POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

CRIU-DSM: a Userspace Live Thread Migration and Transparent Distributed Shared Memory System

Tesi di Laurea Magistrale in
Computer Science And Engineering

Edoardo D'Alessio, 252034

Abstract: Due to Moore's law stalling single core performance because of power density and thermal limits, scaling up applications has become more difficult. In light of rapidly increasing network and memory bandwidth, it is time to revisit scaling out threads with support from Distributed Shared Memory systems. This thesis introduces CRIU-DSM, a userspace DSM system that scales out live POSIX threads across multiple physical machines.
The proposed approach extends Checkpoint/Restore In Userspace (CRIU) to distribute an arbitrary subgroup of threads across physical nodes and leverages recent Linux kernel APIs to maintain coherence among them. We develop a centralized page level MSI write invalidation protocol using `userfaultfd` to intercept missing and write protection faults in user space, `process_vm_readv` to read pages across processes, and `process_madvise` to invalidate them. Our solution preserves standard POSIX `pthread` semantics. CRIU-DSM can operate in a fully transparent mode for unmodified applications, or in an optional manual mode requiring minimal source level modifications to selectively register shared mmap regions and improve performance by mitigating false sharing. The system is evaluated on CloudLab using Stanford's Phoenix MapReduce suite over both TCP and RDMA transports. Results show that performance improves when write sharing is limited and the computation to page fault ratio is high. Matrix Multiply achieves up to 3.39x execution speedup and PCA up to 2.1x, whereas workloads that generate excessive invalidations, such as Kmeans due to fine grained centroid updates, perform worse than native execution because frequent low computation updates prevent amortization of DSM overheads. Overall, the results indicate that a CRIU based live thread redistribution combined with userspace fault handling provides a practical and deployable foundation for transparent DSM.

Advisor:
Marco Domenico Santambrogio
Academic year:
2025-2026

Key-words: Distributed Shared Memory, Live Thread Migration, CRIU, userfaultfd, Memory Coherence, RDMA

1. Introduction

1.1. Goals and Research Questions

The purpose of this thesis is to transparently scale multithreaded POSIX executables across multiple physical machines by designing a Distributed Shared Memory (DSM) system that preserves shared-memory semantics. This system, named CRIU-DSM, is built entirely in user space using existing Linux mechanisms, including CRIU, page fault interception via userfaultfd, and inter-process memory operations, without requiring new kernel recompilation or new programming models.
These are the questions that guided our research:

- **RQ1: Is it feasible to implement a correct page-based DSM entirely in user space using existing Linux primitives?** This evaluates whether CRIU, userfaultfd, and memory syscalls can be composed into a reliable coherence mechanism.
- **RQ2: How transparent can the system be for unmodified applications and what minimal adaptations are needed to achieve good performance?** This analyzes the trade-off between full transparency and performance-oriented hints such as selective page tracking and alignment.
- **RQ3: Under what workloads and thread-distribution strategies does the system achieve speedup, and what factors limit scalability?** This examines performance on realistic multithreaded benchmarks, focusing on page-fault behavior, synchronization, and transport choice.

1.2. Motivation

Moore's Law drove performance improvements yielding steady gains in single-thread execution. This trend stalled in the early 2010s due to power-density and thermal limits, shifting architectural progress toward multi-core designs, NUMA hierarchies, and accelerators [23]. However, these approaches incur growing costs in cache coherence, synchronization, and memory bandwidth, leading to diminishing returns for tightly coupled workloads.
In contrast, memory and networking subsystems have continued to improve rapidly, successive DDR generations have increased aggregate memory bandwidth, while modern interconnects such as RDMA provide sub-microsecond latency and hundreds of gigabits per second of throughput. When normalized in CPU cycles, network latency and bandwidth costs have decreased by orders of magnitude, while DRAM latency has remained nearly constant [24]. As a result, remote memory access is no longer prohibitively more expensive than local DRAM for many workloads. This shift challenges the assumptions that historically made Distributed Shared Memory (DSM) impractical, because early DSM systems suffered from millisecond-scale page migration and coherence overheads, rendering remote memory accesses several orders of magnitude slower than local ones [27]. Recent systems such as ArgoDSM, GAM, and DRust [16, 24, 30] demonstrate that, by leveraging RDMA and optimized coherence protocols, DSM can approach native performance for suitable applications.
Despite these advances, most modern DSM systems require kernel modifications, compiler instrumentation or specialized APIs, limiting their deployability. Recent Linux mechanisms provide an alternative path. The `userfaultfd` interface enables userspace handling of page faults [10], and CRIU [1] supports checkpoint and restore of complete process state without kernel changes.
These capabilities motivate the design of a CRIU-backed DSM that combines userspace page fault handling with thread migration to provide transparent multi-node execution for POSIX applications. The convergence of stagnant single-core performance, fast networks, and userspace memory management makes this a timely opportunity to revisit DSM as a practical foundation for distributed execution.

1.3. Thesis Structure

This thesis is organized as follows:

- **Chapter 2: Background** presents the necessary background on page faults, `userfaultfd`, Distributed Shared Memory systems, CRIU and the Linux kernel cross process memory syscalls.
- **Chapter 3: Related Work** reviews related work in DSM, thread migration, and user space fault handling systems.
- **Chapter 4: System Design** describes the architecture of CRIU-DSM, detailing thread redistribution, the page level MSI coherence protocol, and the interaction between DSM clients and the centralized server.
- **Chapter 5: Implementation** discusses the implementation, including modifications to the CRIU restore workflow, fault handling logic, page ownership tracking, and the integration of TCP and RDMA transports.
- **Chapter 6: Evaluation** presents the results of our solution on CloudLab using the Phoenix MapReduce benchmark suite, analyzes scalability and overheads, and examines the impact of write sharing and

computation to page fault ratio.
- **Chapter 7: Conclusion** summarizes novel contributions, discusses limitations, and outlines directions for future work.

2. Background

This chapter summarizes the Linux mechanisms and system primitives that enable CRIU-DSM, including `userfaultfd` for userspace interception of missing-page and write-protection faults, `process_vm_readv/writev` for efficient cross-process memory copying and `madvise(MADV_DONTNEED)` for page invalidation.
We then explain how CRIU can be leveraged for thread migration and review the DSM principles that inform the coherence model used in CRIU-DSM. The chapter concludes by comparing the two networking mechanisms employed by the system, RDMA and TCP.

2.1. Page Faults and Linux Memory Semantics

Linux virtual memory relies on page faults to both provision memory and enforce access permissions. A page fault [28] is raised when the MMU cannot complete an access using the current page-table entry, transferring control to the kernel. Based on the faulting address, access type, and VMA metadata, the kernel distinguishes three relevant cases: page not-present, protection violations, and invalid address accesses.
Beyond demand paging, protection faults play a key role in detecting and controlling write accesses. A common example is Copy-on-Write (COW) [28], where Linux temporarily revokes write permissions to shared pages, when a write fault is triggered then every owner gets its own copy. This mechanism illustrates how page-level permission changes can be used to intercept writes and trigger software-defined actions. However, in multithreaded processes, threads normally share the same `mm_struct` and retain write access to shared pages, requiring explicit reintroduction of write protection to observe modifications.

Relevant Memory Classes

Linux memory is divided into anonymous and file-backed regions. Anonymous pages include heap allocations, global variables, thread stacks and usually contain shared data structure. File-backed mappings, such as binaries and shared libraries, are fully managed by the kernel and can be safely excluded from coherence management.

2.2. Userfaultfd Mechanism

Under normal execution, page faults are resolved entirely inside the kernel. The `userfaultfd` mechanism [10] allows selected faults to be delegated to userspace by registering virtual-memory regions and fault types of interest. When a fault occurs on a registered region, the kernel suspends the faulting thread and delivers a notification to a userspace handler through a file descriptor.
Each notification is delivered as a `uffd_msg` structure, which describes the event that triggered the fault. For page-fault events, the message reports the faulting virtual address and a set of flags identifying the cause. Two flags are particularly relevant: `UFFD_PAGEFAULT_FLAG_MISSING`, raised when no physical page is mapped, and `UFFD_PAGEFAULT_FLAG_WRITE`, raised on write attempts to protected pages. If requested, the message can also include the identifier of the thread that triggered the fault, allowing the handler to associate the event with the correct execution context.
Together, this information enables a userspace handler to precisely identify the location and type of a fault and to apply the appropriate corrective action before resuming execution.

```
struct uffd_msg {
    __u8  event;            /* Type of event */
    ...
    union {
        struct {
            __u64 flags;    /* Flags describing fault */
            __u64 address;  /* Faulting address */
            union {
                __u32 ptid; /* Thread ID of the fault */
            } feat;
        } pagefault;
        ...
```

```
    } arg;
};
```

2.3. Memory Syscalls

Linux provides several system calls that support efficient inter-process data movement and memory-management operations. In particular, `process_vm_readv`, `process_vm_writev`, and `madvise` enable kernel-mediated copying of memory contents and selective invalidation of physical pages.

Cross-process Memory Copy

The `process_vm_readv` and `process_vm_writev` system calls [9] allow direct copying of data between the virtual address spaces of two processes, provided the caller has sufficient privileges to access the target VMAs. The kernel resolves and pins the involved pages and performs the copy entirely in kernel space. Since no userspace code is executed in the target process, context switches and intermediate buffering are avoided, making these syscalls well suited for repeated bulk data transfer between processes.

Memory Invalidation

The `madvise` system call [6] allows a process to provide usage hints or requests to the kernel regarding the handling of its virtual memory pages. The `MADV_DONTNEED` behaviour cause the physical page to be discarded, deleting its page table entry. For anonymous pages, subsequent accesses trigger not-present faults and recreate pages as zero-filled.
The related `process_madvise` [8] interface extends this functionality to operate on the address space of another process. This enables external control over page invalidation and reclamation without executing code in the target process. However, only a subset of `madvise` behaviors is currently supported, which constrains the set of memory-management operations that can be applied remotely.

2.4. CRIU: Checkpoint and Restore In Userspace

CRIU (Checkpoint and Restore In Userspace) [1] is an open source framework that can store the entire process state of a running program, a checkpoint, and later restore it to the exact execution point.
During the dump phase, CRIU serializes the virtual address space, memory contents, process and thread hierarchy, CPU registers, open file descriptors, network sockets and other kernel-managed resources.
The resulting images include process tree metadata and per-thread core images containing register state, instruction pointers, signal masks, and timers. Together, these artifacts allow precise reconstruction of complex multithreaded applications. Image files can be inspected and modified offline using the CRIU Image Tool (CRIT), enabling controlled changes such as memory edits, remapping, or thread selection prior to restore.
In the restore phase, CRIU recreates the process from the images, reestablishes kernel resources, and resumes execution transparently. CRIU is widely used for container live migration, fault tolerance, and debugging, and it supports cross-machine migration through on-demand page fetching [3]. However, its default version restores a process as a whole and does not allow distributing different subsets of threads across nodes. This limitation motivates the extensions proposed in this thesis, which integrate CRIU into a DSM system capable of splitting threads across machines while preserving coherence and consistency of the shared address space.

2.5. Distributed Shared Memory

Distributed Shared Memory (DSM) [27] provides the abstraction of a single shared address space over a set of physically distributed machines. At page granularity, transparency is achieved through page replication for reads and page migration for writes, enforcing the Single-Writer / Multiple-Reader (SWMR) rule and preserving a consistent data value across replicas.
DSM systems can be realized in hardware or software. Hardware solutions offer low latency but require specialized platforms, whereas software DSM favors portability and flexible consistency models. In both cases, maintaining coherence and consistency across nodes is a core requirement.

Coherence with the MSI Protocol

Page-level coherence is commonly enforced using the MSI protocol [23]. Each page resides in exactly one of three states:

- **Modified (M)**: the page is owned by a single node with exclusive read/write access; all other copies are invalid.
- **Shared (S)**: the page is replicated across nodes in read-only mode with identical contents; any write requires invalidating all replicas.
- **Invalid (I)**: the node holds no valid copy; any access triggers a page fault and a fetch from the current owner.

Local vs. Remote Transitions

As shown in Figure 1, MSI state changes are driven by two classes of events:

- **Local events**: local reads and writes that cause page faults or permission upgrades on the accessing node.
- **Remote events**: coherence messages generated by other nodes, such as read requests or invalidations, which force local downgrades or invalidation.

Together, these transitions fully describe the evolution of page states in a distributed shared memory system.

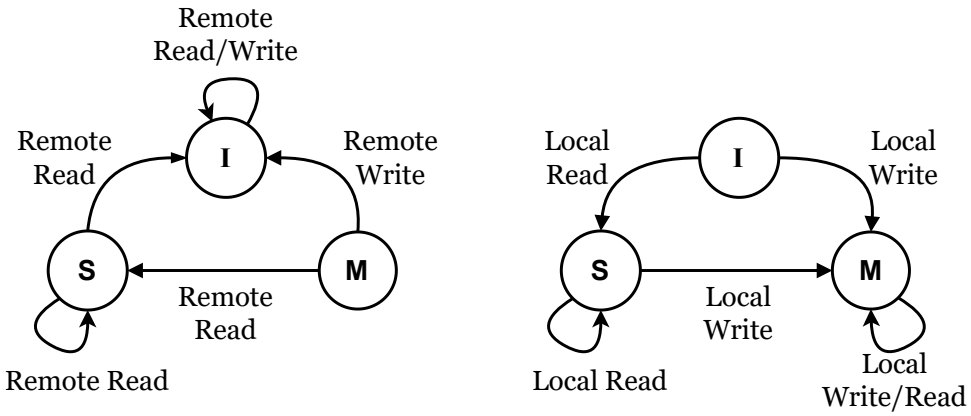

Figure 1: MSI state transitions for local and remote read/write operations.

2.6. Network Connections

Efficient communication is critical in distributed systems, where performance is often dominated by the cost of moving data between nodes. Modern networking mechanisms aim to reduce CPU involvement and software overhead, progressively shifting data movement closer to the hardware.

DMA with TCP/IP Networking

Conventional TCP/IP networking relies on kernel mediation and multiple memory copies, even though network devices internally use DMA. As shown in Figure 2, data generated by a user process is first copied into kernel space, processed by the transport stack, and then transferred by the NIC to its buffers before being sent over the network. On reception, the path is reversed, introducing additional copies and context switches. These transitions add latency and consume CPU cycles, limiting throughput and scalability. Techniques such as zero-copy I/O and TCP offload engines reduce but do not eliminate this overhead.

Remote Direct Memory Access (RDMA)

Remote Direct Memory Access (RDMA) [33] generalizes direct memory transfers across machines. RDMA-capable NICs move data directly between pre-registered memory regions on different hosts, bypassing the TCP/IP stack, kernel buffers, interrupts, and most CPU involvement. This design provides significantly lower latency, higher throughput, and reduced CPU load, making it well suited for workloads dominated by frequent remote memory operations (bottom Figures 2).

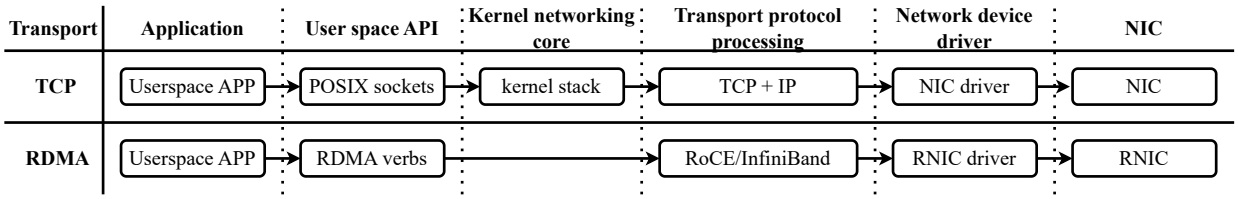

Figure 2: TCP vs RDMA data path

By collapsing multiple software layers into hardware-driven transfers, RDMA shifts inter-node communication from CPU-bound to memory-bound. This directly addresses the dominant cost in distributed shared memory systems, where remote page transfers form the primary performance bottleneck.

3. Related Work

This chapter reviews prior research on distributed shared memory, memory coherence, and framework for process and thread migration. Existing systems address individual aspects of these problems, but none jointly provide live migration of arbitrary thread subsets with a page fault driven userspace DSM, which are the core goals of CRIU-DSM.
It diverges from previous approaches along three main points: (i) it allows selective live migration of any subset of threads, (ii) it supports both fully transparent execution and optional customizable functions and (iii) it enables DSM only when threads are distributed, allowing purely local execution without recompilation.

3.1. Distributed Shared Memory

Early DSM systems such as Ivy [27], Munin [14], and TreadMarks [17] pioneered page-based coherence protocols implemented entirely in software. Although relaxed consistency models improved scalability, these systems incurred high page-fault overheads and were constrained by the limited bandwidth and latency of contemporary networks.
Subsequent DSM designs explored different trade-offs along the axes of programming model exposure, system integration, and network substrate:

- **Language- or API-integrated DSM:** Frameworks such as Grappa [32] and PGAS-based systems [22, 35, 36] expose global memory abstractions through explicit APIs or language extensions. This approach improves control and performance but requires application refactoring and adoption of a specific programming model. Similarly, DRust [30] integrates coherence management at the object level by leveraging Rust's ownership semantics, requiring applications to be rewritten in Rust.
- **Transparent DSM with system-level support:** DEX [25] implements DSM inside the operating system kernel and leverages RDMA for efficient cross-node memory access. It provides transparency to applications but requires kernel modifications and assumes homogeneous architectures. COMET [20] instead implements DSM at the virtual machine level on top of the *Dalvik* VM, enabling transparent thread migration without modifying applications. However, it is limited to managed runtimes and specific virtual machine environments.
- **RDMA-accelerated DSM designs:** Systems such as ArgoDSM [24], Menps [19], Remote Regions [11], and Concordia [34] exploit high-bandwidth RDMA-capable networks to reduce DSM overhead. These systems primarily focus on optimizing coherence and data movement using one-sided communication primitives. In contrast to CRIU-DSM, they typically assume static thread placement, where thread-to-node mappings are fixed at startup and not dynamically reconfigured at runtime.

In contrast, CRIU-DSM targets existing multithreaded C/POSIX programs, operates entirely in userspace without kernel modifications, and activates distributed coherence only when execution spans multiple nodes, thereby avoiding overhead during purely local execution.

3.2. Thread and Process Migration

Traditional migration systems, including Sprite [18], MOSIX [13], and OpenMosix [7], support whole-process migration but neither allow partial thread relocation nor integrate DSM. Similarly, CRIU-based virtualization mechanisms operate at process granularity and do not support distributed execution. More recent work has explored migration in heterogeneous environments.
CRIU-RTX [31] combines thread migration with a userspace DSM for cross-ISA execution, but depends on Pop-

corn Linux and compiler-assisted state transformation, it leverages HetMigrate [12] to enable cross-architecture migration at compiler-inserted equivalence points, but operates at process granularity and does not support pthread semantics.
CRIU-DSM instead focuses on homogeneous systems and avoids compiler instrumentation and kernel modifications. It uniquely supports migration of thread subsets during execution, allowing non-migrated threads to continue running locally.

3.3. Userspace Coherence and Remote Memory

Several systems provide fast remote memory access through RDMA or kernel bypass mechanisms, including Clio [21] and Lite-Kernel RDMA [33]. These solutions typically rely on specialized libraries or kernel components rather than transparent memory interception.
Other projects based on `userfaultfd`, such as Chameleon [29] and Rave [15], showcase advanced userspace control over memory faults and execution state, but do not address distributed shared memory or multithreaded migration.
CRIU-DSM provides coherent shared memory across distributed threads without kernel or compiler changes.

3.4. Design Space Positioning

CRIU-DSM occupies a distinct position among existing systems:

- It is transparent by default, supporting many applications through dynamic instrumentation alone.
- It allows optional, low-effort programmable controls to improve performance without enforcing a new programming model.
- It enables fine-grained thread migration entirely at user level.
- It combines CRIU-based live migration with a userspace DSM, a combination not previously explored.

Overall, CRIU-DSM differs from DSM systems that mandate new APIs, from migration frameworks limited to whole processes, and from heterogeneous solutions requiring compiler or kernel support. It provides a deployable approach for scaling unmodified POSIX multithreaded applications across nodes while preserving a coherent shared memory abstraction.

4. System Design

The goal of CRIU-DSM is to design a transparent framework able to scale out threads across multiple physical nodes, turning a multithreaded application into a distributed one. It leverages several Linux tools and APIs, such as `CRIU` for process snapshotting and thread migration, `userfaultfd` for handling page faults directly in user space, and the `process_vm_readv` and `process_madvise` system calls for efficient remote memory access and invalidation. Together, these tools enable a Distributed Shared Memory (DSM) abstraction, in which threads running on different nodes access a shared address space transparently, as if they were executing on a single machine. In this chapter, we present the design choices behind CRIU-DSM, the main DSM components, the high-level workflow of the overall system.

4.1. Architecture Overview

CRIU-DSM is organized around two main runtime roles, a DSM Server and one or more DSM Clients. These roles are selected during process restoration through two extensions to the standard CRIU restore command, `-dsm_server` and `-dsm_client`, which integrate the DSM runtime directly into the restore phase and determine the behavior of each node.
As shown in Figure 3, each restored process consists of ordinary application threads and a small set of DSM-specific helper threads. Application threads resume execution normally and remain unaware of the distributed nature of the address space. The DSM application is composed of two possible threads: local and remote fault handler. The first polls on the `userfaultfd` file descriptor with which we registered the pages for local page faults, and then resolves them by communicating with the DSM Server and enforce coherence. The latter waits for communication from another node for remote faults and it will execute coherence on local threads.
Nodes communicate using either TCP or RDMA. The control plane carries metadata and coordination messages, including page requests, invalidations, and synchronization events, while the data plane is used for bulk page transfers. By mapping shared-memory accesses and synchronization primitives onto controlled page faults, the runtime preserves the illusion of a single shared address space without requiring application-level modifications.

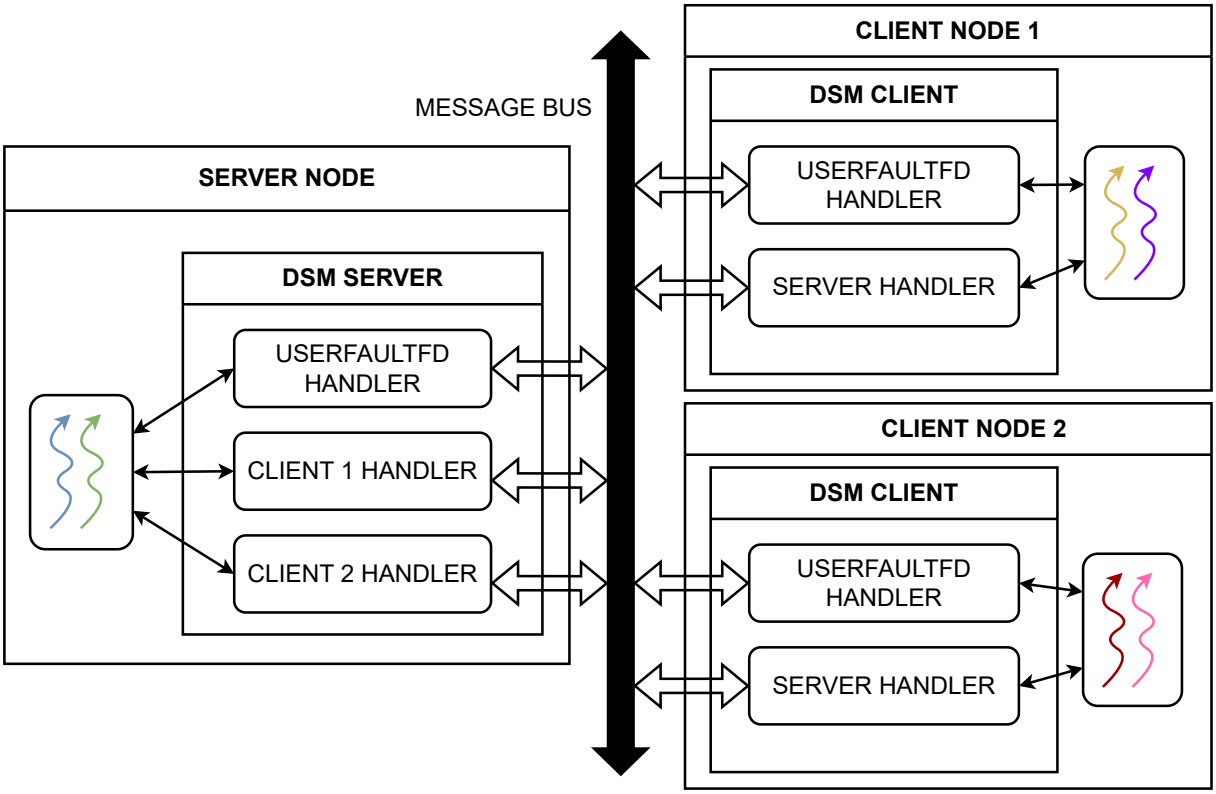

Figure 3: General overview of CRIU-DSM components

4.2. DSM Server

The DSM Server acts as the global coordinator for memory coherence and distributed synchronization. It maintains authoritative metadata for each managed page, including its coherence state and the set of nodes currently holding a copy, enforces single-writer semantics for write operations and it's the arbiter in synchronization requests.
Internally, the server maintains a page directory mapping all tracked addresses to ownership and state information, mutex owner and requests, and barrier synchronization progress. Active communication for local faults with clients is handled by one dedicated thread, passive communication instead is one per client node, which processes incoming DSM messages and triggers the corresponding memory operations.
This organization cleanly separates local fault handling from remote coherence management and limits the number of runtime threads. When RDMA is enabled, the server additionally manages pre-registered memory regions to allow direct remote writes, avoiding intermediate copies during page transfers.

4.3. DSM Client

Each node running application threads includes a DSM Client responsible for intercepting memory accesses and interacting with the DSM Server. The client registers all distributed memory pages with `userfaultfd` and spawns a local fault handler thread that blocks on the corresponding file descriptor.
When a missing or write-protected page is accessed, the handler constructs a DSM request and forwards it to the server. A separate communication thread waits server requests and applies page updates or invalidations locally.
From the application's perspective, execution remains unchanged. Shared memory regions are allocated using standard mechanisms, synchronization primitives follow normal POSIX semantics, and all distributed behavior is mediated by the DSM runtime. By encoding both memory accesses and synchronization events as page faults, CRIU-DSM maintains transparency while supporting distributed execution across nodes.

4.4. Thread Redistribution Workflow

CRIU-DSM redistributes threads by modifying the restoration images of the CRIU checkpoint. The high-level workflow is sketched in Figure 4:

1. The multithreaded application runs until a safe point in the computation is reached, e.g. all threads are created.
2. Then the checkpoint files are sent to the remote nodes, where they are modified so that only a chosen subset of threads is restored on each node.
3. Lastly, depending on the selected role, a node starts either as DSM Server or DSM Client, and shared memory regions are immediately registered with `userfaultfd`. From this point onward, distributed execution proceeds transparently: memory accesses that cannot be resolved locally trigger page faults that are redirected to the DSM layer.

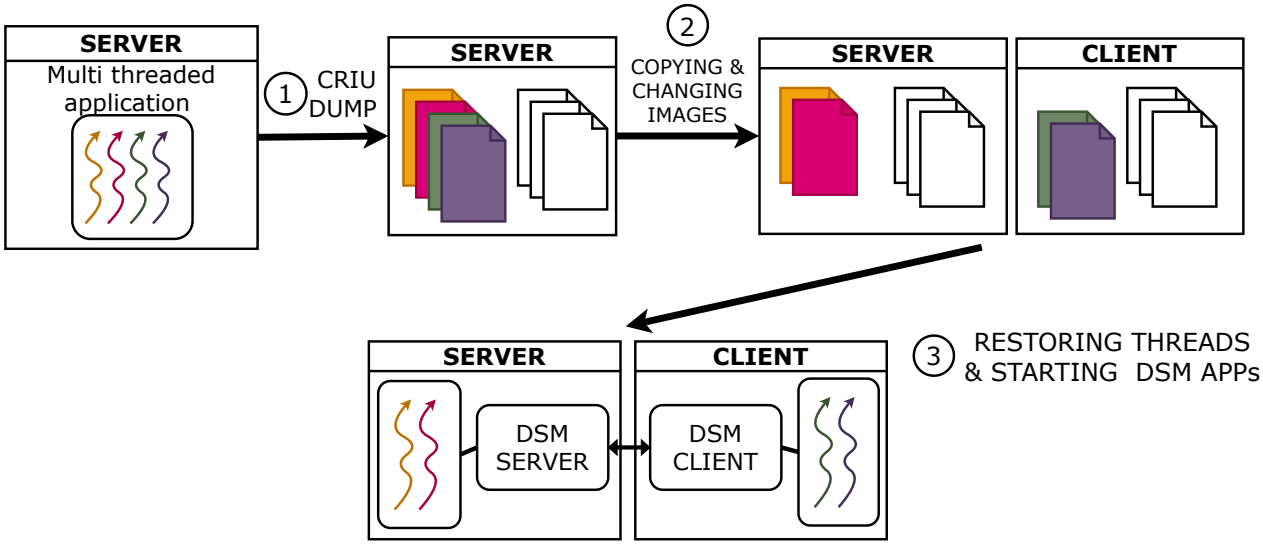

Figure 4: CRIU-DSM workflow for scaling out threads to remote nodes

4.5. Memory Coherence Model

Since CRIU-DSM leverages `userfaultfd` to enforce memory coherence, it works at page granularity, i.e. 4 KB virtual pages. This choice simplifies the design and minimizes metadata overhead, at the cost of potential false sharing and coarse invalidations, that can be handled in different ways. The MSI coherence protocol ensures that all threads observe a consistent memory view equivalent to a shared address space.

Replication and Protocol Selection

To avoid centralized bottlenecks, CRIU-DSM allows multiple nodes to hold read-only replicas of the same page, without it all accesses would be serialized through a single owner, severely limiting scalability for read-dominated workloads. To keep copies consistent, we use write-invalidation protocol coordinated by the centralized DSM Server. When a node intends to modify a page, all other replicas are invalidated before granting write access, other nodes retrieve the page again only if they later access it.
This design reduces unnecessary data transfers compared to write-update approaches, which would propagate full page contents on every write. Given the cost of transferring entire pages over software-managed networks, write invalidation better matches the performance characteristics of a userspace DSM, since whenever a page is `SHARED` or `MODIFIED`, we can use it without contacting the server.

Ownership and Page States

CRIU-DSM implements the MSI protocol with three states: `INVALID`, `SHARED`, and `MODIFIED`. The DSM Server maintains centralized table describing the state of each page and the set of nodes currently holding valid copies, this centralized ownership simplifies conflict resolution and ensures deterministic state transitions.
After restoration, pages are initially marked as `SHARED`, enabling read-only access without coordination. Since write accesses require exclusive ownership, they will trigger invalidation of other replicas through Server and

promotion to the `MODIFIED` state. Read faults on missing pages result in a fetch from the server and an update of the sharer set. The precise fault and message handling logic is detailed in the Implementation chapter 5.

Limitations

Page-based coherence introduces inherent limitations such as false sharing that may arise when unrelated data resides on the same page and even small updates may cause full-page transfers or invalidations. Write accesses are more expensive than reads due to server involvement in invalidating all sharers. We address these challenges through allocation strategies and workload structure: shared regions are page-aligned using `mmap`, instead of `malloc`, and unrelated objects are allocated in different pages. Using large problem sizes increases the amount of computation performed per page, thereby reducing the relative overhead of page movement. Execution is organized such that threads update non-overlapping memory regions, which limits write sharing. Since many target workloads are predominantly read intensive, pages can stay in the `SHARED` state for long intervals with little or no synchronization. In addition, shared regions are divided across threads rather than interleaved, allowing local accesses to be faster.

4.6. Synchronization Design

CRIU-DSM implements synchronization by mapping coordination events to controlled page faults, avoiding changes to application-level synchronization APIs. Special synchronization pages are registered with `userfaultfd`, and accesses to these pages are interpreted by the runtime as synchronization operations.
Barriers follow a hierarchical scheme. Threads first synchronize locally on each node, then, a representative thread triggers a fault on a shared barrier page, notifying the DSM Server. Once all nodes have reached the barrier, the server releases them simultaneously and then the representative thread can join the others for the second local barrier, as shown in Figure 5.
This is to be considered one complete synchronization cycle, because we made sure that all local thread first finish work, first local barrier, then the representative notifies the Server, and once all nodes have arrived, it releases them all. Now it is possible to safely perform page transfers between nodes to prepare for the next computation stage. Then after the page exchange we need to have another synchronization cycle, to make sure that every node finished the page requests.
We currently support `pthread_join`, mutex locks, and `pthread_barrier`, enabling existing barrier-style synchronization to be implemented through fault-driven primitives without modifying application APIs.

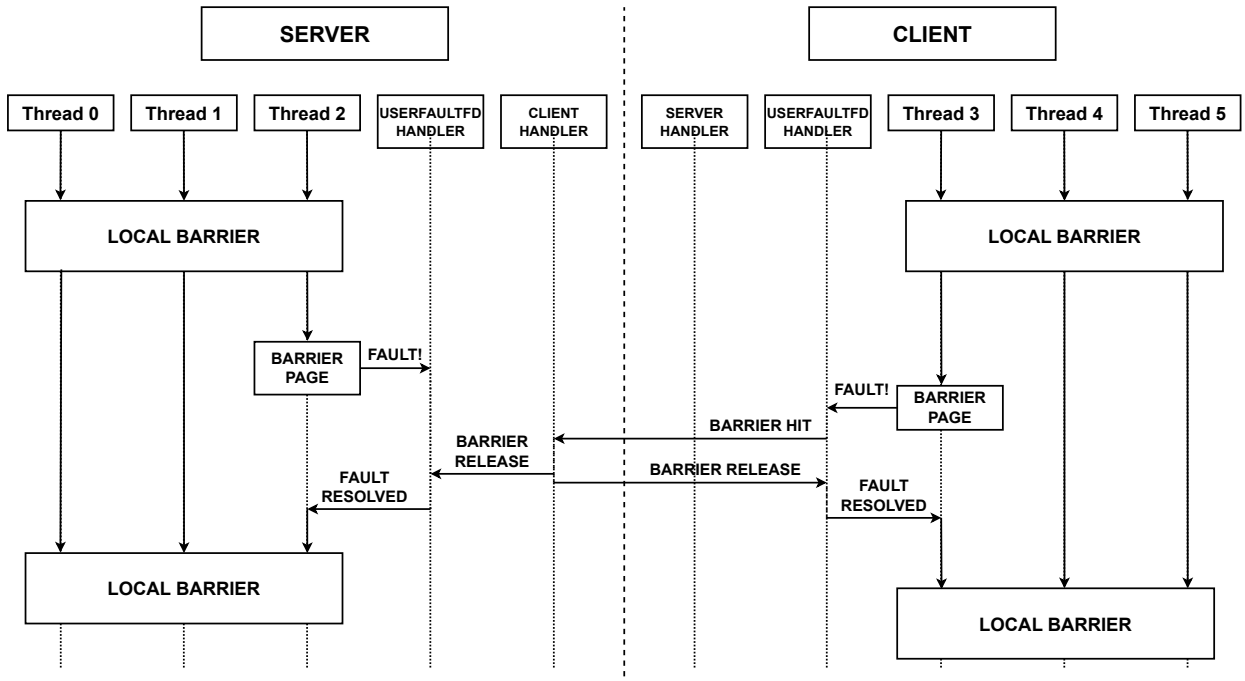

Figure 5: Barrier protocol overview

4.7. Communication and Networking Design

Communication in CRIU-DSM is divided into a control plane, the handling of metadata exchanges such as fault notifications, invalidations, and synchronization events, and a data plane, the transferring of page contents using either TCP or RDMA.
The system follows a centralized hub model in which all DSM Clients communicate exclusively with the DSM Server depicted in Figure 6. Each client userfaultfd thread maintains a single connection to the server (blue arrow), which instead has a dedicated communication thread associated with each client, i.e. "Client X handler". Every thread on the Server may need to communicate with any client, because for example Client 1 could need a page from Client 2. To avoid a full mesh of connections while still allowing any server thread to contact any client, each client connection is protected by a per-client lock. Server threads acquire this lock before sending messages, ensuring correct serialization, for example in Figure 6, Server userfaultfd thread got the lock for communicating with Client 1 Server thread. This design scales linearly with the number of clients and is shared by both the TCP and RDMA backends, keeping higher-level DSM logic independent of the transport mechanism. In Figure 6 the arrows indicate only which component initiates the communication, but the communication is bidirectional.

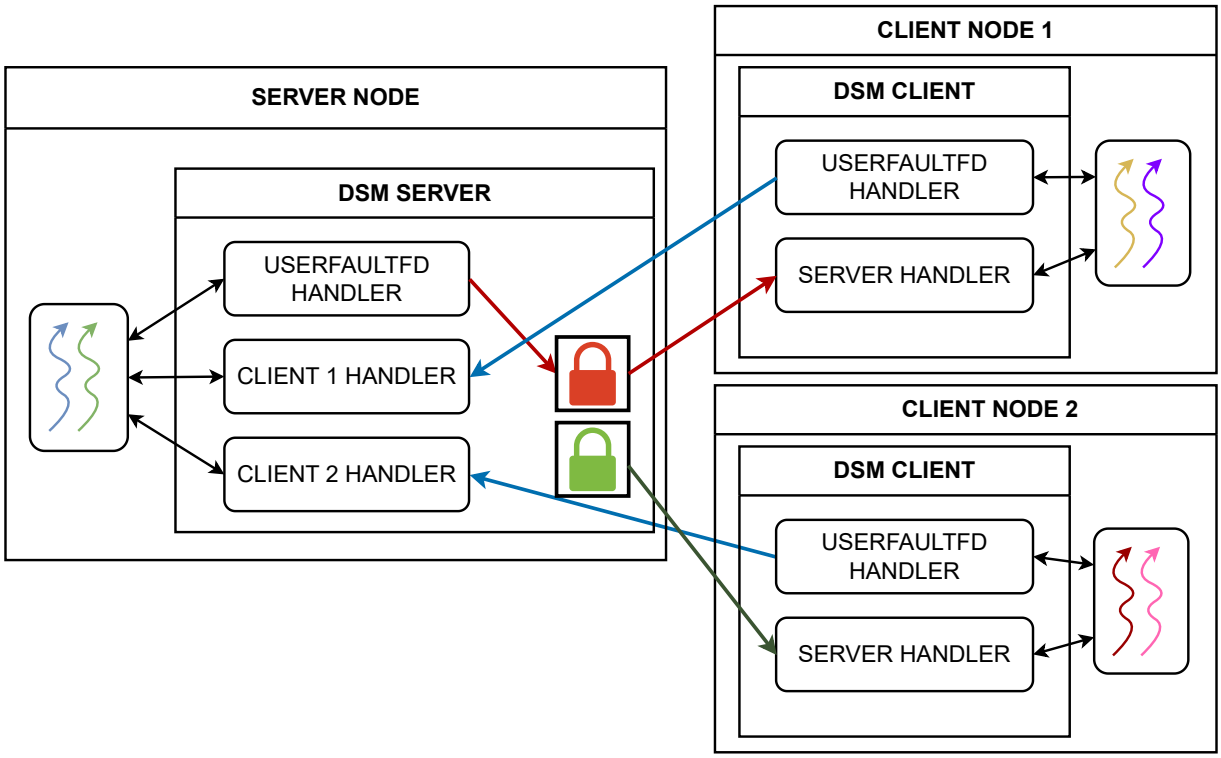

Figure 6: Communication network between Server and Clients

5. Implementation

This chapter explains how CRIU-DSM concretely implements the design introduced in Chapter 4. It describes the mechanism used to redistribute and choose threads through CRIU, the minimal adaptations required from applications and the userspace realization of memory coherence, synchronization, and communication.

5.1. CRIU-based Thread Redistribution

CRIU-DSM relies on CRIU checkpoint–restore to relocate execution contexts across nodes. Thread redistribution is achieved through a two-step pipeline: checkpoint creation and images transformation.

Checkpoint Creation

Applications are executed normally until all relevant threads have been initialized and at a safe code point. Now we can CRIU dump to capture the full process state, including memory mappings, registers, file descriptors, and thread contexts. Performing the dump after thread creation guarantees that all execution contexts are available for later redistribution.
The resulting checkpoint directory is transferred to each node that will participate in distributed execution with a simple `scp` command. Each node then edits the checkpoint images to restore only the threads assigned to it.

Thread Selection

CRIU does not directly support partial restoration of multithreaded processes. CRIU-DSM addresses this limitation by post-processing the checkpoint images using a Python script built on top of CRIT.
The transformation targets two images. First, the `pstree.img`, i.e., the file that contains the hierarchical structure of the process with all its threads, is edited to retain only the selected thread identifiers for the local node, designating one of them as the main thread. All other threads are removed from the local image. For example, in the case below, the main thread is 27545. Assuming two nodes, the server will maintain the old main plus 27546. The client will change the main to 27547 and delete the first two.

```
{
    "magic": "PSTREE",
    "entries": [
        {
            "pid": 27545,
            "ppid": 0,
            "pgid": 11420,
            "sid": 4857,
            "threads": [
                27545,
                27546,
                27547,
                27548
            ]
        }
    ]
}
```

Second, the corresponding new thread leader `core-<PID>.img` file is modified to include the `tc` field from the original process leader. These changes allow each node to restore a coherent subset of threads while preserving the global identity of the process and without changes to the CRIU restoration flow or code.

5.2. Application Adaptation

CRIU-DSM is designed to operate transparently, but small application adjustments can significantly improve performance. In particular, the DSM runtime must be able to identify which memory regions participate in distributed sharing and intercept faults on them.

Page Selection Options

The system supports both implicit and explicit identification of shared pages. In the implicit approach, CRIU-DSM scans `/proc/<pid>/maps` after restoration and registers all anonymous, non file-backed regions with `userfaultfd`. This strategy maximizes transparency but may introduce overhead by tracking memory that is not actively shared. It can also introduce false sharing and contention, increasing DSM traffic and reducing performance.
Alternatively, developers may explicitly allocate shared data using `mmap` and record the corresponding address ranges. Only these regions are registered with `userfaultfd`, reducing fault traffic and improving scalability on large address spaces.

Memory Allocation Strategy

Memory allocated through `malloc` is typically managed via `brk` and cannot be selectively tracked using `userfaultfd`. For this reason, CRIU-DSM relies on anonymous `mmap` allocations for shared data. Applica-

tions can either be modified to use `mmap` directly or rely on an `LD_PRELOAD` library that transparently redirects allocation calls. In both cases, shared regions become page-aligned and fully manageable by the DSM runtime, reducing contention and avoiding false sharing.

Intel Pin developed tool

For this purpose we developed an Intel Pin based tool, that performs fine grained code analysis by capturing per address, per thread read and write events, allowing precise identification of which addresses and therefore pages are accessed by each thread and whether the access is read or write. Then by analyzing /proc/<pid>/maps and readelf symbol resolution we map runtime addresses to global variables and shared data structures, clearly identifying which program objects are accessed by each thread. A JSON exports summarize total reads and writes, thread counts, page usage, and hotspots, enabling detection of read write and write write race conditions and detailed access patterns. This information allows developers to pre evaluate whether a CRIU DSM distribution would be feasible and efficient and, if source code modifications are possible, to restructure and redistribute shared data to eliminate false sharing and improve performance.

5.3. Userspace Coherence Protocol

The MSI memory coherence is implemented entirely in userspace thanks to userfault and syscalls. The DSM Server coordinates ownership decisions, while clients act only as fault reporters and data movers.

Page Metadata

The DSM Server maintains a global page table that maps each tracked page with a coherence state and an ownership mask identifying nodes holding valid replicas. Ownership is encoded as a bitmask, with one bit per node, starting from server. Clients do not participate in coherence decisions and maintain local metadata only for debugging and consistency checks. Centralizing ownership avoids distributed lookups and simplifies the handling of concurrent faults, but with the risk of a potential bottleneck.
As shown in Table 1, the real value of the mask is stored in uint64_t, "Owner (TRUE VALUE)", but we can see binary form "Owner (MASKBIT)". A value of one in the mask indicates that the node holds a copy of the page, starting from the server. For example, in Figure 7 the binary mask `110001` indicates that nodes server, client 4, and client 5 each possess a copy of the page. Since the server is the centralized hub which all message and data go through, whenever a page is requested in `SHARED` state, the server also stores the updated version before sending to the requesting client. This optimization allows for subsequent request on the same page to be handled directly from the server and not be requested again to the writer client.

Table 1: Example of Server ownership table entries

Page Address	Status	Owner (REAL VALUE)	Owner (BITMASK)
0x7f200a7f7000	SHARED	49	110001
0x7f2aa67a3000	INVALID	16	010000
0x7f2ef7faa000	MODIFIED	1	000001

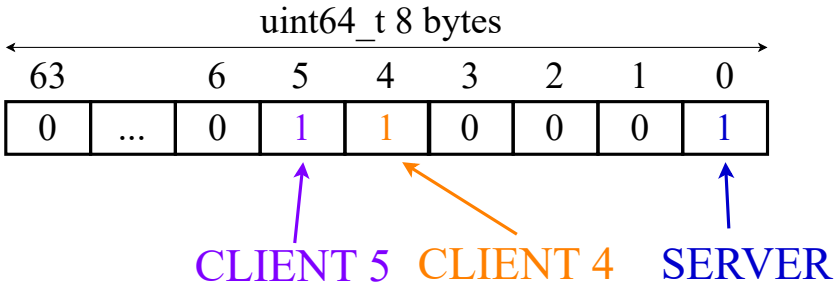

Figure 7: Page table entry ownership mask example

Fault-driven Page Requests

All nodes have a local fault handler that have the file descriptor for polling into the page faults in userspace.There are three page types of requests:

1. write faults on shared pages requiring invalidation of other replicas and ownership granting.
2. read faults on invalid pages
3. write faults on invalid pages requiring exclusive ownership

The server interprets these requests using its page table, performs the necessary state transitions, and replies with acknowledgments or page data.

5.3.1 Remote Invalidation Support

Write invalidation relies on allowing client and server DSM code to call `process_madvise(MADV_DONTNEED)` [8] to force its threads to drop stale mappings. This is a Linux syscall that allows a calling process to execute `madvise` [6] on another process target. For security reasons, the allowed options are simply a suggestion for the kernel how to handle the physical page of the target process, as shown from madvise.c [5] in Algorithm 1. If the check is not passed the syscall will fail. There are three ways to overcome this:

1. Using CRIU Compel [2] infection mechanism to infect the target process with a call for `madvise`, then call it from the DSM applications. Since the call itself is from the context of the process, it can use any `madvise` behaviour, including `MADV_DONTNEED`.
2. Simply add in the kernel code, in `madvise.c` Algorithm 1, the `MADV_DONTNEED` as a valid behavior, and recompile it.
3. Using `kprobes`, our chosen strategy.

Kprobes [4] allow to install without recompiling a kernel module, to dynamically intercept specific kernel instructions and execute custom handlers on entry or return. The mechanism works like this: it temporarily rewrites the `process_madvise` argument at syscall entry, before the check, so that kernel validation succeeds, in our case by changing the `MADV_DONTNEED` into `MADV_WILLNEED` that is a valid value, after the check returns it restores the original value immediately before the memory operation is applied, and cleans up any per-thread state on return. This rewrite–restore pattern preserves kernel security checks while enabling remote invalidation semantics required by the coherence protocol.

By confining all modifications to transient argument rewriting and restoring kernel state before any observable side effects, this approach enables remote page invalidation without requiring a custom kernel build. Also it performs better than the Compel infection mechanism because the latter does more context switching between kernel and userspace for each call.

Algorithm 1 Pseudocode of behavior validation for `process_madvise`, from madvise.c

```
if behavior is MADV_COLD or MADV_PAGEOUT or MADV_WILLNEED or MADV_COLLAPSE then
    return valid
else
    return invalid
end if
```

Algorithm 2 Pseudocode of `process_madvise`, from madvise.c

```
Receive arguments pidfd, iov, behavior, flags
if flags ≠ 0 then
    return -EINVAL
end if
if not process_madvise_behavior_valid(behavior) then
    return -EINVAL
end if
Obtain target memory descriptor mm
for each memory range in iov do
    Call do_madvise(mm, addr, len, behavior)
end for
return success
```

Write Fault Handling on SHARED Pages

After restoration, pages are initially in `SHARED` state and write protected. A `SEND INVALIDATE` request occurs when a client attempts to write to such a page and incurs a write protection fault. The handling sequence is illustrated in Figure 8.

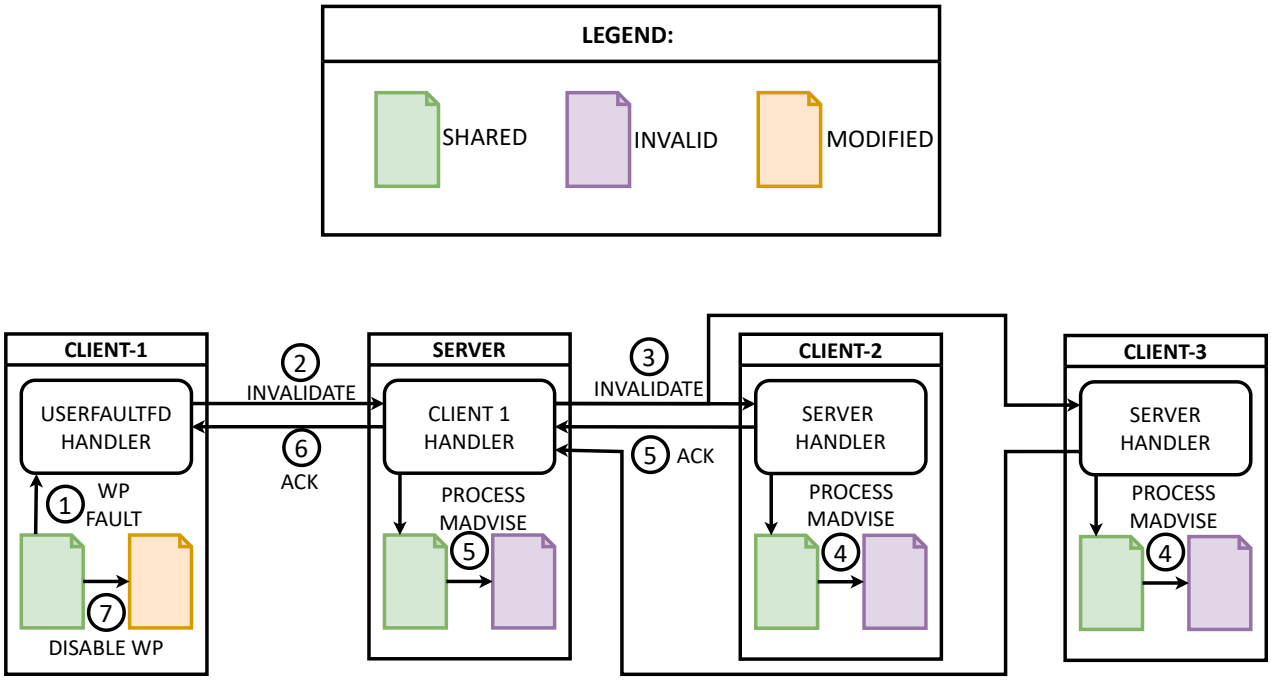

Figure 8: Example of `INVALIDATE` transaction workflow

1. It all starts with a write attempt from a thread on client 1, that triggers a page fault.
2. The fault handler of client-1 sends the invalidation request to the Server node to request all shares to be invalidated and obtain exclusive ownership.
3. The server sends a invalidation order to the other owners, in this case client 2 and 3.
4. Their server handler thread execute `process_madvise(MADV_DONTNEED)` on that page and transitions to `INVALID`.
5. Then they send acknowledgments to the server, that waits until all have arrived, then it invalidates its copy of the page.
6. Now it can update the page ownership table to `MODIFIED` and communicate the acknowledgment of the request to client 1, granting exclusive ownership.
7. Finally, the userfaultfd handler in client 1, can remove the write protection, resolving the fault, and now the thread will try the write operation again.

At the end of the sequence, client 1 is the sole owner of the page in `MODIFIED` state.

Read Fault Handling on MISSING PAGE

Continuing from the ending of Figure 8, we get missing page fault if a clients tries to read the `INVALID` page. Then it would need to request the page from the current owner though the server with a `GET PAGE` request. The handling sequence is shown in Figure 9.

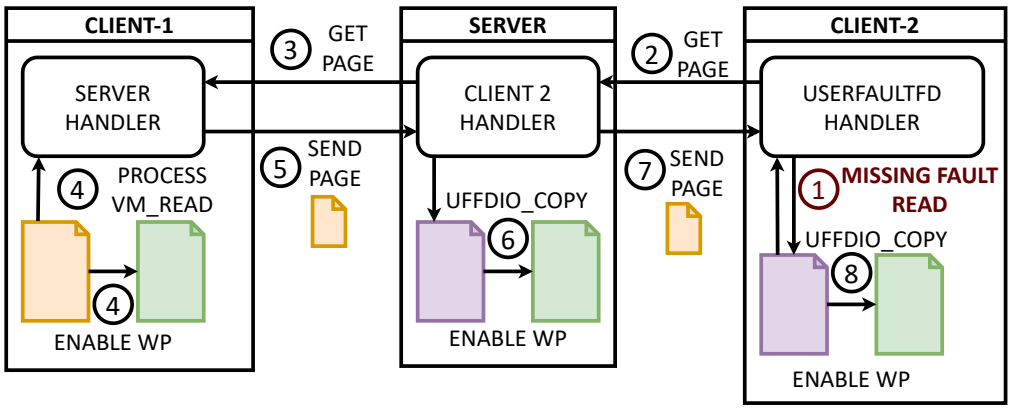

Figure 9: Example of `GET PAGE` transaction workflow

1. Now on client 2, there's a read attempt to an `INVALID` page, that generates a missing fault for read (in red).
2. The userfaultfd handler sends the `GET PAGE` request to the server.
3. The server looks up the page table and, since it doesn't have the page, it sends the `GET PAGE` request to the owner, client 1 in this case.
4. The client 1 server handler processes the request from the server by copying the content of the page from the threads VMAs through `process_vm_readv` and re-enables the write protection on its page, returning it in the `SHARED` state.
5. The client 1 forwards the page to the server.
6. In order to possible handler faster a possible new `GET PAGE` request from another client, for example client 3 (that is not in the Figure since it doesn't participate with this transaction, but it has still the page in `INVALID`), the server restores its local copy of the page, with `UFFDIO_COPY` since the page it's missing. Then it updates the page table return it to `SHARED` and adding client 2 and itself as sharers.
7. Now the server sends the page to the requesting client.
8. Finally, the userfaultfd handler can resolve the page fault by `UFFDIO_COPY` and re-enables the write protection, transitioning it to `SHARED` state.

After this transaction, the sharers are: server, client 1, client 2. Meanwhile client 3 page is still invalid, but now the request can be satisfied directly from the server, so the sequence would be: 1->2->6->7->8.

Write Fault Handling on MISSING Pages

Lastly when a node attempts to write to a `INVALID` page, we have a `GET PAGE & INVALIDATE` request. The node needs the page and exclusive ownership grated by the server. The sequence is shown in Figure 10, again continuing from the ending of Figure 8.

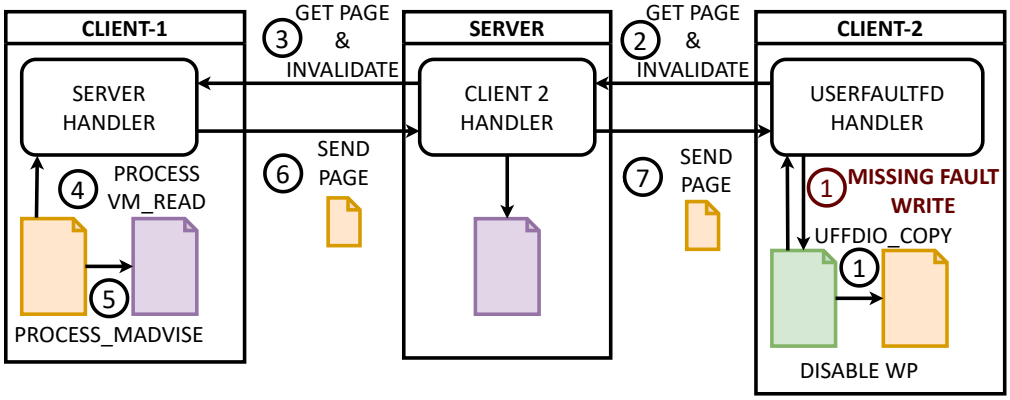

Figure 10: Example of `GET PAGE & INVALIDATE` transaction workflow

1. This time there's a write attempt in client 2 on a `INVALID` page, that generates a missing fault for write (in red).
2. The userfaultfd handler sends the `GET PAGE & INVALIDATE` request to the server.

3. The server looks up the page table and, since it doesn't have the page, it sends the `GET PAGE & INVALIDATE` request to the owner, client 1 in this case.
4. The client 1 server handler processes the request from the server by copying the content of the page from the threads VMAs through `process_vm_readv` and drops the page by `process_madvise(MAVD_DONTNEED)`, turning it in the `INVALID` state.
5. The client 1 forwards the page to the server.
6. This time the server doesn't do anything locally with the page since it's meant for client 2, so it just updates the page table owner to client 2 alone.
7. Now the server sends the page to the requesting client.
8. Finally, the userfaultfd handler can resolve the page fault by `UFFDIO_COPY` without write protection, transitioning it to `MODIFIED` state.

At the end of the sequence, client 2 is the only node that holds the page, and the server and other clients mark the page as `INVALID`.

5.4. Synchronization Implementation

Synchronization primitives are implemented by mapping new pages for coordination events, allowing the DSM userfaultfd handlers to differentiate whether a page fault comes from the computation, which triggers one of the three transactions, or from a barrier, mutex request, or join. Six additional `msg_type` values are needed for these transactions:

```
enum msg_type {
 ...
 MSG_BARRIER_HIT,
 MSG_BARRIER_RELEASE,
    MSG_LOCK_REQUEST,
    MSG_GRANT_LOCK,
    MSG_UNLOCK,
    MSG_JOIN_THREAD,
};
```

5.5. Networking Layer

The networking layer provides a uniform abstraction to the DSM runtime and supports both TCP and RDMA transports. Communication is divided into a control plane for metadata and a data plane for page transfers.

TCP message format

In the TCP backend, both planes are implemented using standard socket operations and explicit buffer copies. Each DSM message begins with a fixed size header that encodes the message type and the relevant page level metadata, followed by an optional payload containing the page contents. The header is represented in C as:

```
struct msg_info {
    int   msg_type;
    long  page_addr;
    int   page_size;
    long  msg_index;
};
```

It contains all the information to allow the receiving node to handle the message. The `msg_type` identifies the operation being requested or acknowledged, page or synchronization. The `page_addr` field carries the virtual address of the target page. With the `msg_index` it allows the server to look up the corresponding entry in its page table. The `page_size` field is used to support variable sized transfers, even though in the current design it is typically set to the system page size (for example 4096 bytes).
For messages that carry a page, the sender first writes the `msg_info` header to the socket and then writes exactly `page_size` bytes containing the page data. The receiver reads the fixed size header, interprets the fields, and then reads the following payload.

RDMA message format

In the RDMA backend, control information is transmitted as a compact command message, while page contents are delivered through one-sided RDMA writes into pre-registered memory regions. The command is represented by the following packed structure:

```
typedef struct __attribute__((packed)) {
    uint64_t target_addr;    /* client's RDMA buffer addr */
    uint64_t faulting_addr;  /* virtual page address */
    uint32_t id;             /* request id */
    uint32_t index;          /* slot index in the page list */
} rdma_cmd_msg;
```

The `target_addr` field stores the address within the client's registered RDMA memory region where the server writes the page contents. In practice the layout of these RDMA zones is static and initialized when the DSM runtime starts, so `target_addr` is mostly redundant and is kept primarily for debugging and for sanity checking that both endpoints agree on the expected buffer location.
Although the concrete encoding differs, both backends expose the same logical operations to the DSM runtime, keeping higher-level coherence logic independent of the transport mechanism. In fact the choice is done entirely at run time by adding (or not) the flag in the CRIU restore command `-rdma-enable`.

6. Evaluation

This chapter presents the hardware and software evaluation environment for CRIU-DSM and introduces two targeted microbenchmarks aimed at isolating synchronization overheads and page-granularity DSM costs.

6.1. Experimental Setup

For the experiments we used four homogeneous `c6525` nodes on CloudLab. Each reservation was dedicated exclusively to CRIU-DSM in order to eliminate interference from co-located workloads.

Hardware Platform

Each node features an AMD EPYC 7302P 16-core CPU with 32 hardware threads (SMT enabled). Nodes are provisioned with over 120 GiB of DRAM. Networking is provided by Mellanox ConnectX-5 adapters configured in RoCE mode, with an active MTU of 1024 bytes and a peak bandwidth of 25 Gbps. RDMA support includes RC transport, read and write verbs, and hardware-managed queue pairs, consistent with the standard mlx5 feature set.
The software stack consists of Ubuntu 24.04, Linux kernel 6.8.0-71-generic, rdma-core, GCC 13, and our custom release of CRIU 4.0.

CRIU-DSM Deployment

Experiments use four physical machines: one DSM Server and up to three DSM Clients. Thread distribution across nodes is expressed as

$$Server{-}Client\ 1{-}...{-}Client\ n$$

where each group denotes the number of restored threads per node. For example, 16-10-8 means: 16 restored threads on server node, 10 on client 1 and 8 on client 2.
For the best case, we chose the manual page registration in which all applications store their userfaultfd-tracked memory page addresses in `/tmp/ranges.txt` and the pages only for the barrier fault in `/tmp/dsm_barrier_pages.txt`. During restoration, on each node the DSM application registers these pages enabling page faults only for this restricted ranges, improving performance and avoiding false sharing.

Experimental Procedure

Execution initially proceeds on the server until threads are created and at a safe point. At that moment, a coordinated CRIU dump captures a consistent global checkpoint. The server then distributes the image files to

the client nodes via `scp`, and each node restores only the threads assigned to it. A DSM barrier ensures that every group of threads is up before execution resumes.
Our goal is to analyzed CRIU-DSM in the case of a maximum resource consumption, in order to simulate this in our experiments we controlled the CPU availability for each test using `taskset`. We evaluate configurations ranging from a single core to all 32 hardware threads, always prioritizing distinct physical cores before SMT to avoid hyperthreading.

6.2. Microbenchmark: POSIX Mutex and Join

To verify that CRIU-DSM preserves POSIX synchronization semantics under distributed execution, we introduce the `mutex_join` microbenchmark. This test evaluates:

1. the correctness and overhead of `pthread_mutex_lock` and `pthread_mutex_unlock`,
2. the correctness of `pthread_join` for both local and remote threads,
3. additional ordering constraints enforced by two DSM-protected write-fault pages.

Every node has a subgroup of threads, in each iteration, a thread:

1. Local lock: acquires the local POSIX mutex,
2. Global lock: triggers a DSM-level lock via a write fault on the lock page, the fault handler request permission from the server,
3. Permission granted: the fault is resolved and execution continues, threads updates a shared counter (that will cause a `GET PAGE & INVALIDATE`),
4. Global unlock: again triggers a DSM-level unlock via a write fault on the unlock page, waits for acknowledgment
5. Local unlock: releases the local mutex.

Thread completion is handled by a DSM-aware join mechanism on the server: local threads rely on standard `pthread_join`, while remote threads signal termination through DSM Clients.

Mutex–Join Results

We executed 100,000 total increments under three configurations:

- 6: all six threads on server.
- 3-3: equal split on server and one client node.
- 2-2-2: equal split on server and two clients node.

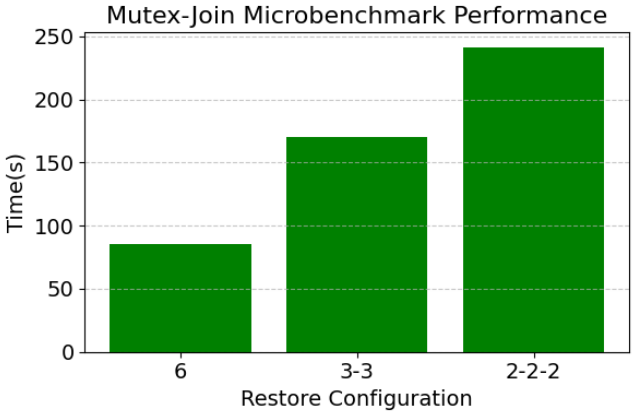

Figure 11: Mutex-Join microbenchmark.

As it was to be expected, distributed configurations have additional overhead due to global lock and unlock operations, which are serialized through the DSM server. The choice of transport is irrelevant, since the workload's bottleneck is sequential lock serialization, which dwarfs any TCP versus RDMA difference. The 2-2-2 setup further increases overhead because spreading threads across three nodes amplifies contention and triggers more write invalidations, leading to a higher number of DSM faults.

6.3. Page-Level Message Microbenchmark

In this three-node microbenchmark we quantify the cost of individual coherence actions. With reference to Fig. 10, we measure the latency of three different data paths. We assume that only one valid copy of the page

exists at any time. All three paths evaluate how long it takes to transfer 512 pages of data to the requester, depending on who owns the page:

- 1) Client–server–client: a client requests a page through the server from another client, corresponding to the scenario shown in Fig. 10.
- 2) Server–client: the server sends the requested page directly to the client.
- 3) Client–server: a client requests a page whose owner is the server.

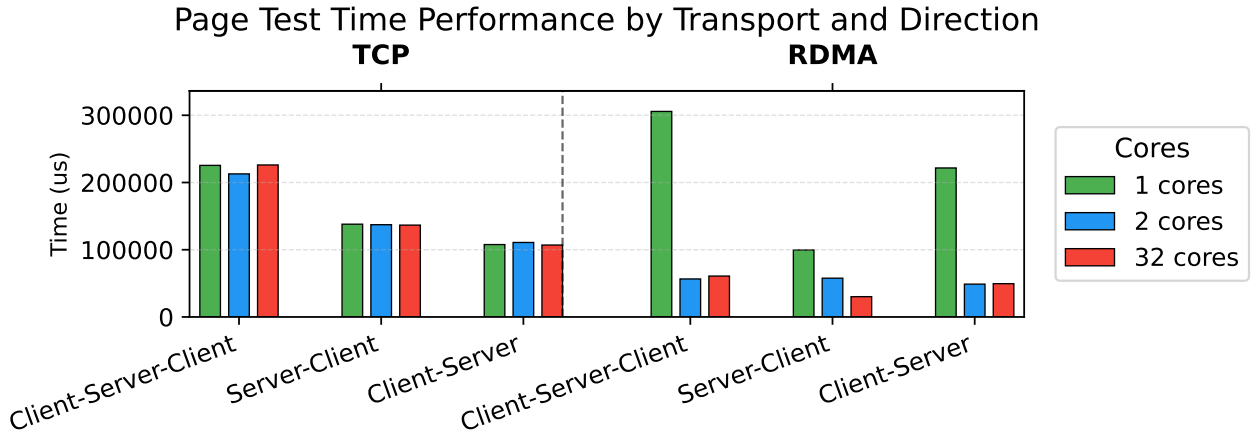

Figure 12: Page Test microbenchmark

A clear latency difference emerges between the two transports when all cores are available, shown by the red bars in Figure 12. The latency is lower in RDMA for all three primitives compared to TCP, because in this configuration, network overhead dominates. RDMA avoids socket wakeups, system calls, kernel buffer copies, and TCP/IP processing, instead relying on hardware completion queues and direct DMA into registered memory. Under CPU contention imposed via `taskset`, RDMA performance degrades substantially, while TCP latency remains almost unchanged. This is likely due to RDMA completion-queue polling, fault handling, and DSM bookkeeping being serialized on a single core, which removes the advantage of bypassing the TCP/IP stack. Allowing the server to use two cores is sufficient for RDMA to recover performance close to the 32-core case, while TCP remains very similar.
These results highlight three key points:

- With sufficient CPU resources, RDMA clearly outperforms TCP, since network latency and socket overhead dominate.
- Under severe CPU constraints, the bottleneck shifts from communication to computation, making RDMA more sensitive to core availability than TCP.
- TCP performance remains comparatively stable across core counts because its cost is dominated by kernel-level TCP/IP processing.

Overall, RDMA provides the highest performance when the DSM server has adequate CPU capacity, while TCP is more robust under extreme CPU contention.

6.4. Benchmark Characterization and Results

We used Stanford's MapReduce benchmark suite, Phoenix 2.0 [26], which consists of seven applications that differ in access patterns, synchronization frequency, data volume, and computational intensity: `Histogram`, `Kmeans`, `Linear Regression`, `Matrix Multiply`, `PCA`, `String Match`, and `Word Count`. We divided the benchmarks into two groups according to how well they performed under distributed execution with our system.

Understanding the plots

In all plots we have four default colors for the configurations, vanilla execution (teal) and local restore (purple), and every DSM vertical line has also for the CRIU restore and dump times, orange and yellow. All other colors are chosen randomly. The protocols are hatched with ".." for TCP and "X" for RDMA. On the y axis we have the time in seconds, on the x axis we have a subdivision based on the number of available cores chosen via taskset.

6.4.1 Input-Constrained, Low-Communication Benchmarks

This group includes String Match, Word Count, Histogram, and Linear Regression. Although they differ in input type and memory footprint, they exhibit very similar scaling behavior. In all cases, the input size could not be further increased, either because the source code would break for larger files or because we reached the maximum available RAM, 100 GB circa. As a result, scalability is bounded by input constraints rather than by DSM communication overhead.
Overall, these benchmarks are either embarrassingly parallel or involve very limited communication, computation dominates execution time, coherence overhead is amortized, and speedup saturates early. In all four cases, we observe the same pattern: meaningful improvements when cores are restricted, best speedup in the 2-2-2-2 TCP configuration with one core, and minimum execution time in the same configuration with two cores. Increasing cores beyond five, String Match and Linear Regression, or eight, Word Count and Histogram, does not provide further gains because parallelization is already maximized for the given input size.

String Match In this benchmark, the input is divided equally among threads, and each scans it searching for specific word matches and then prints them. This is an embarrassingly parallel benchmark because there is no communication between threads and no conflicts possible. So while it is okay to show our system thread distribution, it does not demonstrate DSM capabilities. Even at the largest input size, it remains too small for modern systems, and we had to restrict the cores, simulating a CPU resource shortage, to observe meaningful improvements with CRIU-DSM.
As shown in Figure 13, we get meaningful speedup up to five cores. As we said, the best speedup overall of 3.21x is with one core, protocol TCP, configuration 2-2-2-2. The lowest execution time is in the same configuration with two available cores. After five cores, performance is very similar, most results around 3 seconds, likely due to the absence of communication between threads. Also, confirming the microbenchmark, TCP outperforms RDMA in low-core scenarios.

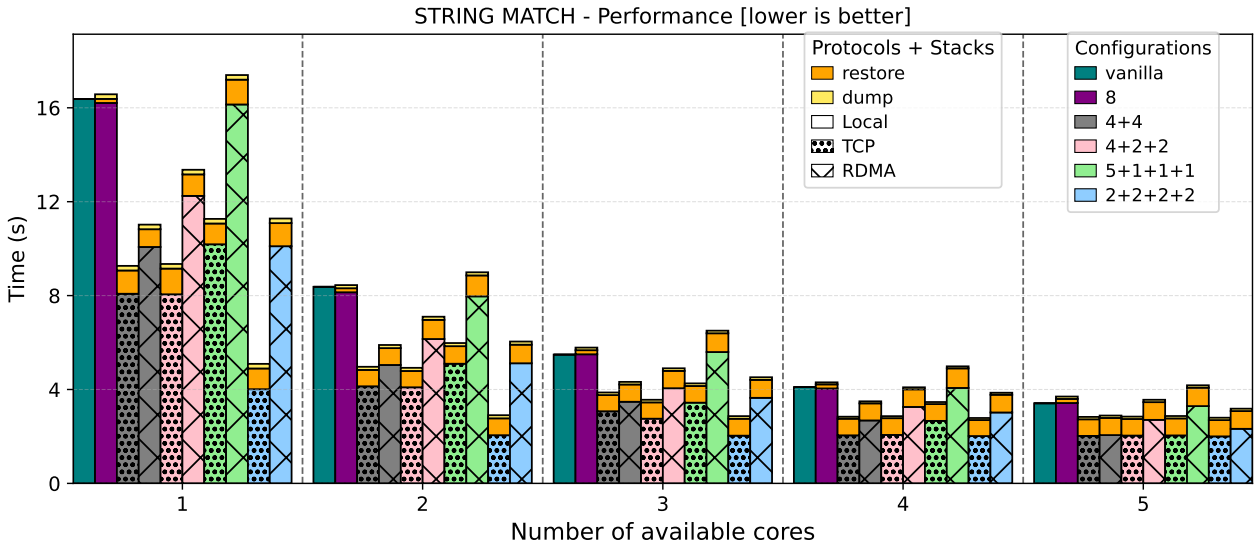

Figure 13: Performance results for String Match

Word Count It is very similar: this benchmark scans an input file linearly, but each thread needs to count every word. The MapReduce phase divides the input and merges results after each thread finishes. We modified the source code to adapt it for CRIU-DSM deployment by giving each thread its own data structure to count words, avoiding overlapping and false sharing. Since the original implementation performed a join before the final reduce, we inserted a custom barrier to synchronize threads and allow page exchange safely.
The benchmark is low in computational intensity and highly parallelizable. Page faults depend on input distribution but remain contained for non trivially repeated texts. Again, the best speedup, 4.06x, is achieved in the 2-2-2-2 TCP configuration with one core, and the minimum execution time occurs in the same configuration with two cores. Due to the limited input size, performance stabilizes around eight available cores. Also this benchmark is the best in terms of deployability: it has the lowest dump and restore compared with the execution.

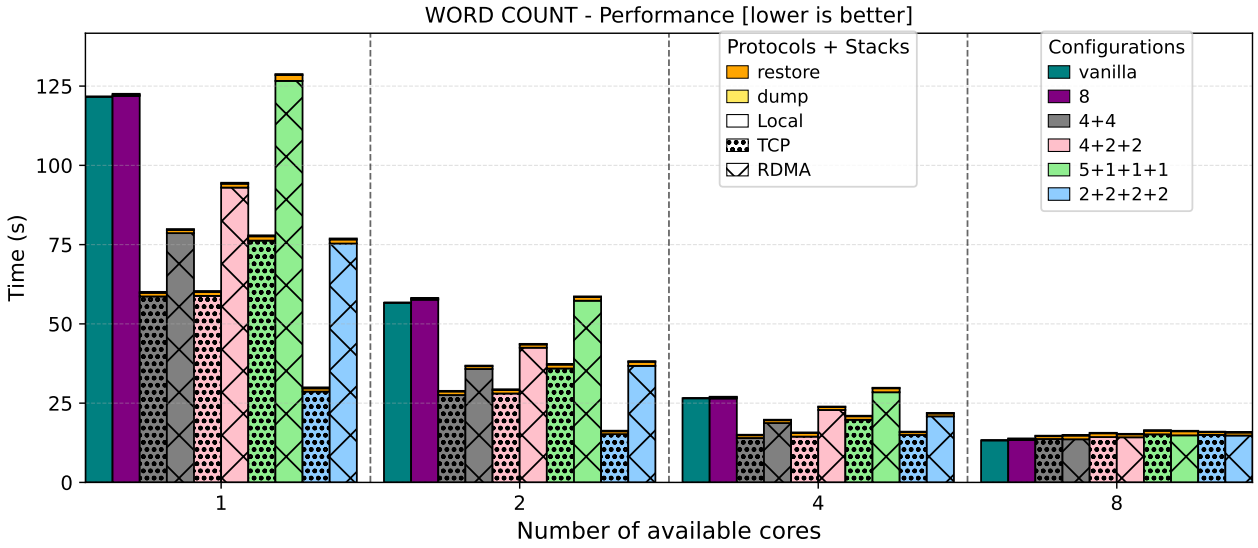

Figure 14: Performance results for Word Count

Histogram For Histogram, we created the largest file allowed by the machine, approximately 100 GB in `/dev/shm`. Further increase was impossible due to disk and RAM limits. Even if the input size grows, the histogram counter data remains fixed: each thread uses only two pages to store RGB arrays of 255 entries, 8 bytes each, independent of input size. Again, DSM communication happens only at the end, after each thread finishes its chunk, so computation dominates and coherence overhead is amortized.
The best speedup, 3.45x, is again obtained with 2-2-2-2 TCP and one core, and the minimum time is achieved with two cores. Performance plateaus after five cores because input growth is bounded. Compared to the previous benchmarks, the restoration phase is more impactful, meaning that computation scales but deployment cost becomes noticeable.

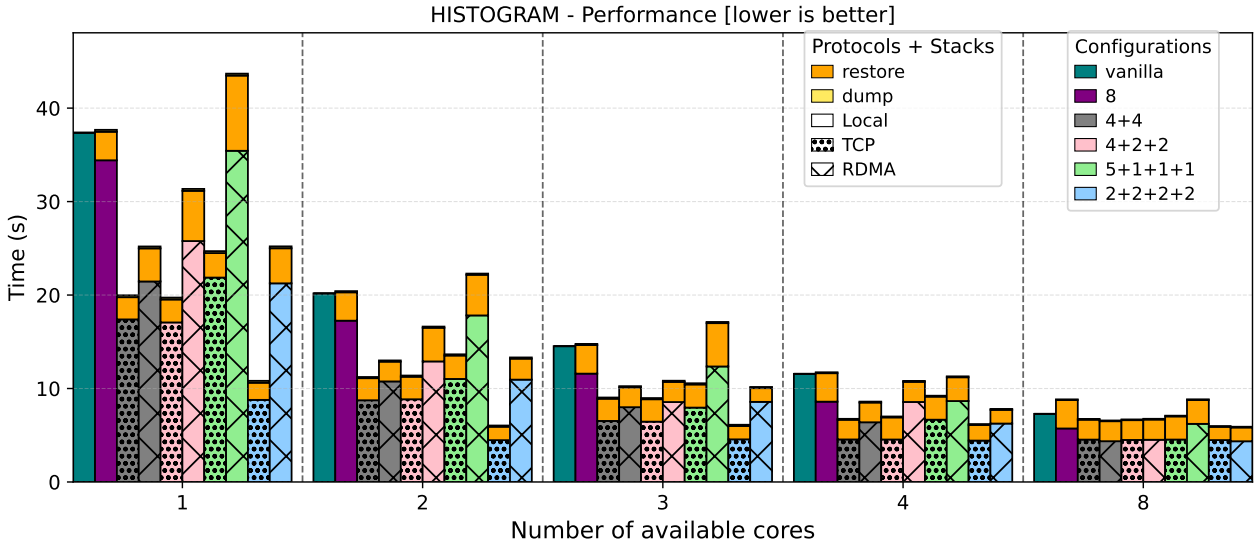

Figure 15: Performance results for Histogram

Linear Regression Linear Regression is less suitable for CRIU-DSM because it aggregates results into only five running sums, therefore each thread produces only a small output, but since CRIU-DSM operates at page granularity, one full page is still transferred, mostly empty, resulting in very low information density per transfer. With an bigger input it would be probably possible to better amortize the DSM cost even with low info per page because we would do more computation. However, we maxed the input size available on the `c6525` nodes.
Again, the best speedup, 3.16x, is achieved with 2-2-2-2 TCP and one core, and the minimum time occurs with two cores. Similarly to Histogram, restoration cost is more noticeable, especially when core availability

increases coming to roughly 37% of the total time. Although Figure 16 shows improvements up to five cores, the absolute times are very small, around four seconds, because the input could not be increased further.

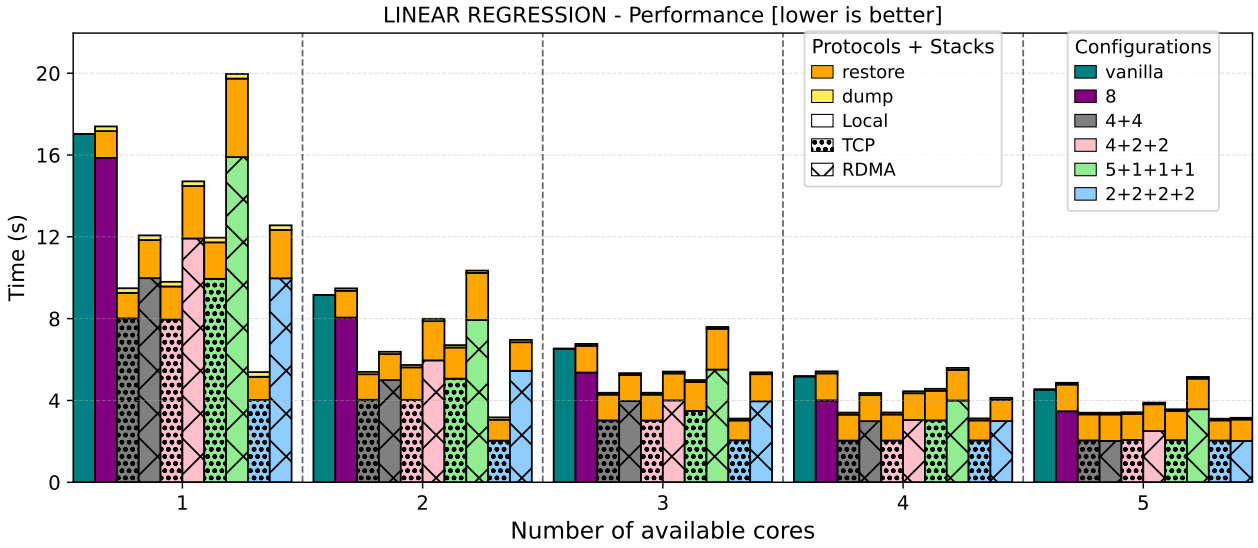

Figure 16: Performance results for Linear Regression

6.4.2 Full Power Benchmarks

Finally, we have three benchmarks that have no hardware resource constraints and can fully utilize all 32 cores available on the `c6525` nodes. In this setting, scalability behavior is clearer and RDMA begins to outperform TCP, although only by a small margin. Unlike the previous group, input size is not the limiting factor, and performance trends are determined by computational intensity and communication patterns.

PCA As in previous benchmarks, the maximum speedup occurs when cores are limited, but here we still observe improvements at 8, 16, and 32 cores. Execution time decreases significantly under CRIU-DSM; however, dump-restore overhead is impactful, reaching peaks of 47% of total execution time. Therefore, CRIU-DSM improves computation time but has very high deployment costs.
Since covariance calculation is asymmetric and the work balance is done trivially by dividing input matrix rows equally among threads, the first ones handle more computation. Therefore, the best speedups are not achieved in configurations where threads are equally divided among nodes, but in those that balance work more evenly. The best is the 12-15-36 over TCP in which there is also a clear difference between steady speedup, 2.1x, computed as vanilla execution over distributed execution, and total speedup, 1.7x, computed as vanilla execution over full deployment time.

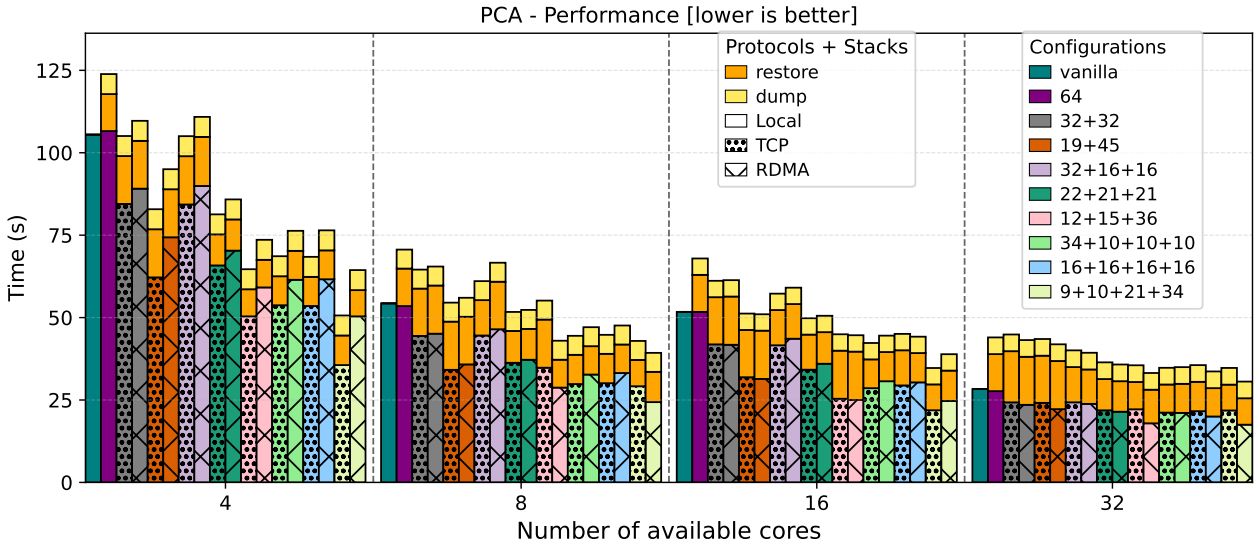

Figure 17: Performance results for PCA

Matrix Multiply We executed Matrix Multiply with a 6000x6000 input matrix size, large enough to use the whole machine and achieve the best DSM scaling. This is the best benchmark among all in this suite. Since we have a lot of computation per page, each page fault carries much more information compared to other benchmarks. Therefore, the overhead of dumping and restoring threads across nodes is matched by very information dense page exchanges. Deployment overhead is the lowest among these realistic benchmarks, circa 12% in the 32 core configuration and around 7% with 4 cores. This workload best demonstrates the conditions under which CRIU-DSM is effective, namely high computation per transferred page and limited fine grained synchronization. We get the best speedup, ranging from 3.39x with four cores available to 1.9x with all machine.

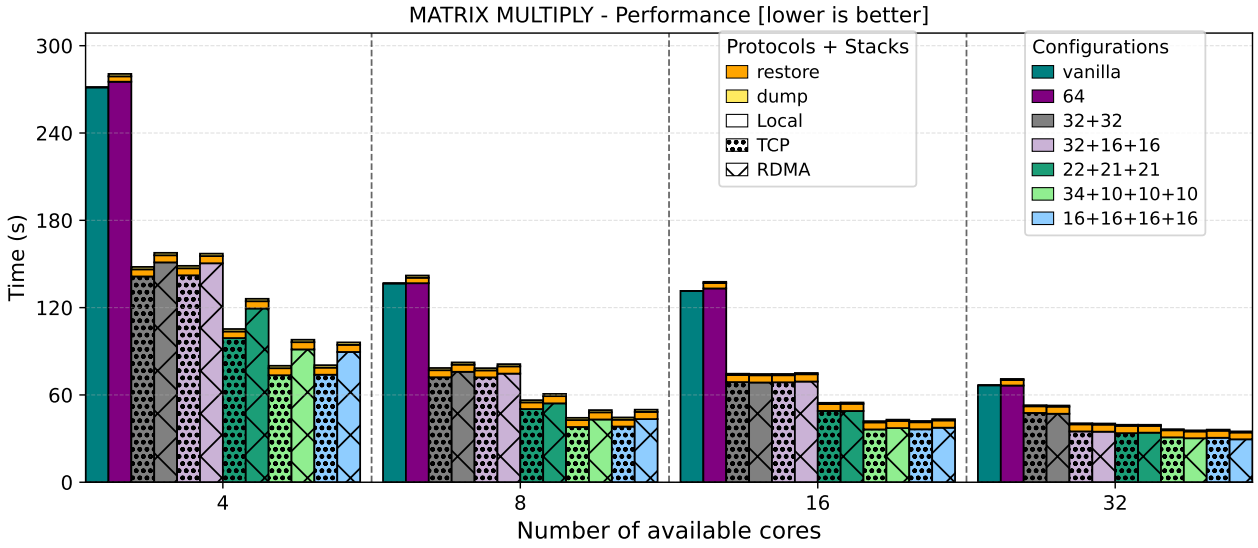

Figure 18: Performance results for Matrix Multiply

Kmeans Among all benchmarks, this is the most challenging. It is not well suited for distributed execution since threads need to repeatedly update each other every iteration during centroid computation, generating a lot of low information traffic. Also considering the fact that, depending on the input, convergence may require as few as 10 iterations or as many as 10,000, and in such cases CRIU-DSM incurs excessive page exchange overhead.
This is the only benchmark where performance degrades as nodes increase within the same core group. And also when the threads are equally divided 8-8-8-8 performs worse than asymmetric distributions such as 17-5-5-5, since in this case it's better to have more threads locally on the server so that they are easier to update instead

of distribute computation equally. For all tested inputs and core availability, normal execution outperforms CRIU-DSM.

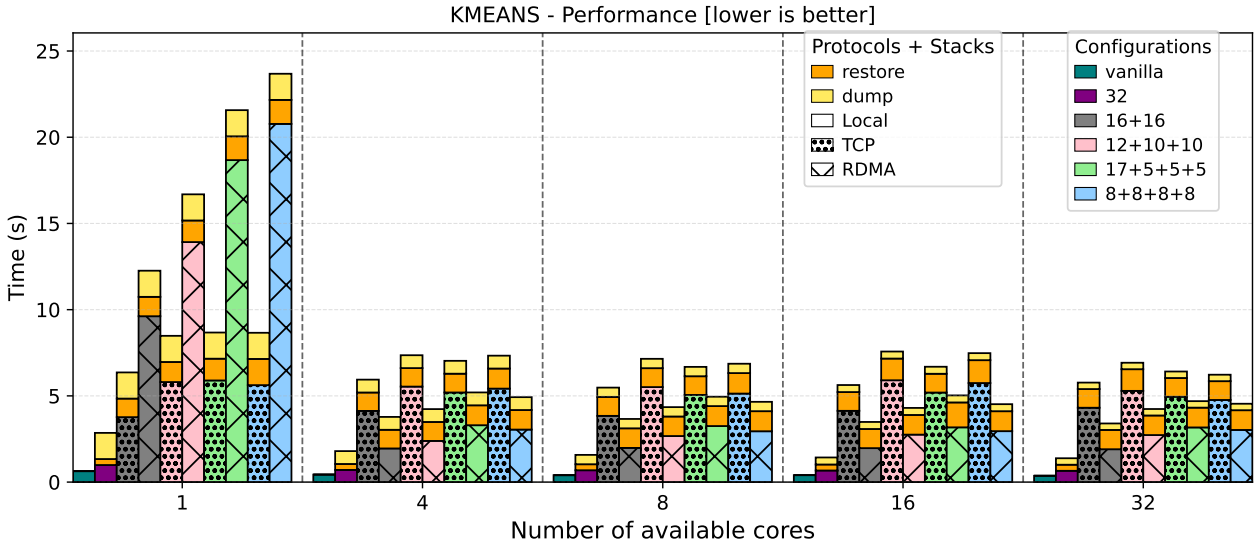

Figure 19: Performance results for Kmeans

6.4.3 Key Idea: Amortizing the Page Fault Cost

The main insight from these results is that the cost of a page fault must be amortized over either a large amount of useful information or a large amount of computation. If each transferred page carries substantial data or enables significant computation before the next fault, the overhead of CRIU-DSM is amortized over the total execution time.
Matrix Multiply demonstrates this clearly, with enough input size and appropriate thread distribution, CRIU-DSM achieves up to 1.9x real speedup at full core availability. In this case, each page exchanged contains dense numerical data that is heavily reused in computation, so the cost of dumping and remote page transfer is effectively amortized.
Kmeans, instead, is poorly suited for transparent DSM execution. The problem is not only the number of iterations, but the very low information density per transferred page: a whole page exchanged only to send a byte of information. At each iteration, the authoritative thread synchronizes at the DSM barrier, checks whether centroids were updated and propagates this information. The communication pattern generates high traffic with limited computation per transferred page. If each iteration required substantially more computation, the amortization would improve.
The remaining benchmarks lie between these two extremes. They exhibit a more balanced ratio between computation and page transfers, but when many cores are available the total work per thread becomes small, limiting achievable speedup regardless of DSM efficiency.
Overall, this benchmark suite shows that CRIU-DSM improves performance when either computation per page or information density per page is sufficiently high. When this condition holds, deployment overhead can be amortized and meaningful speedups can be achieved.

7. Conclusion

In this thesis, we presented CRIU-DSM, a userspace live thread migration and transparent distributed shared memory system. By leveraging existing Linux tools such as CRIU, and kernel APIs including `userfaultfd`, `process_vm_readv`, and `process_madvise` for cross process memory coherence and invalidation, we implemented a fully transparent userspace framework for distributing threads across physical nodes while maintaining the illusion of a shared memory system. Our results show that, with moderate modifications to the application source code to make it more DSM friendly, distribution can improve execution time with meaningful speedups.

7.1. Original contributions

In this thesis we introduced and extended the following components:

- **New modes in the CRIU 4.0 runtime.** This work builds upon the latest version of CRIU, extending the runtime while preserving the standard dump command. We introduced additional CLI commands to select client or server mode *–dsm-client, –dsm-server*, optionally enable RDMA, *–rdma-enable* with TCP as default, and configure the number of client nodes handled by the server.
- **An Intel Pin tool for thread access analysis.** This Intel Pin based tool performs fine grained code page access analysis and helps identify shared data structures and false sharing patterns. With this tool, developers can refine the source code to make it more DSM friendly and improve performance.
- **A new coherence framework with reduced context switching.** Unlike previous systems such as CRIU-RTX [31], which relied on CRIU's Compel parasite framework [2] to retrieve pages with `vmsplice` and invalidate them with `madvise`, we adopted the cleaner and more powerful `process_vm_readv` and `process_madvise` APIs. This significantly reduces overhead during page coherence operations, since it causes less context switching between kernel and userspace.
- **A more modular and maintainable codebase.** Instead of concentrating logic inside large and complex dsm_server.c and dsm_client.c files, we introduced dsm.c and dsm.h to encapsulate shared functionality. This separation improves readability, modularity, and maintainability.
- **Restore support for multiple threads per node.** Each CRIU-DSM node can restore and manage an arbitrary subset of threads from the main application, increasing flexibility and deployment versatility.
- **Transparent synchronization primitives.** By introducing custom barriers recognized by the DSM runtime as synchronization commands, we support workloads beyond embarrassingly parallel scenarios.
- **Optional page tracking.** Developers can choose the desired level of transparency. The system can run without source modifications, or selectively track specific page ranges to reduce false sharing and coherence overhead. Although this requires recompilation, it typically improves performance compared to fully automatic tracking.

7.2. Limitations

Although CRIU-DSM demonstrates a correct and usable DSM system, several limitations remain:

- **Transparency vs performance.** Full transparency is supported without application changes, but optimal performance requires reducing tracked pages and separating objects that would otherwise reside on the same page. Changing the source code, it is not always allowed or easy.
- **Pthread mutex lock performance.** Since acquiring a mutex triggers a page fault that is resolved only after server authorization, it adds context switching and latency. In DSM settings, it is preferable to partition work to minimize lock usage.
- **Network latency limitations.** RDMA reduces network latency compared to TCP, but remote memory access remains slower than local access.
- **Security problems.** CRIU-DSM applications require elevated privileges to access Linux APIs, effectively operating with root level capabilities. In addition, the Kprobe module exposes `process_madvise`, which could be restricted to CRIU-DSM processes, but still introduces potential attacks options. Also a malicious process with sufficient privileges could invoke `process_vm_readv` or `process_vm_writev` to read or modify the memory of another process. A real world deployment would therefore require strict isolation mechanisms and fine grained permission control. We acknowledge this limitation, but security concerns were outside the scope of this work.

7.3. Future Work

There are some interesting future prospects for this work:

- **RDMA cache-like table.** For `SHARED` memory pages in the Server, could reduce latency and accelerate coherence transitions if the latest pages where ready to be sent to requesting clients, instead of having to re-read them from the process.
- **Transparent tracking of shared memory regions.** By combining the Intel Pin with readelf and source code analysis, we could automatically identify and monitor shared memory at runtime without requiring manual annotation or configuration.
- **Automatic block pages transfers.** Instead of waiting for the page fault to request the page via `userfaultfd`, for some benchmarks that have a known and fixed set of stages, we could anticipate and send pages in block before they fault. We still would have to send the same amount of data, in fact the improvement is in the page coherence actions, they can be faster because they work in range mode not in fixed 4KB page, therefore instead of reading X pages one by one by X `process_vm_read`, we would need just one. It would also remove userfaultfd related latency.

In the end, this work showed that a userspace CRIU backed DSM is both correct and fast if designed well

enough. CRIU-DSM probes into the new generation of distributed multi-node execution, and the result points in the direction that it could become a powerful tool for scaling live thread if local resources get too congested.

References

[1] Criu: Checkpoint/restore in userspace. `https://criu.org`.

[2] Criu compel framework. `https://criu.org/Compel`.

[3] Criu page server framework. `https://criu.org/Page_server`.

[4] Linux kernel documentation: Kprobes. `https://docs.kernel.org/trace/kprobes.html`.

[5] Linux kernel source: mm/madvise.c. `https://elixir.bootlin.com/linux/v6.6/source/mm/madvise.c#L1493`.

[6] madvise(2) linux manual page. `https://man7.org/linux/man-pages/man2/madvise.2.html`.

[7] Openmosix project overview. `https://en.wikipedia.org/wiki/OpenMosix`.

[8] process_madvise(2) linux manual page. `https://man7.org/linux/man-pages/man2/process_madvise.2.html`.

[9] process_vm_readv(2) linux manual page. `https://man7.org/linux/man-pages/man2/process_vm_readv.2.html`.

[10] userfaultfd(2) linux manual page. `https://man7.org/linux/man-pages/man2/userfaultfd.2.html`.

[11] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 775–787, Boston, MA, July 2018. USENIX Association.

[12] Abhishek Mandar Bapat. *HetMigrate: Secure and Efficient Cross-architecture Process Live Migration*. PhD thesis, Virginia Tech, 2023.

[13] Amnon Barak, Shai Guday, and Richard G. Wheeler, editors. *The MOSIX Distributed Operating System*, volume 672 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 1993.

[14] John K Bennett, John B Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 168–176, 1990.

[15] Christopher Blackburn, Xiaoguang Wang, and Binoy Ravindran. Rave: A modular and extensible framework for program state re-randomization. In *Proceedings of the 9th ACM Workshop on Moving Target Defense*, MTD'22, page 3–10, New York, NY, USA, 2022. Association for Computing Machinery.

[16] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with rdma and caching. *Proc. VLDB Endow.*, 11(11):1604–1617, July 2018.

[17] Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations. In *USENIX Winter Technical Conference*, 1994.

[18] Frederick Douglis. Process migration in the sprite operating system. Technical report, USA, 1987.

[19] Wataru Endo, Shigeyuki Sato, and Kenjiro Taura. Menps: A decentralized distributed shared memory exploiting rdma. In *2020 IEEE/ACM Fourth Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM)*, pages 9–16, 2020.

[20] Mark S Gordon, D Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, and Xu Chen. {COMET}: Code offload by migrating execution transparently. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, pages 93–106, 2012.

[21] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: a hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 417–433, New York, NY, USA, 2022. Association for Computing Machinery.

[22] Akihiro Hayashi, Sri Raj Paul, Max Grossman, Jun Shirako, and Vivek Sarkar. Chapel-on-x: Exploring tasking runtimes for pgas languages. In *Proceedings of the Third International Workshop on Extreme Scale Programming Models and Middleware*, ESPM2'17, New York, NY, USA, 2017. Association for Computing Machinery.

[23] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[24] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, page 3–14, New York, NY, USA, 2015. Association for Computing Machinery.

[25] Sang-Hoon Kim, Ho-Ren Chuang, Robert Lyerly, Pierre Olivier, Changwoo Min, and Binoy Ravindran. Dex: scaling applications beyond machine boundaries. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 864–876. IEEE, 2020.

[26] Kozyraki. Phoenix-2.0 a mapreduce suite. `https://github.com/kozyraki/phoenix/tree/master/phoenix-2.0`, 2025. GitHub repository.

[27] Kai Li. Ivy: A shared virtual memory system for parallel computing. *ICPP (2)*, 88:94, 1988.

[28] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3 edition, 2010.

[29] Robert Lyerly, Xiaoguang Wang, and Binoy Ravindran. Dynamic and secure memory transformation in userspace. In *European Symposium on Research in Computer Security*, pages 237–256. Springer, 2020.

[30] Haoran Ma, Yifan Qiao, Shi Liu, Shan Yu, Yuanjiang Ni, Qingda Lu, Jiesheng Wu, Yiying Zhang, Miryung Kim, and Harry Xu. DRust: Language-Guided distributed shared memory with fine granularity, full transparency, and ultra efficiency. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 97–115, Santa Clara, CA, July 2024. USENIX Association.

[31] Mohamed Husain Noor Mohamed. Criu-rtx: Remote thread execution using checkpoint/restore in userspace. Master's thesis, Virginia Polytechnic Institute and State University, 2023.

[32] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-Tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, Santa Clara, CA, July 2015. USENIX Association.

[33] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 306–324, New York, NY, USA, 2017. Association for Computing Machinery.

[34] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with In-Network cache coherence. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 277–292. USENIX Association, February 2021.

[35] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, et al. Titanium: a high-performance java dialect. *Concurrency and Computation: Practice and Experience*, 10(11-13):825–836, 1998.

[36] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. Upc++: a pgas extension for c++. In *2014 IEEE 28th international parallel and distributed processing symposium*, pages 1105–1114. IEEE, 2014.

Abstract in lingua italiana

A causa del rallentamento della legge di Moore nelle prestazioni a singolo core, dovuto a limiti di potenza e vincoli termici, scalare verticalmente le applicazioni è diventato più difficile. A seguito del rapido aumento della larghezza di banda di rete e memoria, è possibile riesaminare la scalabilità orizzontale dei thread con il supporto di sistemi di memoria condivisa distribuita (DSM).
Questa tesi introduce CRIU-DSM, un sistema DSM in user space che distribuisce live thread POSIX su più macchine fisiche.
L'approccio proposto estende Checkpoint Restore In Userspace, CRIU, per distribuire un sottogruppo arbitrario di thread su nodi fisici e sfrutta API del kernel Linux per mantenerne la coerenza. Abbiamo sviluppato un protocollo centralizzato stile MSI con granularità di pagina in write invalidation, utilizzando `userfaultfd` per intercettare in user space le fault di pagine mancanti e di protezione in scrittura, `process_vm_readv` per leggere pagine tra processi, e `process_madvise` per invalidarle. La soluzione preserva la semantica standard POSIX `pthread`. CRIU-DSM può essere distribuito in modalità completamente trasparente per applicazioni non modificate oppure in modalità manuale basata su regioni condivise personalizzate tramite `mmap`, per migliorare le prestazioni mitigando il false sharing.
Il sistema è valutato su CloudLab utilizzando la suite Phoenix MapReduce di Stanford su trasporti TCP e RDMA. I risultati mostrano che le prestazioni migliorano quando la condivisione in scrittura è limitata e è elevato il rapporto tra computazione e page fault. Matrix Multiply raggiunge un'accelerazione fino a 3.39x e PCA fino a 2.1x, mentre applicazioni che causano invalidazioni eccessive, come Kmeans per gli aggiornamenti dei centroidi, risultano più lenti dell'esecuzione nativa perché frequenti aggiornamenti e una bassa intensità computazionale impediscono l'ammortizzazione degli overhead del DSM. Complessivamente, i risultati indicano che una ridistribuzione live dei thread basata su CRIU combinata con la gestione delle page faults in user space fornisce una base pratica e impiegabile per un DSM trasparente.

Parole chiave: CRIU, userfaultfd, RDMA, sistema distribuito di memoria condivisa, migrazione di live threads

Acknowledgements

I would like to thank Professor Marco Domenico Santambrogio for his work in sustaining the PoliMi–UIC exchange program. This opportunity profoundly changed my Master's experience and contributed greatly to my growth as an engineer and as a person overall.
I am indebted to my family for their unconditional support, without which this thesis would not have been possible. Finally, I thank my friends, whose presence made this journey immeasurably better.

Edoardo D'Alessio