different transfer sizes for `memcpy`, `memmove`, and `memset`.

small transfers, (2) stall the pipeline as new instructions cannot issue (ROB full), or (3) old instructions cannot commit as they are waiting for memory to be available.

Several works have characterized the distributions of sizes of data movement operations [10, 18, 20], establishing that the size of such primitives follows a bimodal distribution, with the peaks both at low and high transfer sizes. While the exact location of such peaks is application dependent, we can assume that most small transfers involve initialization, movement, and copying of data structures (hence, on the order of hundreds of bytes [10]), whereas most large transfers involve I/O operations [33] or page management. For this reason, it is particularly hard to devise a one-size-fits-all approach, as different transfer sizes have different performance pitfalls and optimization objectives.

We quantify these inefficiencies in Figure 2. We profile the impact of spinning up and tearing down docker instances (**SU**) [11], serving a webserver under load (**WS**) [24], loading two Large Language Models (LLMs) with different parameter counts, Gemma2 27B (**LLM-G**) [48] and Llama2 7B (**LLM-L**) [49] and executing the TPC-B benchmark on the PostgreSQL database (**TPC-B**) [26]. Finally, we profile the Apache Hadoop Distributed FileSystem (**HDFS**) [21] while running the NNThroughput benchmark to simulate a filesystem under load on an Intel Xeon Platinum 8480 [15]. These workloads consistently spend between 4% and 10% of their cycles in data movement primitives, with the highest percentages in workloads involving virtualization (**SU**), filesystem management (**LLM-G, HDFS**) and networking (**WS**).

We further benchmark single-core `memcpy`, `memmove`, and `memset` performance across four server platforms, namely AMD EPYC 7742 [4], AWS Graviton 3 [51], Intel Xeon Platinum 8480 [15] and Huawei Kunpeng 920 [53], varying transfer sizes between 64 B to 32 MB, and display the results in Figure 2. From this, we gather the following conclusions. Firstly, we observe that performance of move and set operations scales with the size of the transfers. This is due to both prefetchers being able to work more effectively, and reducing the overhead of branching in the transfer loops and from function calls. Secondly, performance of copy primitives is similar across all platforms, never exceeding 12% of peak theoretical bandwidth. The highest bandwidth utilization is achieved by Graviton 3, reaching $\approx$ 30 GB/s. Still, this equates to not even saturating one of the eight channels that this platform has, thus indicating that the resources of a single core saturate long before hitting a memory bottleneck. Thirdly, Arm CPUs perform considerably better than x86_64 architectures in performing `memset`s, achieving upwards of 21% of peak bandwidth. This stems from both the x86_64 platforms having less efficient write-allocate evasion policies [13, 25] and enforcing Total Store Ordering, compared to the Relaxed Memory Consistency model employed by Arm architectures. From this analysis, we conclude that data movement primitives remain a major source of inefficiency on modern CPUs, particularly for the more common small-to-medium transfers where core resource saturation, not memory bandwidth, is the binding constraint.

2.2 Data Movement in the Linux Kernel

In addition to the usual `memcpy`, `memmove`, and `memset` functions, for which the kernel might employ a different version of the `C` standard library, the kernel uses several other data movement primitives depending on the operation at hand. The `copy_{from,to,in}_user` family of functions is used to move data buffers between user and kernel space, most commonly when a system call occurs [33]. These functions behave similarly to `memcpy`, with few additional checks regarding the permissions of the copied buffers. Being at the heart of most syscall invocations, they are frequently employed, especially in I/O operations (reads/writes to disk and networking [33, 44]).

A second fundamental operation in NUMA systems is page migration across NUMA nodes, performed by the `move_pages` family of functions. These functions move a page from one node to another while preserving the original virtual address. The process faces two main bottlenecks: the kernel overhead of rewriting the virtual-to-physical mapping, and the cost of the underlying data copy, which inherits all the inefficiencies discussed in Section 2.1. In recent years, there has been a noticeable increase in both research efforts [8, 17, 22] and products [3, 38, 41] concerning heterogeneous NUMA systems. This trend is driven by a shift in workloads demanding either much higher bandwidth than the one provided by DDR or much larger memory capacity [12, 29, 31, 36, 39]. The need for large memory pools is further accelerated by the CXL 3.0 specification, which enables memory pooling and sharing across hosts, expanding the scenarios in which pages must migrate between devices. Operating system policies for tiered memory management,

such as Linux's AutoNUMA balancing, DAMON [30] and recently proposed tiered page placement policies, trigger page migrations in response to access pattern changes, making migration frequency a function of workload behavior rather than explicit programmer action. Especially in this latter case, this leads to page movement being a concerning bottleneck, considering that higher density memory devices over CXL generally have higher latency and lower bandwidth than common DDR devices, in addition to being heavily constrained by the interconnect features [46, 47].

2.3 ISA Support for Data Movement

In recent revisions of their ISA, Arm has included the `cpy` [5] and `set` [6] family of instructions (hereafter referred to as `mops`), available in the Armv8.8 and Armv9.3 architectures. While the core still needs to execute the instructions to initiate data movement, the amount of instruction per operation is considerably reduced, requiring just three instructions to initiate a `memcpy` or a `memset` and therefore putting less strain on core resources and ultimately avoiding the dynamic instruction flow that exists in traditional implementations. Furthermore, modern compilers are allowed to inline such instructions in place of the usual data movement primitives. For instance, the `gcc` compiler will inline `mops` starting from optimization level 1 (**-O1**), whereas `clang` would do it starting from optimization level 0 (**-O0**). However, as of today, no publically available Server-grade CPU implements either Armv8.8 or Armv9.3, and the benefits of MOPS remain largely untapped in this scenario.

While there is no direct equivalent in the `x86_64` ISA, Sapphire Rapids CPUs supporting Intel DSA have access to several instructions to submit commands to the accelerators, namely `movdir64b`, `enqcmd`, `enqcmds`. Such instructions forward user created job descriptors, which can trigger, among others, data movement operations. However, to the best of the authors' knowledge, there is no compiler support for such instructions, and using the DSA requires employing the manufacturer's DTO Library [23], which, in turn, use those instructions via several inline assembly calls. This creates a significant usability gap compared to the MOPS approach, where acceleration is transparent to the programmer.

3 Architecture of Hermes

Hermes is a NoC-attached engine able to perform data movement operations. The Hermes engine sits within a Request Node (RN-F) [1] (Figure 3), which hosts a CPU and its private L2 cache. From this, Hermes communicates with Home Nodes (HN-Fs) [1] - hosting directory and System Level Cache (SLC) slices - to issue requests (see Section 3.2). By default, Hermes requires no intervention from the programmer, leveraging preexisting instructions (Table 1 - `cpym`, `cpymnt`, `setm`) in the Arm ISA to accelerate operations both in kernel and user-space. Furthermore, we propose enhancements to the ISA to provide support for asynchronous versions of data movement primitives (Table 1 - `acpym`, `acpymnt`, `asetm`).

3.1 Application Programming Interfaces

We design the Hermes engine to be as transparent to the programmer as possible, leveraging the `mops` family of instructions. For starters, modern compilers are allowed to inline existing `cpy` and `set` instructions in place of `memcpy`, `memset` and `memmove` (from optimization level 0 in `clang` and optimization level 1 in `gcc`). Hence, using the Hermes engine only requires software to be compiled either for Armv8 with the `mops` flag enabled, Armv8.8, or Armv9.3 and no code modification. While there exist compiler intrinsics (i.e., `__builtin_nontemporal_load()` and `__builtin_nontemporal_store()`) that generate non-temporal loads, to the best of the authors' knowledge, there exists no equivalent to output non-temporal transfer `mops` instructions. Hence, software requiring those primitives would need modifications, in the form of inline assembly calls to such instructions. For temporal transfers, the Hermes engine is able to independently write to either private L2 of the calling CPU or to the distributed SLC. To this end, location of the destination data can be configured statically. For what concerns asynchronous operations, we propose three new families of instructions; namely, `acpy`, `acpyn`, and `aset`, operating via interrupts. Hence, the CPU is expected to wait transfer completion using the `wfi` (**w**ait **f**or **i**nterrupt) instruction. The list of supported instructions and their variations is available in Table 2.

While Hermes utilizes the `mops` family of instructions available in the Armv8.8 and Armv9.3 ISAs to initiate data transfers, the engine itself is architecture-agnostic, with the sole requirement being an AMBA CHI 5-compliant NoC.

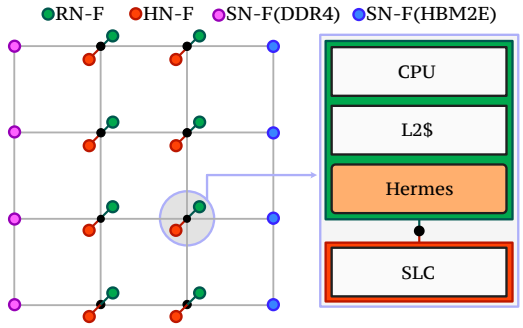


Figure 3: Disposition of Hermes engines within the SoC.

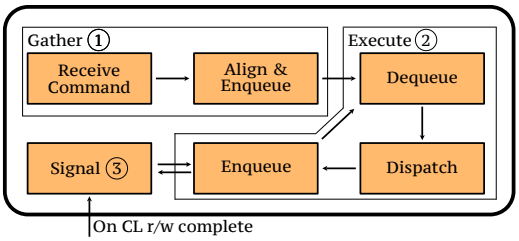


Figure 4: Internal Flow of an operation in the Hermes engine.

3.2 Executing Memory Operations

We outline the functional architecture of Hermes in Figure 5. When the CPU reaches a Hermes-handled operation (Table 1), the instruction executes with *fence* semantics: it proceeds only at the head of the ROB and blocks subsequent instructions until commit (Figure 5 -①). This eliminates the need for commit/rollback mechanisms and considerably simplifies the engine's design. Control passes to

Table 1: Hermes Engine operation description.

Operation	Category	Description	Software Equivalent
Kernel Data Movement	**Copy**	Copy buffers between user and kernel space	`void* copy_{from,to,in}_user(void*, const void*, size_t n)`
Move Page	**Copy**	Move page between NUMA nodes	`long move_pages(int, unsigned long, void*, const int, int, int)`
Copy/Move	**Copy**	Copy n bytes between userspace addresses	`void* std::memcpy(void*, const void*, size_t n)`
Non Temporal Copy/Move	**Copy**	Copy n bytes between userspace addresses, bypass caches	No libc equivalent
Async Copy/Move	**Copy**	Copy n bytes between userspace addresses asynchronously	`auto t = std::thread([&](){std::memcpy(dst, src, n)}); t.join();`
Initialize	**Set**	Write n bytes of a userspace address	`void* std::memset(void*, int, size_t n)`
Async Initialize	**Set**	Write n bytes of a userspace address asynchronously	`auto t = std::thread([&](){std::memset(dst, 'a', n)}); t.join();`

Table 2: ISA primitives accelerated by Hermes. `Xd`, `Xs`, `Xn` refer to destination, source and size respectively. Square brackets indicate that the register contains an address.

Operation	ISA Instruction	Write Destination	Status
Copy	`cpym [Xd], [Xs], Xn`	SLC or L2	Existing
Non-Temporal Copy	`cpymtn [Xd], [Xs], Xn`	Memory	Existing
Set	`setm [Xd], Xn, Xs`	SLC or L2	Existing
Async Copy	`acpym [Xd], [Xs], Xn`	SLC or L2	Proposed
Async Non-Temporal Copy	`acpymtn [Xd], [Xs], Xn`	Memory	Proposed
Async Set	`asetm [Xd], Xn, Xs`	SLC or L2	Proposed

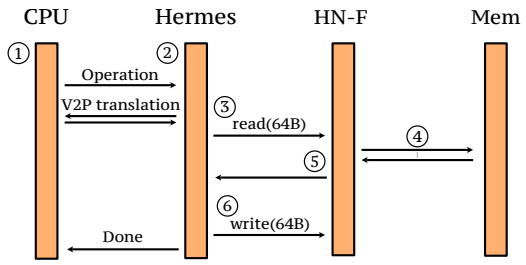


Figure 5: Functional Flow of a Copy operation in Hermes.

Hermes, which requests Virtual-to-Physical translations (Figure 5 (2)) from the calling CPU's L2 TLB, one per cacheline involved in the transfer. The Memory Management Unit (MMU) arbitrates between Hermes and the CPU, prioritizing the latter. By reusing the CPU's MMU, Hermes delegates fault handling entirely to the preexisting CPU's fault mechanisms, so that if this happens, the engine simply stalls until the translation is provided. For `copy` operations, Hermes issues one read per cycle (Figure 5 - (3)) to the HN-F. Depending on coherence state, the request is serviced from the SLC, from another RN-F via Direct Cache Transfer (DCT), or from the memory interface (Figure 5 - (4), (5)). On read completion, a write is issued to L2 or SLC (Figure 5 - (6)), making the cacheline immediately visible across the NoC. For `set` operations, only the write is performed. This loop continues for $(\frac{transfer_size}{cacheline_size})$ iterations.

Non-temporal transfers use two new CHI 5 messages: `NT_START` and `NT_COMPLETE`. After translating and invalidating both source and destination cachelines, Hermes issues `NT_START` to the HN-F, triggering a DMT request that routes data directly between memory interfaces, bypassing caches entirely. The `NT_COMPLETE` message signals transfer completion.

3.3 Hermes Operations

Hermes operates in three pipeline stages (Figure 4):

Gather (Figure 4 - (1)). The engine receives the operation parameters and constructs a Transfer Job containing virtual addresses, transfer size, remaining bytes, and an optional completion signal address. An Align unit splits unaligned requests at cacheline boundaries, producing aligned transfers that proceed in cacheline increments.

Execute (Figure 4 - (2)). A Dequeue unit selects from read and write queues using a round-robin heuristic (up to 16 consecutive operations per queue) to prevent either queue from starving the other. The Dispatch unit forwards requests to the HN-F or private L2. To minimize queue occupancy, at most one cacheline-sized job per operation is enqueued at a time; jobs are re-enqueued with an updated remaining-bytes count until the transfer completes.

Signal (Figure 4 - (3)). On temporal transfers, the Signal unit receives read acknowledgments and enqueues the corresponding writes. On non-temporal transfers, it acknowledges `NT_COMPLETE` messages. When all reads and writes for a Transfer Job have completed, the unit notifies the CPU: synchronous transfers commit the initiating instruction; asynchronous transfers send an interrupt, allowing the core to resume from a low-power `wfi` state.

4 Experimental Setup

4.1 Architectural Simulator

We use the gem5 v22.2 [27] simulator to evaluate an Arm SoC, based on the Neoverse V1 [32] architecture. The network is a mesh NoC, based on the CMN-650 [2] specification which employs the Advanced Microcontroller Bus Architecture (AMBA) CHI 5 [1] protocol. The NoC is designed to support two NUMA nodes, where each node is associated with a specific type of memory technology – one with four channels of DDR5 memory and the other with 8 channels of HBM2E. The Hermes engine is developed within gem5 and faithfully recreates the architecture described in Section 3. In our analysis, we instantiate one engine per core, and each engine communicates directly with the MMU of its corresponding core. A summary of key architectural characteristics of the explored CPU is provided in Table 3.

Finally, to contain simulation time, we model a CPU having 16 Out of Order cores clocked at 3GHz, with prefetchers enabled in L1I, L1D, and L2 caches.

Table 3: Summary of the features of the Simulated System on Chip (SoC) of Figure 3.

Cores	16 Out-of-Order Neoverse N1 Cores (Armv8.2) at 3.0GHz
Vector Units	2 × 256-bit SVE
L1D, L1I per Core	64 kB, 4-way, 2 cycle data access
Private L2 per Core	1 MB, 8-way, 4 cycle data access
Prefetchers	L1I **Tagged**, L1D **Stride**, L2 **Best Offset** [28]
Shared Last Level Cache (LLC)	16 × 1 MB, 16-way, 10 cycle data access, 256 MSHRs per slice
Hermes Engine	1 per RN-F (16 total) at NoC frequency (2 GHz), 16 jobs read / write queues, total 1 KiB storage
Network-on-Chip	4 × 4 2D mesh with AMBA 5 CHI, 1-cycle route and link latency
Memory Technology	DDR5, HBM2E or Memory-over-CXL
Memory Channels	4, 8/1
NUMA Nodes	2
Peak Bandwidths	153.6 GB/s (DDR5), 307.2 GB/s (HBM2E), 43.6 GB/s (CXL)
Operating System	Ubuntu 22.04
Compiler	`g++ 13.1.0`

4.2 RTL Implementation: Area And Power Analysis

We implement Hermes in SystemVerilog using the parameters listed in Table 3. To provide an estimation of the hardware overhead and maximum performance of our design, we synthesize the design in a 12 nm technology (GF12LPPLUS) using Cadence EDA tools. Considering a supply voltage of 0.8 V, our Hermes implementation can run with a clock frequency of up to 2 GHz. The total area of the module is 0.031 mm^2, and its peak power consumption at 2 GHz is 16 mW. To put these values in perspective, adding Hermes into the in-order micro-controller-like core Sargantana [42] CPU results in an area and power overhead (during peak operation, scaled to 1.2 GHz - frequency of Sargantana) of 4.4% and 11%, respectively. On the Arm realm, adding Hermes to the Cloud-optimized Neoverse-N1 core results in an area increase of 1.6% and a power increase of 1.15%, when scaled to 2.6 GHz and 7nm [32, 43].

4.3 Benchmarks

Firstly, we test the capabilities of Hermes in accelerating page movement. To this end, we devise a benchmark invoking the `move_pages` syscall. Similarly to the previously described data movement benchmarks, a number of pages specified as program argument are allocated in source and destination NUMA nodes using the `numa_alloc_onnode` function. These pages are then initialized by the core to prevent `move_pages` syscall failure, which occurs if pages have not been touched by the calling CPU. Finally, the specified number of pages is moved back and forth between the NUMA nodes. We benchmark page movement at a varying amount of pages transferred (from 1 to 1024), varying page size (4 KiB, 64 KiB), and across different memory interfaces. To this end, we leverage the HBM2E model provided by Qureshi et al. [34] and Memory-Over-CXL, leveraging the Mess simulator integration for gem5, as provided by Esmaili-Dokht et al. [13] with Bandwidth–Latency curves for Memory-Over-CXL provided by Micron, a major memory manufacturer. Both HBM and Memory-Over-CXL are exposed to the operating system as separate NUMA nodes, thus further replicating the behavior of heterogeneous memory systems at kernel level.

Secondly, we evaluate the performance of our engine in workloads representing HPC scenarios and compare the relative performance improvement against the Intel Xeon MAX 9480 CPU. This CPU supports Intel DSA, an in-network accelerator [23] tailored towards, among others, `memcpy` and `memsets` operations. This allows us to test how hardware accelerators for memory operations interact with extremely high amounts of bandwidth available (HBM). For what concerns HPC operations, we employ the Intel MPI Benchmarks [16] suite of benchmarks, whose goal is to provide microbenchmarks for the most common operations found in MPI workloads (e.g., Gather, Scatter, Reduce). Furthermore, the benchmark suite allows to specify the size of messages exchanged between MPI processes, allowing to assess potential benefits of acceleration at specific amounts of data exchanged. In our analysis, we run all the benchmarks for `MPI1`-type operations, while varying the message size in power-of-two increments from 64 B to 8 MB.

Thirdly, to characterize the throughput of the Hermes engine `memcpy` and `memset` functions on a single core. The benchmark allocates and initializes 256 MiB buffers as destination and source in the case of memcpy. Then it performs data movement operation in $CHUNK_SIZE$ steps, specified as a program argument while skipping a page length worth of addresses between operations. This loop continues for $\frac{256\,\text{MiB}}{\texttt{stride}}$ iterations, where `stride` is defined to be $max(PAGE_SIZE, CHUNK_SIZE)$. In addition to this, the loop is unrolled by a factor of 16 to minimize the overhead of branching and the eventual misprediction. We employ this benchmark for three basic implementations: (i) software (i.e., `libc` data movement operations, supporting SVE instructions), (ii) Hermes in synchronous mode, and (iii) Hermes in asynchronous mode.

Finally, we test how performance scales with an increasing number of Hermes engines active. To this end, we benchmark a `memcpy` operation (since it comprises both reads and writes) in moving data totaling 80% of the available Dynamic RAM (DRAM). Such work is then distributed across multiple threads (1, 2, 4, 8, or 16), and the performance is evaluated.

4.4 Leveraging Accelerated Data Movement

For what concerns our Simulated SoC, enabling architectural support for data movement is straightforward. As outlined in Section 1, modern compilers would inline `cpy` and `set` instructions in place of synchronous memory operations - assuming the appropriate optimization level is selected. At the time of writing this document, the gem5 simulator supports up to the Armv8.2 ISA, hence, we compile our code using the `+mops` feature flag (`-mcpu=armv8.2-a+mops` in gcc). Conversely, for instructions not available in the `mops` family (i.e., `acpy`, `aset`) we insert inline assembly in the source code.

For what concerns the Intel Max 9480 CPU, using the DSA accelerator is not as straightforward as its Arm counterpart. For starters, while communication with the accelerator is achieved via instructions (similarly to Arm), to the best of the authors' knowledge,

there is no compiler support for the generation of such instructions. This is furthermore corroborated by the fact that the manufacturer supported method of engaging the DSA relies on the DTO library [23], which, upon inspection, relies on several `asm(.word <instruction bytes>)` calls. Nonetheless, using accelerated functions in place of their regular `libc` equivalent requires either compiling the code with the `-ldto` option, or loading the library via `LD_PRELOAD`. We enable all the engines present in our Intel MAX (4) to allow the DTO library to parallelize data movement operations among different devices, along with 4 work queues of depth 128, as suggested by the documentation of the Intel DTO Library.

5 Experimental Results

We organize the evaluation around the following three questions: (i) how does Hermes accelerate page migration in heterogeneous NUMA systems? (ii) would Hermes improve the performance of HPC workloads that rely on MPI data movement, and how does it compare to Intel DSA? (iii) What are the raw throughput characteristics of the engine, and where do its performance boundaries lie? We address these in turn.

5.1 Page Movement

We start our evaluation by determining the capabilities of Hermes in accelerating page movement across NUMA domains. We evaluate this by invoking the `move_pages` syscall, varying the number of page transferred (1 to 1024), the kernel page size (4 kB, 64 kB) and memory devices (DDR5+HBM2, DDR5+CXL). We display the speedup in moving pages with respect to the non-accelerated version in Figure 6.

Firstly, Hermes consistently improves performance, regardless of page size, number of pages moved or memory interfaces involved in the movement. As outlined in Section 2, the `move-pages` syscall effectively performs a copy between source and destination nodes, before rewiring addresses to point to the same virtual address. Hence, copying the content of the page to a new location constitutes the dominant cost, especially when few pages are moved. Since the minimal size of page copy is 4 KiB, Hermes is well within the range where it outperforms the software baseline (Figure 8).

Secondly, for 4 KiB pages, Hermes provides moderate but steady improvements, reaching up to 1.18× on DDR5+CXL at 1024 pages. The speedup is bounded by the fact that each individual page copy involves only 4 KiB of data, a transfer size where Hermes's capabilities are shadowed by the overhead of PTE invalidation and TLB shootdown within the kernel. Nonetheless, CXL configurations consistently yield slightly higher speedups than HBM ones, as Hermes' ability to sustain more outstanding memory requests allow it to better hide the higher access latency of CXL-attached memories.

Thirdly, this picture changes considerably with larger pages. At 64 KiB per transfer, Hermes sits well into the regime in which it is able to achieve maximum throughput (Figures 8 and 9). Due to this and the overhead of auxiliary kernel operations lessening, Hermes achieves up to 3.67× speedup when transferring pages between DDR and HBM and up to a 9.38× speedup when moving pages between DDR and Memory-Over-CXL. Furthermore, Hermes achieves considerable speedups (up to 2×) even at a relatively small amount of pages transferred in batch (<32), which are the most representative amounts of pages transferred by online migration policies such as AutoNUMA or DAMON. As with 4 KiB pages, the DDR+CXL configuration yields equal or higher speedups than DDR+CXL at all operating points, outlining the latency-hiding capabilities of Hermes. This property makes Hermes particularly relevant for emerging tiered memory deployments, where page migration between fast local DDR and slower CXL tiers is a routine operation triggered by kernel policies such as AutoNUMA and DAMON.

Recent work [7] has explored integrating Intel DSA into the Linux kernel's `migrate_pages()` path. Their approach modifies the kernel to batch page copy submissions to the accelerator, amortizing DSA's startup overhead across multiple pages. While batching can yield competitive throughput when migrating large numbers of pages simultaneously, it requires kernel-level modifications and remains constrained by DSA's per-submission overhead at smaller batch sizes, precisely the operating regime of online migration policies such as AutoNUMA and DAMON, where pages are typically promoted or demoted in small batches in response to changing access patterns. In contrast, Hermes' negligible initiation cost (three instructions, no descriptor construction) provides consistent benefits across all batch sizes without any kernel modification: any kernel compiled for Armv8.8 or Armv9.3 is transparently accelerated. Furthermore, our evaluation employs realistic CXL memory modeling based on manufacturer-provided bandwidth-latency curves [13], whereas prior work relies on frequency-throttled NUMA emulation to approximate CXL behavior.

5.2 Intel MPI Benchmarks

We evaluate Hermes on the Intel MPI benchmark suite [16], running all MPI1-type operations on all 16 cores of our simulated SoC. Each benchmark is executed across 17 message sizes (64 B to 8 MiB, in power of two increments). For clarity, Figures 7a and 7b, report the median execution time of the accelerated implementation, normalized to the non-accelerated one. The shaded portion of each bar represents the fraction of execution time spent in the accelerated memory operation.

Figure 7a shows that Hermes accelerates the majority of MPI operations under test. The most pronounced improvements occur within collective operations involving gather and broadcast patterns, where ranks spend a considerable part of the execution time copying buffers in and out the MPI internal staging areas. Alltoallv (**A2AV**) reduces to roughly 15% of the baseline execution time, with the median performance (Gather - **G**, Bcast - **BC**, SendRecV - **SRV**, Allgather - **AG**, Exchange **EX**) being around 50-65% of the unaccelerated execution time. The fraction of time attributable to Hermes itself also varies considerably across benchmarks, with peaks at ≈ 40% where Hermes is on the critical path for the majority of the work.

A small subset of benchmarks show limited to no improvement. For instance, Barrier (**B**) is dominated by synchronization with negligible data movement and, similarly, Reduce-local (**RL**) hover nearly at the same performance as the unaccelerated version, with Hermes occupying a negligible amount of the execution time.

Across the suite, Hermes achieves a median speedup of ≈ 1.56 ×, with peak speedups exceding 6.6× on collective patterns that

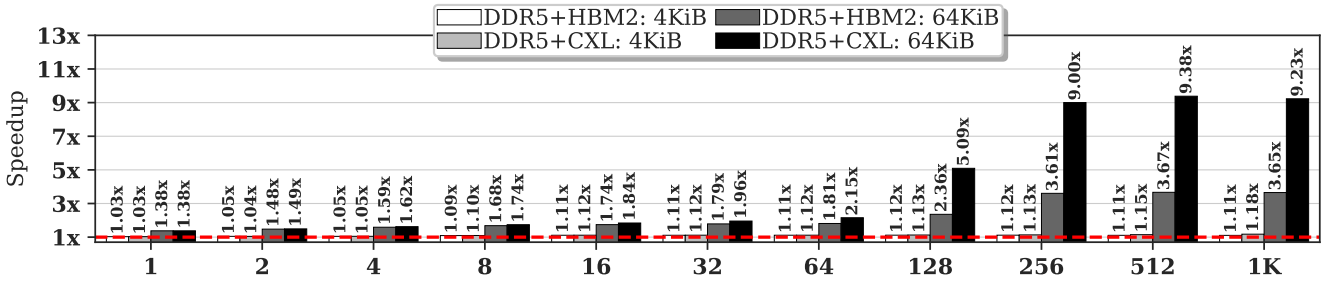


Figure 6: Speedup w.r.t. non accelerated simulated CPU in moving 4 KiB and 64 KiB pages across different memory technologies.

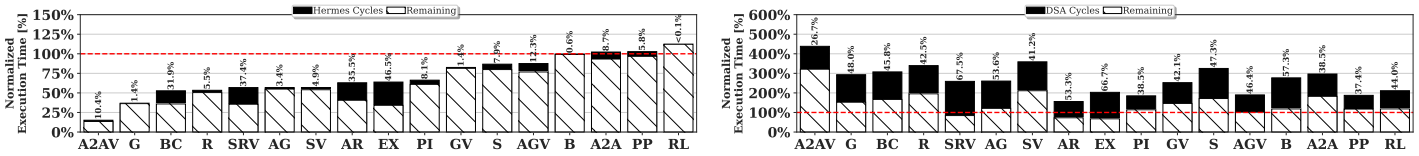


Figure 7: Median normalized execution time for Intel MPI benchmarks w.r.t software baseline for (a) Hermes and (b) Intel DSA. Shaded area is the time spent in accelerated memory operations.

excercise large buffer transfers. This indicates that Hermes is well suited for accelerating common MPI operations in HPC workloads.

We compare against Intel DSA on the Intel Xeon MAX 9480 platform, using the manufacturer DTO library to intercept `memcpy`, `memset` and `memmove` calls in Figure 7b. In contrast to Hermes, Intel DSA consistently degrades performance across the entire suite. Slowdowns reach 3-5×, on several benchmarkks, including Allgatherv (**A2AV**), SendRecV (**SRV**) and Scatter (**S**). More importantly, the amount of time spent in the accelerated portion account from ≈26% to ≈ 67% of the time, indicating that the DSA is actively part of the bottleneck for such operation. The combination of high invocation rate and the high start overhead implies that the per-call overhead of DSA exceeds any throughput gain from offloading the actual transfer. As outlined in Section 4.4, each DSA transfer requires the construction and subission of a 64 B job descriptor, forllowed by a wait for completion (typically signalled via either interrupt or polling). This setup and teardown overhead is typically only amortized for large transfers [23], which might be infrequent even on MPI operations exchanging big messages since copies of too big of messages can be split into multiple ones.

With Hermes employing no descriptor construction, no DMA setup, and no completion interrupt for synchronous transfer, the per-call overhead is bounded purely by the latency of the three instructions traversing the pipeline, allowing Hermes to remain effective across the full range of MPI sizes, while DSA is restricted to a narrow regime of large transfers in which its setup cost can be amortized.

5.3 `memcpy` and `memset` performance

We present the speedup of the `memcpy` operation with respect to the `glibc` software implementation on the simulated SoC in Figure 8.

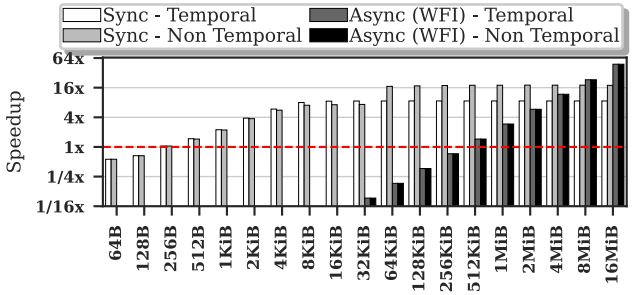


Figure 8: Speedup w.r.t. non accelerated simulated CPU in `memcpy`, varying size of transfer and mode of execution.

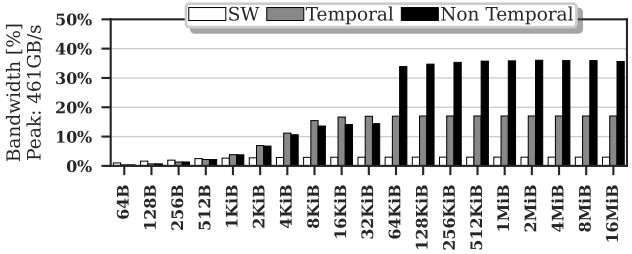


Figure 9: Percentage of peak bandwidth for synchronous `memcpy`, varying size of transfer and mode of execution.

From this, we gather the following insights. Hermes in synchronous

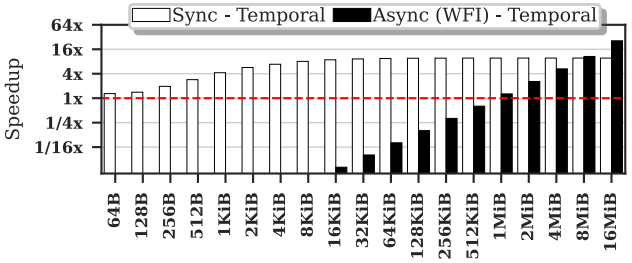


Figure 10: Speedup w.r.t. non accelerated simulated CPU in `memset`, varying size of transfer and mode of execution.

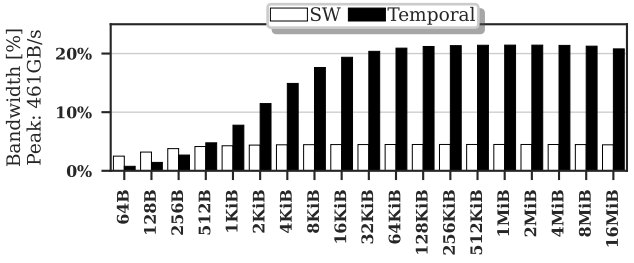


Figure 11: Percentage of peak bandwidth in synchronous `memset`, varying size of transfer and mode of execution.

mode has better performance than the pure software implementation starting from 256 B. This is due to the fact that, in order to prevent data from being written at the source and destination locations while the cpy instruction is executing, we require the instruction to execute with *fence* semantics, thus executing only if it is at the head of the ROB and preventing the insertion of other instructions before termination. Such overhead lessens after 256 B, providing consistent speedup up to 16× for transfers bigger than 64 KiB, at saturation of the R/W bandwidth of the NoC connection to Hermes.

While asynchronous copies incur in considerable overhead for small transfers, they perform better on Hermes starting at 512 KiB, and they exceed the speedup provided by synchronous copies after 8 MiB, up to a maximum of ≈ 102×. Differently from synchronous copy, we allow the acpy instruction to commit immediately and we leave the responsibility to the programmer to appropriately wait for completion using the `wfi` instruction.

We also observe that the speed of non-temporal operations exceeds the one of temporal operations starting from 64 KiB. The reason behind this is two-fold. First, non-temporal operations require the Hermes engine only for initiating the transfer and performing invalidation of the cachelines to be written; hence, after this has been performed, the transfer only involves memory interfaces. Furthermore, non-temporal transfers do not need to pass through the engine for writes. This halves the outward bandwidth required from the RN-F and allows transfers to proceed at essentially double the throughput of the temporal version. We further analyze this in Figure 9, where we display the bandwidth requested to the memory interfaces across modes of execution. First, we determine that the bandwidth of the baseline software implementation never exceeds 3% (≈ 13 GB/s) of peak in our system. Second, we notice that the bandwidth of accelerated implementations steadily increases, with non-temporal operations slightly trailing behind due to additional messages (see Section 3.2) being sent throughout the NoC. Finally, as expected, the bandwidth usage of non-temporal operations is essentially double the one of temporal operations after the 64 kB mark.

Figure 10 presents Hermes speedup of the memset operation with respect to the `glibc` version. From this, we gather the following insights. In contrast to the copy, the engine provides speedup starting from 64 B. This is due to the fact that initializing a region of memory has half the number of operations than copying regions

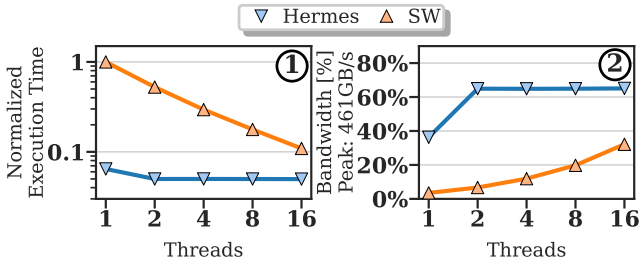


Figure 12: Scalability to multiple Hermes engines. ① shows the execution time normalized to the single core software implementation, at varying number of threads active. ② shows the throughput, at varying number of threads active.

of memory; therefore, the time the pipeline is stalled due to fence semantics is shorter, allowing execution to resume earlier. For the same reason, the peak expected speedup is lower than the copy operation. While the engine works at the same rate regardless of the operation, the CPU software version executes half of the operations, thus putting less pressure on the cores' resources.

Secondly, similarly to copy operations, speedup plateaus at the 16 KiB mark. This is due to saturation of the NoC link towards cache devices. This is further corroborated by Figure 11, where, starting from 16 KiB, bandwidth usage does not increase after hitting the plateau point. Albeit bandwidth usage of software `memset` is essentially double that of software `memcpy`, we still achieve upwards of 7× more bandwidth usage.

Finally, asynchronous set operations follow a similar distribution to asynchronous copy operations, with the overhead of interrupts being overcome at the 1 MiB mark.

5.4 Scaling to Multiple Engines

We present the scalability of the Hermes engine in Figure 12. Firstly, as the evaluated transfer accounts for 80% of the available DRAM capacity (totalling ≈ 410 MB), we note that all Hermes engines receive a minimum of ≈ 25 MB to transfer, thus operating in the optimal range (Figures 8 and 9). Secondly, we note that execution times (1) do not improve noticeably when more than 2 Hermes engines are active at the same time. We further explore this in (2),

where we achieve 62% of the peak bandwidth of the system with 2 engines active, after which saturation of the NoC links occurs in our SoC. Finally, software `memcpy` requires 16 hardware threads to equate the throughput provided by a single Hermes engine, which further corroborates the result seen in Section 5.3.

Overall, these results outline a deeper matter in blindly substituting `mops` in place of data movement operations while ignoring transfer size. Regardless of the implementation, `mops` instructions must execute with at least partial *fence* semantics (i.e., at least blocking younger loads and stores from executing), to guarantee consistency. If this were not to be the case, a CPU executing Out-of-Order might modify data contained at the addresses involved in the transfer while the `mops` operation is executing, as CPU's functional units cannot track dependencies on data contained within a range of virtual addresses. Hence, having reduced performance in byte-sized transfers is a consequence that would accompany every implementation of such instructions. Nonetheless, hardware acceleration for data primitives shows very promising results in workloads that lean on the heavier side for transfer sizes, as we have shown in Section 5.2.

6 Related Work

In this section, we provide an analysis of the most prominent efforts in software optimizations, DMA accelerators, and in-memory accelerators. A qualitative comparison of the state-of-the-art is available in Table 4.

Software Optimizations: `glibc` already provides several implementations for data movement primitives, which are chosen at compile time by the `glibc` distributor, depending on the target architecture. Kappes et al. [20] propose a framework for optimizing `memcpy`, based on creating several versions of data movement primitives based on different combinations of vector length and temporal or non-temporal loads and stores. They enumerate all possible combinations to determine which instance performs best based on a certain transfer size. Similarly, Chatelet et al. [10] extend this idea to `memset` as well to achieve upwards of 2× over `glibc`. While this demonstrates that pure software approaches can obtain considerable speedup, when optimizing for architectures and transfer sizes, they still incur in most of the bottlenecks outlined in Section 2.1, which Hermes is able to hide.

Dedicated Accelerators: In the recent Sapphire Rapids architecture, Intel provides the DSA [23], a device capable of enhancing several operations that constitute the datacenter tax [19]. This accelerator sits within the processor network, and communicates with the host CPU via memory-mapped registers, referred to as *portals*. Portals receive 64 B descriptors for the job to be processed, and submit them to *work queues*, to be read by the DSA to perform the requested operation. Via this mechanism, Intel DSA is able to support both synchronous and asynchronous operations.

Communicating with the DSA is performed either directly via instructions submitting a user-created job descriptor (i.e., `movdir64b`, `enqcmd`, `enqcmds`) or transparently (assuming DSAs and *work queues* have been configured beforehand) via the DTO library. The DTO library can be used both without recompiling preexisting software (via `LD_PRELOAD`) or via linking (using the `-ldto` linker flag). This library intercepts, among others, `memcpy` and `memset` operations and submits them to the DSA. While they provide multiple DSAs per package, the accelerators are unable to perform data movement operations while saturating the memory system. Specifically, peak throughput per accelerator plateaus at ≈ 30 GB/s, while incurring considerable startup overhead. As we show in Section 5.2, this renders the accelerator mostly applicable to the less common bigger transfers, while leading to considerable slowdowns for `memsets` below 16 kB, and `memcpys` below 4 kB [23]. Wong et al. [50, 52] propose a dedicated in-cache accelerator for copy operations that is able to perform data movement lazily, by keeping track of the regions of memory that are supposed to be moved and performing the actual copy solely on write. However, their approach is limited by the fact that both source and destination need to be cacheline aligned. Su et al. [45] propose a DMA based accelerator between SLC and main memory. This accelerator supports both synchronous and asynchronous memcpy operations. While they achieve considerable speedups with respect to Intel I/OAT (up to 10×), the initialization incurs significant DMA overhead, making it suitable only for relatively large (1 kB) transfers.

Overall, these approaches are plagued by the overhead of starting up DMA devices, thus rendering them usable only for bigger transfers. Furthermore, they require heavy intervention from the programmer and the use of specific libraries, compromising usability.

In-memory Data Movement: There exists a substantial body of work that leverages the capabilities of internal DRAM structures to perform memory copies at the DRAM level [9, 14, 35]. While these research efforts are able to provide considerable (up to 9×) speedup with respect to software implementations, they are constrained by the fact that source and destination buffers must sit in the same DRAM device. More recently, Kamath et al. [18] proposed a mechanism to perform `memcpy` lazily at the memory controller level. $(MC)^2$ relies on a new set of instructions to initiate copies. Once the instruction is executed, cachelines referring to source and destination buffers are invalidated and written to memory. The actual copy is not performed until new data is accessed. While this approach is promising, it is not transparent, and only supports non-temporal copy operations.

To the best of the authors' knowledge, Hermes is the first hardware accelerator that is able to (1) perform synchronous and asynchronous data movement operations; (2) support non-temporal copies; (3) is transparent to the programmer, i.e., a drop-in replacement to current data movement primitives; (4) is able to perform data movement operations leveraging high transfer rates from the

Table 4: Qualitative comparison with related work.

Features	Software optimizations [10, 20]	Dedicated accelerators [23, 45, 50, 52]	In-memory Data Movement [9, 14, 18, 35]	Hermes engine
Reduces ROB fills		✓	✓	✓
Reduces I-cache misses		✓	✓	✓
Reduces pipeline flush		✓	✓	✓
Improves bandwidth usage	✓	✓	✓	✓
Linux Kernel Support	✓			✓
Asynchronous offloading	✓	✓	✓	✓
Temporal & Non-Temporal	✓	✓		✓
Transparent to the programmer	✓			✓

memory interfaces; and (5) accelerates data movement operations for both kernel and user space.

7 Conclusions

In this work we present Hermes, a NoC-attached engine for data movement primitives. Hermes reduces latency and core stall time for mid-to-large (256 B) transfers — common in page migration and MPI communication — by offloading them to a dedicated engine within each RN-F, freeing core resources and leveraging otherwise underutilized memory bandwidth. The engine supports temporal and non-temporal transfers (avoiding cache pollution on large copies), asynchronous operation (allowing cores to perform other work or enter low-power states), and integrates transparently with the Arm `mops` family of instructions, requiring no code modification. > Our evaluation shows that Hermes (1) achieves up to 9.4× speedup in page migration across heterogeneous NUMA nodes, (2) accelerates MPI operations by up to 6.6× where Intel DSA instead incurs slowdowns, and (3) delivers consistent throughput improvements on individual `memcpy` and `memset` operations above 256 B all at an area cost of 1.6% relative to an Arm Neoverse N1 core.

Acknowledgments

This work is supported by the project PID2023-146511NB-I00 funded by the Spanish Ministry of Science, Innovation and Universities MCIU/AEI 10.13039/501100011033 and EU ERDF; by the Ministry for Digital Transformation and Public Service through the Càtedra Chip UPC project [grant number TSI-069100-2023-0015] and within the framework of the Recovery, Transformation and Resilience Plan [NextGenerationEU REGAGE22e00058408992]; by the Generalitat of Catalunya through contract [2021-SGR-00763], by the Arm-BSC Center of Excellence and by the European Processor Initiative (EPI) under grant agreement No. 101036168. A. Armejach is a Serra Hunter Fellow.

References

[1] [n. d.]. AMBA CHI Architecture Specification. https://developer.arm.com/documentation/ihi0050/latest/. (Accessed on 07/30/2024).

[2] [n. d.]. Arm Neoverse CMN-650 Coherent Mesh Network Technical Reference Manual r2p0. https://developer.arm.com/documentation/101481/0200/What-is-CMN-650-/About-CMN-650. (Accessed on 07/30/2024).

[3] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. 2022. Enabling CXL memory expansion for in-memory database management systems. In *Proceedings of the 18th International Workshop on Data Management on New Hardware*. 1–5.

[4] AMD. 2020. *High Performance Computing: Tuning guide for AMD EPYC 7002 Series Processor*. A. Kashyap.

[5] Arm. 2024. CPY Instruction Reference. https://developer.arm.com/documentation/ddi0602/2024-03/Base-Instructions/CPYP--CPYM--CPYE--Memory-Copy-. Accessed: 2024-07-07.

[6] Arm. 2024. SET Instruction Reference. https://developer.arm.com/documentation/ddi0602/2024-03/Base-Instructions/SETP--SETM--SETE--Memory-Set-. Accessed: 2024-07-07.

[7] Jongho Baik, Jonghyeon Kim, Chang Hyun Park, and Jeongseob Ahn. 2025. Accelerating Page Migrations in Operating Systems With Intel DSA. *IEEE Computer Architecture Letters* 24, 1 (2025), 37–40. https://doi.org/10.1109/LCA.2025.3530093

[8] Evgeny Bolotin, David Nellans, Oreste Villa, Mike O'Connor, Alex Ramirez, and Stephen W. Keckler. 2015. Designing Efficient Heterogeneous Memory Architectures. *IEEE Micro* 35, 4 (2015), 60–68. https://doi.org/10.1109/MM.2015.72

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

[10] Guillaume Chatelet, Chris Kennelly, Sam Xi, Ondrej Sykora, Clement Courbet, David Li, and Bruno De Backer. 2021. automemcpy: A framework for automatic generation of fundamental memory operations. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management (ISMM '21), June 22, 2021, Virtual, Canada*.

[11] Inc Docker. 2020. Docker. *linea].[Junio de 2017]. Disponible en: https://www.docker. com/what-docker* (2020).

[12] Douglas Doerfler, Farzad Fatollahi-Fard, Colin MacLean, Tan Nguyen, Samuel Williams, Nicholas Wright, and Marco Siracusa. 2021. Experiences Porting the SU3_Bench Microbenchmark to the Intel Arria 10 and Xilinx Alveo U280 FPGAs. In *Proceedings of the 9th International Workshop on OpenCL* (Munich, Germany) *(IWOCL '21)*. Association for Computing Machinery, New York, NY, USA, Article 1, 9 pages. https://doi.org/10.1145/3456669.3456671

[13] Pouya Esmaili-Dokht, Francesco Sgherzi, Valeria Soldera Girelli, Isaac Boixaderas, Mariana Carmin, Alireza Monemi, Adria Armejach, Estanislao Mercadal, German Llort, Petar Radojkovic, Miquel Moreto, Judit Gimenez, Xavier Martorell, Eduard Ayguade, Jesus Labarta, Emanuele Confalonieri, Rishabh Dubey, and Jason Adlard. 2024. A Mess of Memory System Benchmarking, Simulation and Application Profiling. In *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[14] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. 2019. Computedram: In-memory compute using off-the-shelf drams. In *Proceedings of the 52nd annual IEEE/ACM International Symposium on Microarchitecture*. 100–113.

[15] Intel. 2023. Intel Xeon CPU Max Series Configuration and Tuning Guide. https://www.intel.com/content/www/us/en/content-details/769060/intel-xeon-cpu-max-series-configuration-and-tuning-guide.html.

[16] Intel. 2024. Intel MPI Benchmarks. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-mpi-benchmarks.html. Accessed: 2024-07-10.

[17] Xiaojuan Jing and Haixiang Li. 2022. Construction and optimization of heterogeneous memory system based on NUMA architecture. In *2022 7th International Conference on Intelligent Computing and Signal Processing (ICSP)*. 767–772. https://doi.org/10.1109/ICSP54964.2022.9778754

[18] Aditya K Kamath and Simon Peter. 2024. (MC)2: Lazy MemCopy at the Memory Controller. (2024).

[19] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd annual international symposium on computer architecture*. 158–169.

[20] Giorgos Kappes and Stergios V Anastasiadis. 2021. Asterope: A cross-platform optimization method for fast memory copy. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*. 9–16.

[21] A Kala Karun and K Chitharanjan. 2013. A review on hadoop—HDFS infrastructure extensions. In *2013 IEEE conference on information & communication technologies*. IEEE, 132–137.

[22] Hamidreza Khaleghzadeh, Ravi Reddy Manumachu, and Alexey Lastovetsky. 2020. A Hierarchical Data-Partitioning Algorithm for Performance Optimization of Data-Parallel Applications on Heterogeneous Multi-Accelerator NUMA Nodes. *IEEE Access* 8 (2020), 7861–7876. https://doi.org/10.1109/ACCESS.2019.2959905

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

[24] Grafana Labs. 2024. Grafana Labs - K6. https://k6.io/. Accessed: 2024-07-07.

[25] Jan Laukemann, Thomas Gruber, Georg Hager, Dossay Oryspayev, and Gerhard Wellein. 2024. CloverLeaf on Intel Multi-Core CPUs: A Case Study in Write-Allocate Evasion. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 350–360.

[26] Martin Lindblom and Fredrik Strandin. 2015. Performance analysis and improvement of PostgreSQL. (2015).

[27] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. 2020. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152* (2020).

[28] Pierre Michaud. 2016. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 469–480.

[29] Tan Nguyen, Colin MacLean, Marco Siracusa, Douglas Doerfler, Nicholas J. Wright, and Samuel Williams. 2022. FPGA-based HPC accelerators: An evaluation on performance and energy efficiency. *Concurrency and Computation: Practice and Experience* 34, 20 (2022), e6570. https://doi.org/10.1002/cpe.6570 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.6570

[30] SeongJae Park, Yunjae Lee, and Heon Y. Yeom. 2019. Profiling Dynamic Data Access Patterns with Controlled Overhead and Quality. In *Proceedings of the 20th International Middleware Conference Industrial Track* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3366626.3368125

[31] Alberto Parravicini, Luca Giuseppe Cellamare, Marco Siracusa, and Marco D Santambrogio. 2021. Scaling up HBM efficiency of Top-K SpMV for approximate

embedding similarity on FPGAs. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 799–804.

[32] Andrea Pellegrini, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, Tushar Ringe, Ashok Tummala, et al. 2020. The arm neoverse n1 platform: Building blocks for the next-gen cloud-to-edge infrastructure soc. *IEEE Micro* 40, 2 (2020), 53–62.

[33] Amit Purohit, Joseph Spadavecchia, Charles Wright, and Erez Zadok. 2003. *Improving application performance through system call composition.* Technical Report. Citeseer.

[34] Yasir Mahmood Qureshi, William Andrew Simon, Marina Zapater, Katzalin Olcoz, and David Atienza. 2021. Gem5-X: A Many-core Heterogeneous Simulation Platform for Architectural Exploration and Optimization. *ACM Transactions on Architecture and Code Optimization* 18, 4, Article 44 (jul 2021), 27 pages. https://doi.org/10.1145/3461662

[35] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, et al. 2013. RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 185–197.

[36] Francesco Sgherzi, Alberto Parravicini, Marco Siracusa, and Marco D. Santambrogio. 2021. Solving Large Top-K Graph Eigenproblems with a Memory and Compute-optimized FPGA Design. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 78–87. https://doi.org/10.1109/FCCM51124.2021.00017

[37] Francesco Sgherzi, Marco Siracusa, Ivan Fernandez, Adrià Armejach, and Miquel Moretó. 2024. SpChar: Characterizing the sparse puzzle via decision trees. *J. Parallel and Distrib. Comput.* 192 (2024), 104941. https://doi.org/10.1016/j.jpdc.2024.104941

[38] Nikolay A Simakov, Matthew D Jones, Thomas R Furlani, Eva Siegmann, and Robert J Harrison. 2024. First Impressions of the NVIDIA Grace CPU Superchip and NVIDIA Grace Hopper Superchip for Scientific Workloads. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops*. 36–44.

[39] Marco Siracusa, Emanuele Del Sozzo, Marco Rabozzi, Lorenzo Di Tucci, Samuel Williams, Donatella Sciuto, and Marco Domenico Santambrogio. 2022. A Comprehensive Methodology to Optimize FPGA Designs via the Roofline Model. *IEEE Trans. Comput.* 71, 8 (2022), 1903–1915. https://doi.org/10.1109/TC.2021.3111761

[40] Marco Siracusa, Víctor Soria-Pardos, Francesco Sgherzi, Joshua Randall, Douglas J. Joseph, Miquel Moretó Planas, and Adrià Armejach. 2023. A Tensor Marshaling Unit for Sparse Tensor Algebra on General-Purpose Processors. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 1332–1346. https://doi.org/10.1145/3613424.3614284

[41] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights landing: Second-generation intel xeon phi product. *Ieee micro* 36, 2 (2016), 34–46.

[42] Víctor Soria-Pardos, Max Doblas, Guillem López–Paradís, Gerard Candón, Narcís Rodas, Xavier Carril, Pau Fontova–Musté, Neiel Leyva, Santiago Marco-Sola, and Miquel Moretó. 2022. Sargantana: A 1 GHz+ In-Order RISC-V Processor with SIMD Vector Extensions in 22nm FD-SOI. In *2022 25th Euromicro Conference on Digital System Design (DSD)*. 254–261. https://doi.org/10.1109/DSD57027.2022.00042

[43] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration* 58 (2017), 74–81.

[44] Jingbo Su, Jiahao Li, Luofan Chen, Cheng Li, Kai Zhang, Liang Yang, and Yinlong Xu. 2023. Revitalizing the Forgotten {On-Chip}{DMA} to Expedite Data Movement in {NVM-based} Storage Systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 363–378.

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

[46] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 105–121. https://doi.org/10.1145/3613424.3614256

[47] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, Cheng Chen, Hui Zhang, Fei Liu, Shuai Zhang, Xiaoning Ding, and Jianjun Chen. 2024. Exploring Performance and Cost Optimization with ASIC-Based CXL Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 818–833. https://doi.org/10.1145/3627703.3650061

[48] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. 2024. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295* (2024).

[49] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

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

[51] B. Wheeler. 2022. *Graviton3 Debuts Neoverse V1.* Technical Report. Linley Group Microprocessor.

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

[53] Jing Xia, Chuanning Cheng, Xiping Zhou, Yuxing Hu, and Peter Chun. 2021. Kunpeng 920: The first 7-nm chiplet-based 64-Core ARM SoC for cloud services. *IEEE Micro* 41, 5 (2021), 67–75.

[54] Lilia Zaourar, Mohamed Benazouz, Ayoub Mouhagir, Carlos Falquez, Antoni Portero, Nam Ho, Estela Suarez, Polydoros Petrakis, Manolis Marazakis, Francesco Sgherzi, Ivan Fernandez, Romain Dolbeau, and Dirk Pleiter. 2024. Case Studies on the Impact and Challenges of Heterogeneous NUMA Architectures for HPC. In *Architecture of Computing Systems*, Dietmar Fey, Benno Stabernack, Stefan Lankes, Mathias Pacher, and Thilo Pionteck (Eds.). Springer Nature Switzerland, Cham, 251–265.