Stretch: A Fault-Driven DSM Runtime for Distributed Multithreaded Applications

Edoardo D'Alessio*
University of Illinois Chicago
Chicago, USA

Mohamed Husain Noor Mohamed*
Virginia Tech
Blacksburg, USA

Xiaoguang Wang
University of Illinois Chicago
Chicago, USA

Binoy Ravindran
Virginia Tech
Blacksburg, USA

Abstract

Recent Linux memory-management interfaces make it practical to revisit distributed shared memory (DSM) as a deployable runtime substrate for conventional multithreaded software. We present Stretch, a userspace fault-driven page-granularity DSM runtime that combines `userfaultfd`-based fault interception, centralized MSI-style coherence (i.e., *Modified*, *Shared*, or *Invalid*), and CRIU-based thread placement to extend a process across multiple machines. Missing-page and write-protection faults are translated into fetch, invalidation, and ownership-transfer operations, while distributed barriers and coarse-grained mutexes reuse the same mechanism. Stretch supports both automatic tracking of anonymous regions and an explicit tracked-region mode that focuses coherence on genuinely shared memory.

We evaluate Stretch on coherence microbenchmarks and seven Phoenix workloads on four CloudLab servers. The results show that the computation-to-page-fault ratio is the dominant performance predictor. Compute-intensive workloads such as Matrix Multiply achieve up to 3.39× speedup, whereas fine-grained write sharing in KMeans triggers invalidation storms that defeat page-granularity DSM. RDMA reduces key coherence operations by up to 4× relative to TCP when the server has sufficient CPU provisioning, but its advantage largely disappears when the centralized server is CPU-bound. These results identify both the practical regime in which a Linux-based fault-driven DSM is effective and the limitations that remain fundamental at page granularity.

CCS Concepts: • **Software and its engineering** → **Distributed memory**; **Virtual memory**.

*The first two authors contributed equally to this work and are listed alphabetically by last name.

ISMM '26, Boulder, CO, USA

ACM ISBN 979-8-4007-2720-7/2026/06
https://doi.org/10.1145/3814942.3816130

Keywords: Distributed Shared Memory, Page-Based Memory Coherence, Userspace Memory Management, Fault-Driven Protocols

ACM Reference Format:
Edoardo D'Alessio, Mohamed Husain Noor Mohamed, Xiaoguang Wang, and Binoy Ravindran. 2026. Stretch: A Fault-Driven DSM Runtime for Distributed Multithreaded Applications. In *Proceedings of the 2026 ACM SIGPLAN International Symposium on Memory Management (ISMM '26), June 16, 2026, Boulder, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3814942.3816130

1 Introduction

Modern compute and memory intensive applications increasingly outgrow the performance, memory capacity, and energy budget of a single machine [7, 19, 24, 26, 38]. Scaling such applications across multiple nodes is therefore attractive for both throughput and resource efficiency, especially when clusters combine powerful servers with lower-power machines. Yet this form of scale-out remains difficult for conventional multithreaded software. Existing applications are written against a shared-memory abstraction, and moving them to distributed environments typically requires either rewriting them for message passing [8, 10] or relying on substantial kernel support [17, 20]. Both approaches reduce deployability and raise the barrier to adoption.

Distributed shared memory (DSM) offers a more natural abstraction by preserving the shared-memory programming model while allowing memory to span machine boundaries [19]. However, classic and modern DSM systems often depend on one or more strong assumptions: kernel modifications, specialized hardware, language or compiler support, or restrictive execution models [2, 4, 8, 17, 20, 22, 37]. As a result, they leave open a practical systems question that has become increasingly relevant with the availability of Linux user-space memory-management primitives: *how far can recent Linux memory-management interfaces take a deployable DSM runtime for existing multithreaded applications?*

This question is timely because the hardware landscape has shifted in DSM's favor. While DRAM access latency

has remained largely flat over the past three decades (approximately 200–250 CPU cycles), network latency has improved by orders of magnitude: modern RDMA interconnects achieve sub-microsecond round trips, bringing remote memory access within an order of magnitude of local DRAM [7, 11]. At the same time, Linux has introduced userspace interfaces (most notably `userfaultfd` [13], since Linux 4.3) that allow page faults and protection changes to be handled entirely in user space, opening a design space that was not available to classic DSM systems.

This paper answers that question with Stretch, a fault-driven DSM runtime for distributed execution of multithreaded applications on commodity Linux. Stretch is built around recent Linux memory-management interfaces to intercept missing-page and write-protection faults in user space and to drive coherence actions [13]. At page granularity, Stretch maintains coherence across nodes through a centralized MSI-style protocol [28], combining fault-driven page fetch, invalidation, and ownership transfer using existing Linux facilities. In this sense, Stretch should be viewed not merely as a thread offloading mechanism, but as a deployable memory-management runtime that extends a process's shared address space beyond a single machine.

To make this memory abstraction usable for existing software, execution contexts must also be distributed across machines. Stretch addresses this by integrating with Linux CRIU (Checkpoint/Restore In Userspace) [30] for selective thread checkpointing and restoration: individual threads can be extracted from a running process and resumed on remote nodes, where they participate in the same logical address space through the DSM runtime. Optionally, Stretch can transform thread contexts across instruction set architectures (ISAs) to support heterogeneous platforms [1, 2, 23]. While these placement mechanisms broaden the system's applicability, our evaluation focuses primarily on characterizing the performance envelope and deployment tradeoffs of the fault-driven memory-coherence runtime itself.

Overall, this paper makes the following contributions:

- We design and implement Stretch, a fault-driven page-granularity DSM runtime for existing POSIX-threaded applications on commodity Linux, with both automatic and explicit tracked-region modes. The code is publicly available at https://popcornlinux.org/stretch/.
- We show how `userfaultfd`-driven page faults, centralized MSI-style coherence, cross-process memory operations, and distributed barriers and coarse-grained mutexes can be composed into a practical fault-driven coherence protocol.
- We characterize when page-based DSM is effective through evaluation on seven benchmarks and coherence microbenchmarks, identifying the computation-to-page-fault ratio as a key performance predictor and exposing the tradeoffs of page-granularity coherence for different sharing patterns.
- As a secondary use case, we show how the same runtime can support selective thread placement and optional heterogeneous-ISA execution via CRIU-based checkpoint/restore.

2 Background and Problem Setting

This section summarizes the memory-management mechanisms that make Stretch possible and clarifies the problem setting addressed in this paper. We also identify the specific substrate needed to realize a coherent shared-memory runtime for existing multithreaded applications on Linux.

Distributed Shared Memory. Distributed Shared Memory (DSM) provides the abstraction of a single shared address space across physically separate machines [19]. Under this abstraction, threads executing on different nodes can access shared data using ordinary loads and stores, while the runtime maintains coherence and propagates updates behind the scenes. DSM designs differ primarily in their coherence granularity, consistency semantics, and implementation layer. Hardware-supported DSM can offer low-latency coherence but typically depends on specialized interconnects and limited deployment environments [36]. Software DSM, by contrast, implements coherence at the operating-system, language-runtime, middleware, or user level, trading lower hardware assumptions for higher runtime overhead and more visible policy decisions [4, 5, 7, 12, 17, 26].

For this paper, the important observation is that coherence is fundamentally a memory-management problem: the runtime must decide when a page is valid, who owns the writable copy, when stale copies must be invalidated, and how missing data is fetched on demand. Stretch adopts a page-granularity software DSM design because page faults and page protection are the natural control points exposed by stock Linux user-space interfaces. This choice favors deployability and transparency, but it also makes protocol costs (such as invalidation traffic, ownership transfer, and false sharing) central to the system design.

Linux User-Space Memory-Management Interfaces. A key enabler for Stretch is `userfaultfd`, introduced in Linux 4.3, which allows user-space software to intercept and resolve page faults on registered memory regions [13]. When a thread accesses an unmapped or write-protected page inside such a region, the kernel notifies a user-space handler, which can then resolve the fault by supplying page contents, changing protection state, or delaying completion until external work finishes. This interface has been used for demand paging, live migration, and checkpointing [30], and it is the primary mechanism that allows Stretch to turn memory accesses into protocol events without requiring kernel modifications.

Stretch also relies on additional Linux interfaces to manipulate address spaces from user space. The `process_vm_readv` and `process_vm_writev` system calls support direct cross-process memory reads and writes, allowing monitors to extract or modify process state without full duplication [16]. The `madvise` family provides page-level control over memory contents and reuse, which Stretch leverages to invalidate stale pages and force subsequent accesses to re-fault so that coherence can be re-established [14]. Together, these interfaces provide the low-level substrate for a fault-driven DSM runtime: `userfaultfd` detects when access cannot proceed locally, while `madvise` and cross-process memory operations support invalidation, refill, and page-state transitions.

Tracked Regions. `userfaultfd` only reports faults for regions that the runtime explicitly registers. A practical DSM therefore needs a policy for deciding which virtual-memory areas participate in coherence. Stretch supports two such policies. In *automatic* mode, the runtime registers anonymous regions after restore. In *explicit* mode, the application or benchmark supplies the shared ranges and synchronization pages that should participate in DSM coherence. The explicit mode avoids coherence traffic on thread-private heaps and allocator metadata; we use it in the evaluation so that the reported faults correspond to intended sharing rather than unrelated memory activity.

CRIU and Selective Thread Placement. CRIU (Checkpoint/Restore In Userspace) captures a running process into a collection of image files containing registers, memory mappings, file descriptors, thread-local storage (TLS), and other execution state, and can later restore that state on a compatible system [30]. CRIU uses a `protobuf`-based image format that can be inspected and edited via CRIU's Image Tool (CRIT) [32], which Stretch leverages to manipulate per-thread state for selective restoration. Existing CRIU usage primarily targets whole-process checkpointing, migration, and fault tolerance. In Stretch, CRIU serves a different purpose: it is the execution-state substrate that makes the DSM abstraction usable for existing multithreaded software.

Specifically, Stretch builds on CRIU's image format to isolate and rewrite per-thread state so that selected threads can be restored on remote nodes while others continue locally. This selective placement capability is important because distributed execution does not arise from memory management alone; it also requires the runtime to place execution contexts on different machines while preserving access to a common logical address space. In Stretch, CRIU therefore complements the DSM layer: CRIU moves execution contexts, while the DSM runtime maintains coherent access to shared pages after those contexts are distributed.

Heterogeneous Execution Contexts. Prior work has shown that execution contexts can be migrated across ISAs by combining compiler-assisted multiversioning, ABI-aware state translation, and runtime rewriting of thread contexts [1, 2, 23]. These systems address the problem of preserving a thread's control state, register contents, stack layout, and calling conventions when execution moves between heterogeneous processors. In practice, this requires both static support, such as generating compatible code variants for different architectures, and dynamic support for rewriting saved contexts so that restored threads can resume correctly on a different ISA. Stretch leverages these techniques to support cross-ISA distributed execution.

Problem Setting. Taken together, these mechanisms define the problem setting for Stretch. We consider multithreaded applications written against a conventional shared-memory model and seek to execute their threads across multiple machines without changing their fundamental programming model. The runtime must therefore: (1) detect when a memory access cannot be satisfied from a valid local copy, (2) fetch or transfer page ownership on demand, (3) invalidate stale copies when writes occur, and (4) continue to do so while threads are selectively placed on different nodes, and (5) support synchronization primitives (barriers, mutexes) that span distributed threads. These requirements motivate the page-based, fault-driven design presented in the next section.

3 Design and Implementation

This section presents the design of Stretch as a fault-driven memory-coherence runtime for existing multithreaded applications. The key challenge is to preserve a coherent shared-memory abstraction after threads are placed on different machines. To do so, Stretch combines four elements: (1) selective checkpointing and restoration of thread execution contexts, (2) a page-granularity DSM protocol driven by user-space fault handling, (3) a distributed synchronization mechanism that maps the barriers and mutexes used by our workloads onto the same fault-driven substrate, and (4) implementations for invalidation, page extraction, and optional cross-ISA thread placement.

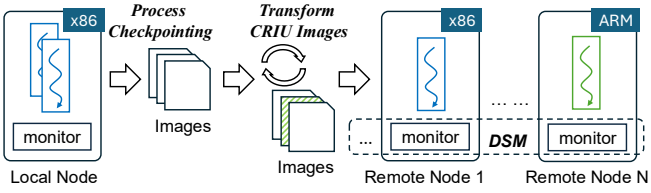

Figure 1. Overview of Stretch's workflow.

Figure 1 illustrates this workflow. Stretch first checkpoints a running multithreaded process using CRIU, producing a set of image files that encode per-thread register state, stacks, TLS, memory mappings, and process metadata [33]. It then selects a subset of threads for remote placement, *rewrites*

the corresponding images so that those threads can be restored independently, and transfers them to destination nodes. Once threads resume across multiple machines, Stretch registers shared-memory regions with `userfaultfd` and manages all subsequent missing-page and write-protect faults in user space. In this way, thread placement and page coherence are composed into a single runtime that extends the shared-memory abstraction across machine boundaries.

3.1 Execution Placement via CRIU

Before the DSM layer can maintain coherence, execution contexts must be distributed across machines. Stretch achieves this by checkpointing a running application with CRIU and rewriting the resulting image files so that selected threads can be restored independently on remote nodes. CRIU normally treats a multithreaded process as a single restoration unit; Stretch modifies the process-hierarchy metadata so that selected threads are decoupled and can be reconstructed on destination nodes without restoring the entire original process. Optionally, Stretch can leverage cross-ISA thread migration techniques [1, 2, 23] to translate thread contexts for heterogeneous platforms.

The output of this phase is a set of execution contexts placed across machines, but not yet a functioning distributed program. After restoration, local and remote threads still need coherent access to shared pages—the central challenge addressed by the DSM layer described next.

3.2 Page-Based DSM Design

Once threads are distributed, Stretch provides a page-based distributed shared-memory abstraction that allows them to operate over a logically unified address space while physically executing on different nodes. The design goal is not to emulate hardware cache coherence, but to preserve the shared-memory programming model for multithreaded software using page faults and page protection as the available control points.

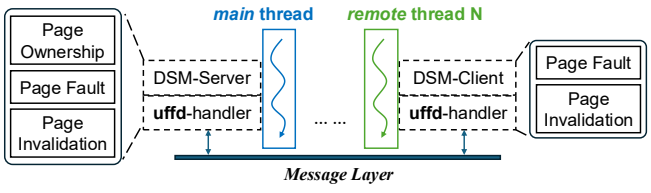

Figure 2. Overview of DSM components in Stretch.

Figure 2 shows the main components. On the primary node, a userspace monitor process hosts a DSM server and a `userfaultfd` handler that intercepts page faults from the local thread. The DSM server maintains global metadata for shared pages, including ownership, sharing state, and pending coherence actions. Each remote node runs a symmetric monitor process containing a DSM client and its own `userfaultfd` handler. Remote clients forward local faults and invalidation events to the server and apply the server's decisions on behalf of the corresponding thread. All control and data messages, including page requests, page payloads, and invalidation notifications, flow through the message layer between the server and clients.

Memory model and page ownership. Stretch distinguishes between pages that are private to one execution context and pages that may be accessed across nodes after thread placement. Only the latter are managed by the DSM runtime. For these shared pages, Stretch uses page granularity as the coherence unit because page presence and page permissions are the mechanisms exposed by Linux user-space interfaces. This choice favors deployability and transparency, but it also means that protocol overhead is governed by page movement, invalidation frequency, and false sharing.

Tracked regions. Because `userfaultfd` only reports page faults for registered virtual-memory areas, Stretch must decide which restored mappings are placed under DSM control. Stretch provides two registration modes. In *automatic mode*, the runtime scans the restore threads' memory map after restore and registers all eligible anonymous mappings with `userfaultfd`, requiring no application annotations but potentially including thread-private heap regions and allocator metadata that are not intentionally shared across nodes. In *explicit mode*, the application or runtime specifies the shared data ranges and dedicated synchronization pages to register. This mode avoids tracking obviously private regions and reduces unnecessary coherence traffic. All performance results in Section 4 use the explicit mode so that measured faults more closely reflect intended inter-node sharing.

MSI-style page states. Stretch uses a centralized MSI-style protocol [28] to maintain consistency across nodes. Each shared page is in one of three states: *Modified (M)*, *Shared (S)*, or *Invalid (I)*. A page in the *Modified* state is writable on exactly one node, which holds the authoritative copy. A page in the *Shared* state may be replicated on multiple nodes, but those copies are read-only. A page in the *Invalid* state is either absent or stale on a node and must be revalidated before access can continue. The centralized DSM server serializes ownership changes, tracks which nodes hold valid copies, and resolves conflicting requests.

Fault-driven operation. A page access becomes a DSM event only when it cannot be satisfied locally. Stretch relies on `userfaultfd` to surface two cases to user space: missing-page faults and write-protect faults. Missing-page faults occur when a page is not currently resident or has been invalidated. Write-protect faults occur when a thread attempts to write to a read-only shared copy. In both cases, the local

`userfaultfd` handler converts the fault into a DSM protocol request and sends it to the server.

For a read access to an invalid page, the server identifies the current owner or another valid holder, retrieves the page contents, installs a readable copy at the requester, and updates the metadata so that both nodes may hold the page in the *Shared* state. For a write access to a shared page, the server invalidates all other readable copies, waits until those copies are revoked, and then grants exclusive ownership to the requester, transitioning that node's copy to *Modified*. For a write access to an invalid page, the server first obtains the latest contents from the current owner if necessary, then performs the same ownership transfer and invalidation sequence before allowing the write to proceed. This protocol ensures that exactly one node holds a writable copy of a shared page at a time.

Protocol message types. The fault-driven protocol uses three message types that correspond to the MSI state transitions above. A `MSG_GET_PAGE_DATA` request is issued on a read fault to an invalid page: the server retrieves the page from the current owner via `process_vm_readv`, applies write protection on the owner's copy (M→S), installs a shared copy at the server, and forwards the page to the requester (I→S). In this way, the server retains a cached copy when transitioning to the Shared state, reducing subsequent read-fault latency for future pages accesses.
A `MSG_SEND_INVALIDATE` request is issued on a write fault to a shared page: the server sends invalidation commands to all other holders, waits for acknowledgments confirming each node has discarded its copy via `madvise(MADV_DONTNEED)`, and then grants exclusive ownership to the requester (S→M). A `MSG_GET_PAGE_DATA_INVALIDATE` request combines both: for a write fault to an invalid page, the server retrieves the page from the owner, invalidates all other copies, and grants exclusive ownership to the requester in a single round (I→M).

Design implications. The protocol is intentionally centralized. This keeps the design simple and makes coherence decisions explicit at the page level, which is useful for a deployable userspace runtime. The tradeoff is that the DSM server lies on the control path for ownership transfers and invalidations, making server CPU resources and sharing patterns important determinants of performance. In particular, write-intensive workloads and false sharing amplify coherence traffic, while read-dominant or coarse-grained sharing patterns are more favorable.

Invalidation mechanisms. A key design choice is how stale pages are invalidated inside the target process. Our current prototype uses `process_madvise()` [15] together with a lightweight kprobe-based helper to discard remote pages from the monitor without injecting code into the target process; this is the configuration used in the evaluation. We also maintain a slower fully user-space fallback based on CRIU's Compel utility [31], which injects a helper into the target process to perform `madvise(MADV_DONTNEED)` locally. Both mechanisms expose the same protocol semantics; they differ only in how page discard is triggered.

By combining user-space fault interception, centralized page-state management, and explicit invalidation, Stretch realizes a coherent shared-memory abstraction across nodes while keeping the coherence control path in user space.

3.3 Distributed Synchronization

Distributing threads across machines introduces a challenge for synchronization primitives that originally operated within a single address space. Stretch addresses this by mapping existing POSIX synchronization semantics onto the fault-driven DSM protocol, allowing barriers and mutexes to function across distributed threads without changing the application level synchronization API.

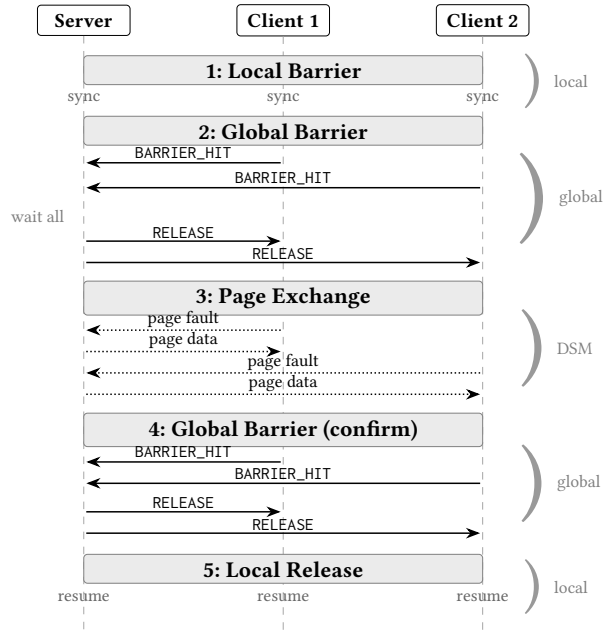

Figure 3. Five-phase distributed barrier protocol. Representative threads trigger faults on a dedicated barrier page; the server coordinates global synchronization. Phase 3 uses DSM page faults to prefetch shared pages.

Barrier protocol. Stretch implements a five-phase barrier protocol that leverages `userfaultfd`-registered *barrier pages*—dedicated pages whose faults are interpreted as synchronization events rather than coherence requests. Figure 3 illustrates the protocol:

1. **Local barrier.** All threads within each node synchronize at a local `pthread_barrier`, ensuring that intra-node computation is complete before any global coordination begins.

2. **First global barrier.** A representative thread on each node triggers a read fault on the barrier page, generating a `BARRIER_HIT` message to the DSM server. The server waits until all nodes have reported before proceeding.
3. **Page exchange.** After global alignment, threads on each node issue page-fault-driven requests to fetch any shared pages needed for the next computation phase. This phase ensures that all nodes hold consistent copies of shared data before resuming.
4. **Second global barrier.** Representative threads fault again on the barrier page to signal that page exchanges are complete. The server confirms all nodes have finished before releasing.
5. **Local release.** Each node releases its local threads, and computation resumes.

This design reuses the existing `userfaultfd` fault-handling infrastructure: barrier pages are simply pages whose fault handlers dispatch synchronization messages rather than coherence requests. No additional communication channel or kernel mechanism is required.

Mutex support. For POSIX mutexes, Stretch follows the same page-fault-driven approach as barriers. A thread first acquires the mutex locally, then triggers a write fault on a dedicated lock page, which the DSM server interprets as a `MSG_LOCK_REQUEST`. The server serializes these requests and grants the lock to exactly one client at a time. On unlock, the thread triggers a second fault on the same page, signaling a `MSG_UNLOCK` request to the server, after which the local lock is released. A key property of this design is transparency: since the synchronization is mediated entirely through page faults, the application code requires no modification and remains functional even without Stretch deployed, as the kernel handles the faults normally in that case. The cost of each lock acquisition includes a full page-fault and ownership-transfer round trip, making this mechanism best suited for coarse-grained critical sections where the lock-hold time dominates the acquisition overhead.

The current prototype implements distributed barriers and coarse-grained mutexes, which are sufficient for the workloads we evaluate. Condition variables and reader-writer locks are not yet implemented in the distributed runtime and remain future work.

3.4 Communication Architecture

Stretch uses a hub-based communication topology in which all DSM clients connect exclusively to the centralized server, and clients never communicate directly with one another. This design keeps the number of connections linear in the number of nodes (n connections for n clients) and ensures that the server has a consistent, serialized view of all coherence traffic.

On the server side, one dedicated communication thread is spawned per connected client, enabling parallel handling of concurrent requests from different nodes. Each client connection is protected by a per-client lock: when any server thread needs to send a message to a particular client, it acquires that client's lock, transmits the message, and releases the lock. This strategy guarantees serialized message ordering to each client without requiring a global lock that would serialize all server activity.

On the client side, two threads handle DSM operations: a `userfaultfd` handler thread that intercepts page faults from the application and converts them into DSM requests, and a receiver thread that processes incoming messages from the server (MSI commands, page data, barrier signals). Both TCP and RDMA transports share this threading model; the transport layer is abstracted behind a common message interface, allowing the same coherence protocol to operate over either transport without changes to the protocol logic.

3.5 Implementation Details

We implemented Stretch by extending CRIU and integrating a userspace DSM runtime to support thread-level distribution across nodes. At a high level, our implementation has four pieces: (1) selective thread restoration, (2) fault interception and server-side page tracking, (3) page invalidation and extraction, and (4) optional cross-ISA context transformation.

Selective thread restoration. To control which threads are reconstructed on each node, we modify CRIU's Image Tool (CRIT) to parse and edit `pstree.img`, which encodes the process and thread hierarchy in a `protobuf`-based JSON representation [21, 32]. By removing selected thread identifiers from this hierarchy on one node and constructing corresponding image sets for other nodes, Stretch determines the placement of threads after restoration.

To restore a child thread independently of the original main thread, Stretch also adjusts the per-thread core images generated by CRIU. In particular, we copy the `tc` (task context) field from the main thread's `core-<tid>.img` file into the `core` images of selected child threads. This change allows CRIU's `restore` operation to treat the chosen thread as a self-contained process, effectively decoupling thread migration from the original process hierarchy.

Fault interception and page tracking. Shared-memory regions are registered with `userfaultfd` so that missing-page and write-protect faults can be intercepted in user space. The local `userfaultfd` handler runs as part of the monitor process and executes an event loop that receives fault notifications from the kernel, classifies them by type, and converts them into DSM messages. These messages are sent to the centralized DSM server, which is implemented as a poll-based event dispatcher. The server tracks per-page metadata in a `struct page_data` table and coordinates coherence actions

by communicating with local clients over UNIX pipes and with remote nodes over TCP sockets or RDMA.

Page invalidation and extraction. In the evaluated prototype, the monitor extracts pages with `process_vm_readv` and invalidates them with `process_madvise()` mediated by the kprobe helper. This avoids repeated parasite injection on the coherence fast path while preserving the same logical page-state transitions described above. We retain a Compel-based fallback for environments that prioritize kernel independence over throughput.

Optional heterogeneous execution. For cross-ISA deployment, Stretch compiles the application into semantically equivalent binaries from shared LLVM IR, then transforms per-thread register state, stack layout, and calling-convention details at migration time using techniques from Popcorn Linux and Dapper [1, 2]. This is an optional extension; the core memory-coherence runtime operates independently of ISA translation.

Overall, the implementation reflects the central thesis of this paper: recent Linux memory-management mechanisms—`userfaultfd` for fault interception, `process_vm_*` for page extraction, and `madvise`-based invalidation—are sufficient to realize a practical page-coherent DSM runtime for existing multithreaded software.

4 Evaluation

We evaluate a prototype of Stretch under both homogeneous and heterogeneous hardware environments.

For *homogeneous cluster setting*, we conduct experiments on CloudLab's Utah cluster (c6525-25g) to demonstrate that Stretch can be deployed on production-grade infrastructure. Each node is equipped with an AMD EPYC 7302P processor (16 cores / 32 threads) with SMT enabled and 120 GiB of DRAM. Networking is provided by Mellanox ConnectX-5 adapters operating in RoCE mode, configured with an MTU of 1024 B and 25 Gbps link bandwidth. All nodes run Ubuntu 24.04 with Linux kernel 6.8.0-71-generic, GCC 13, and our instrumented version of CRIU 4.0 integrated with Stretch.

We evaluated Stretch on this homogeneous machine setting using the Phoenix benchmark suites [6, 29]. Unless otherwise noted, the evaluation uses Stretch's explicit tracked-region mode. Each benchmark supplies the DSM runtime with the ranges of genuinely shared data and with dedicated synchronization pages, and the shared arrays used in the experiments are page-aligned to reduce false sharing. These adjustments do not change the benchmark algorithms, but they avoid conflating DSM cost with unrelated heap metadata or allocator placement. Unless stated otherwise, all experiments also use the lower-overhead page-transfer and invalidation path based on `process_vm_readv()` and `process_madvise()` from Section 3.5.

To evaluate distributed thread execution on *heterogeneous machines* and measure energy efficiency, we use a local x86-64 Xeon server as the primary node (similar to the CloudLab server) and three Raspberry Pi 4 (RPi4) boards as remote nodes. All machines run Ubuntu Server 22.04 with Linux kernel 5.14, a configuration chosen specifically to satisfy the Popcorn Linux compiler's requirement for matching kernel and toolchain versions [27]. Power consumption is measured using a SURAIELEC Energy Watt Meter.

4.1 Microbenchmarks

To quantify the cost of individual DSM coherence operations under the two network transports, we design a three-node, page-level microbenchmark exchanging 512 pages. As shown in Table 1, we measure the latency of `GET_PAGE_INVALIDATE` across three distinct transfer directions, corresponding to the possible ownership configurations in the MSI protocol.

Table 1. Page state transitions across nodes (M for Modified, S for Shared and I for Invalid)

Phase	C1	S	C2
Startup	S	S	S
Set Up	M	I	I
Server–Client 1 (S-C1)	I	M	I
Client 2–Server (C2-S)	I	I	M
Client 1–Server–Client 2 (C1-S-C2)	M	I	I

After an initial setup phase where Client 1 acquires exclusive ownership via `SEND_INVALIDATE`, we evaluate: (1) *Server–Client 1* (S-C1), where the server requests the page from Client 1; (2) *Client 2–Server* (C2-S), where Client 2 requests the page from the server; and (3) *Client 1–Server–Client 2* (C1-S-C2), the three-hop case where the server must forward the request to the current owner and retrieve the page on behalf of the requester. We evaluate three CPU allocation configurations (1 core, 2 cores, and 32 cores) to study the impact of CPU provisioning on coherence latency, comparing TCP and RDMA transports while holding the number of nodes, workload, and shared-memory footprint constant.

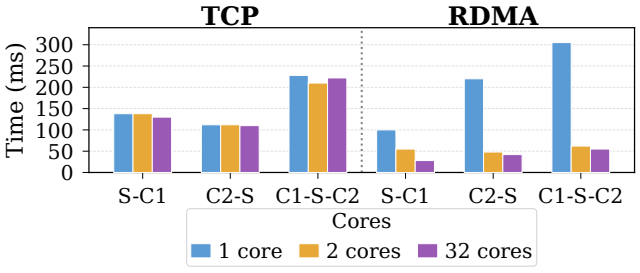

Figure 4. Page test time by transport and direction.

Figure 4 highlights a fundamental asymmetry between the two networking methods. Under TCP, latency remains stable

across all CPU configurations; specifically, the C1-S-C2 case ranges from 212 to 226ms, while S-C1 and C2-S remain steady at approximately 136ms and 106–110ms, respectively. These results confirm that core availability does not significantly impact TCP performance.

Under RDMA, behavior is strongly dependent on CPU provisioning. With a single core, C1-S-C2 reaches 305ms and C2-S reaches 221ms, both worse than TCP in the same configuration. Moving to 2 cores, latency drops sharply to around 50ms across all directions. With 32 cores, performance remains in a similar range, with the exception of the S-C1 direction which improves further. The poor single-core RDMA performance stems from its higher CPU involvement: our implementation relies on busy-polling and completion queue processing, which contend with application and fault-handler threads when only one core is available. TCP instead benefits from kernel-level scheduling and interrupt-driven progress, yielding more predictable behavior under core pressure. Once additional cores are available, RDMA can dedicate execution resources to communication progress, exposing its lower-latency data path and outperforming TCP.

4.2 Phoenix Benchmark Suite

Figure 5 presents the execution time of the Phoenix benchmark suite [6, 29] under Stretch using TCP compared to vanilla multithreaded execution, i.e. the unmodified Phoenix binary on a single machine. Each plot reports total execution time broken down into three components: (i) the normal execution phase, (ii) the CRIU-based process checkpoint phase (dump), and (iii) Stretch's process snapshot rewriting and thread restore phase (restore). We evaluate several distribution configurations for each benchmark by restoring different subsets of threads across multiple remote nodes (e.g., 4+4, 32+32, or 2+2+2+2), while keeping the total number of active cores constant. These results should be read as a regime characterization rather than a claim of universal scale-out: the suite intentionally includes both workloads that amortize DSM faults well and workloads that do not.

Overall, the results show that Stretch is able to scale out multithreaded applications across multiple machines with modest overhead. Across all benchmarks, the dump and restore overheads remain small relative to total execution time, typically less than 5–10%, and become negligible as the number of threads increases. However, the benchmarks exhibit qualitatively different scaling behavior depending on their memory access patterns and input sizes. We classify them into two groups and discuss each below.

Scalable benchmarks. Matrix Multiply, PCA, and KMeans admit arbitrarily large inputs, thereby stressing the DSM runtime at scale.

Matrix Multiply (6000×6000) is the clearest success case for Stretch. For the experiment, we used a 6,000-by-6,000 matrix. Each fetched page of matrix data sustains $O(n)$ floating-point operations before the next page fault, yielding a high computation-to-page-fault ratio. Stretch achieves up to 3.39× speedup at 4 cores distributed across 4 nodes, and multi-node configurations (e.g., 16+16+16+16) consistently outperform single-node execution by distributing both computation and memory bandwidth.

PCA computes an asymmetric covariance matrix where earlier threads handle proportionally more work. A naive equal distribution of threads across nodes leaves some nodes idle while others are overloaded. With a work-balanced configuration (9+10+21+34 threads), Stretch achieves 2.08× speedup, compared to 1.38× with an equal split. This result highlights that thread placement strategy, not just thread count, is an important factor for DSM performance. However, PCA incurs a significant deployment overhead, particularly in the full-core configuration, which limits speedup gains at high core counts.

KMeans is the most challenging benchmark and a clear negative result. Centroid updates touch shared pages in tight loops, generating write-invalidation storms that dominate execution time. Stretch achieves only 0.17–0.22× of vanilla performance across all configurations and core counts. This result is expected: iterative clustering with fine-grained global updates is fundamentally incompatible with page-granularity coherence, regardless of transport or CPU provisioning.

Input-limited benchmarks. String Match, Word Count, Histogram, and Linear Regression exhaust their maximum feasible input sizes at moderate parallelism, limiting the opportunity for DSM-based scaling.

String Match is a read-only scan that generates no write-sharing. Stretch achieves 3.22× speedup at 1 core (2 nodes), declining to 1.22× at 5 cores as the vanilla baseline saturates. This benchmark validates correct thread distribution but provides no insight into DSM coherence costs. *Word Count* allocates per-bucket counters across shared pages, exhibiting moderate page reuse. Speedups range from 4.06× at 1 core to 0.90× at 8 cores, where vanilla execution already approaches hardware limits and DSM overhead becomes non-amortizable. *Histogram* updates a small, fixed-size output array (two pages per thread). With few shared pages, DSM overhead is modest. Speedups range from 3.45× at 1 core to 1.23× at 8 cores. *Linear Regression* is purely compute-bound with limited input scaling; speedups range from 3.16× at 1 core to 1.46× at 5 cores.

RDMA versus TCP across benchmarks. The Phoenix results reinforce the microbenchmark finding from Figure 4: RDMA's advantage depends on server CPU availability. At low core counts, where the DSM server is CPU-bound on completion-queue polling and coherence bookkeeping, RDMA can underperform TCP across multiple benchmarks. As more cores become available, RDMA's ability to bypass the kernel networking stack yields lower coherence latency and faster page transfers. This observation has practical deployment implications: RDMA-accelerated DSM should be

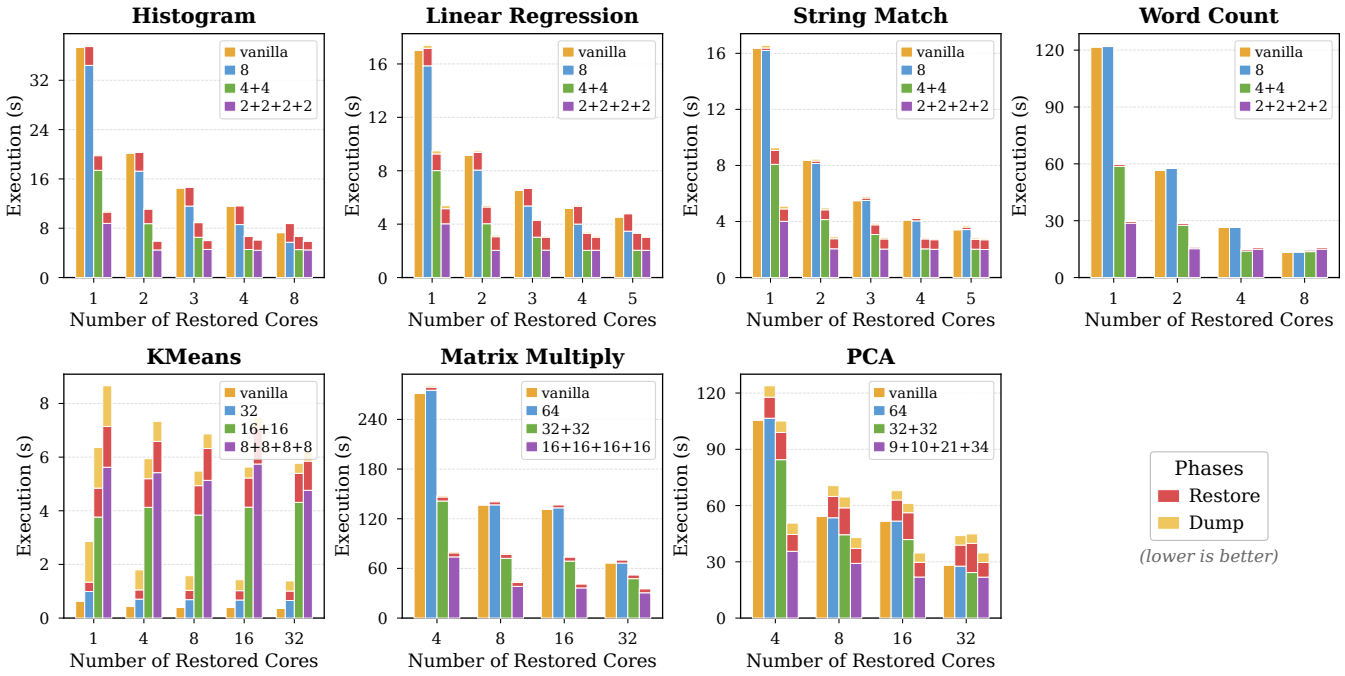

Figure 5. Execution time of the Phoenix benchmark suite.

Table 2. Summary of Phoenix benchmark results under STRETCH.

Benchmark	Best	Config	Dominant Pattern
Matrix Mult.	3.39×	4c, 4 nodes	Compute-intensive
Word Count	4.06×	1c, 4 nodes	Moderate sharing
Histogram	3.45×	1c, 4 nodes	Few shared pages
String Match	3.22×	1c, 4 nodes	Read-only
Lin. Regr.	3.16×	1c, 4 nodes	Compute-bound
PCA	2.08×	4c, balanced	Asymmetric compute
KMeans	0.22×	4c, 4 nodes	Write-storm

provisioned with dedicated server-side CPU resources to realize its latency advantage.

Table 2 summarizes the best speedup achieved by each benchmark, the corresponding configuration, and the dominant memory access pattern. The results confirm that STRETCH is most effective for workloads with high computation per page access and coarse-grained sharing, while fine-grained write-sharing remains a fundamental challenge for page-based DSM.

4.3 Secondary Use Case: Heterogeneous Deployment (x86 + ARM)

As an additional use case, we evaluate STRETCH on heterogeneous x86–ARM hardware to demonstrate that the DSM runtime operates correctly across ISAs. We run three multithreaded applications (`Blackscholes`, `grep`, `matrix` multiplication) from the Popcorn compiler suite [2, 27], each with one main thread and three workers, distributing workers to Raspberry Pi 4 nodes.

Figure 6 (left) shows that scaled-out execution consistently outperforms the single-core local baseline, with execution time decreasing as more remote nodes are added. Figure 6 (right) reports energy efficiency: `matrix` multiplication improves by 22% and `grep` by 14% in jobs per kilojoule when offloading threads to low-power ARM nodes. These results confirm that STRETCH's coherence protocol operates correctly across ISA boundaries. The memory-management behavior under heterogeneous deployment is identical to the homogeneous case; the additional cost is limited to the one-time thread-context transformation at migration time.

5 Discussion

The evaluation in Section 4 shows that STRETCH can sustain distributed execution for a range of multithreaded workloads, but with significant performance variation across benchmarks. In this section we analyze the factors that determine when page-based DSM is effective, discuss the tradeoffs inherent in our design, and identify current limitations.

5.1 Computation-to-Page-Fault Ratio

The single most important predictor of STRETCH's performance is the ratio of useful computation performed between successive page faults to the cost of servicing each fault. A

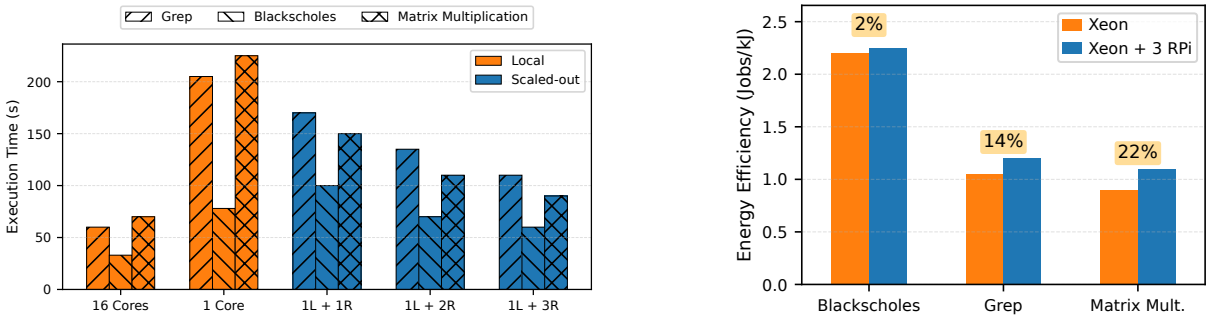

Figure 6. Heterogeneous x86–ARM deployment. Left: execution time under local (L) and remote (R) configurations, each with 1 CPU core. Right: energy efficiency (jobs/kJ) of local vs. scaled-out execution.

page fault in Stretch involves a kernel-to-userspace notification via `userfaultfd`, a protocol round-trip to the centralized server, a page transfer over the network, and installation of the page into the faulting process's address space. When this cost is amortized over a large volume of computation per fetched page, the DSM overhead becomes negligible and distributed execution achieves near-linear speedups.

Our Phoenix results (Figure 5, Table 2) illustrate this directly. Matrix Multiply achieves up to 3.39× speedup because each fetched page of matrix data sustains $O(n)$ floating-point operations before the next fault—this benefit is *scalable*, persisting as more cores and nodes are added. At the other extreme, KMeans achieves only 0.17–0.22× of vanilla performance because centroid updates touch shared pages in tight loops, generating invalidation storms that dominate execution time. Between these extremes, benchmarks like PCA (2.08×) and Word Count (4.06× at 1 core) occupy a middle ground. Word Count achieves the highest raw speedup number in our suite, but this reflects the weak single-core vanilla baseline rather than scalable DSM performance: its speedup declines to 0.90× at 8 cores as the vanilla baseline saturates the available parallelism.

This metric provides a practical guideline: workloads are good candidates for Stretch when their per-page computation substantially exceeds the round-trip cost of a coherence operation. For the CloudLab hardware used in our evaluation, this cost is on the order of 30–60 ms with RDMA and 130–225 ms with TCP (Figure 4). Applications whose inner loops perform hundreds of thousands of operations per page access, such as dense linear algebra or Monte Carlo simulations, are natural fits. Applications with tight update loops over shared data structures, such as iterative clustering, are not.

5.2 Page Granularity and False Sharing

Stretch operates at page granularity (4 KB) because page faults and page-level protection are the control points exposed by Linux's memory-management interfaces. This choice is fundamental to deployability: it requires no compiler support and no new programming model. However, it also means that coherence operates at a coarser grain than the application's actual sharing patterns.

This page-granularity design creates a transparency-versus-control tradeoff. Stretch can automatically register anonymous regions after restore, but fully automatic tracking includes allocator metadata and thread-private pages that generate faults unrelated to intended sharing. For that reason, the evaluation uses explicit region registration and page-aligned shared allocations. These are modest DSM-aware adjustments, but they mean that the strongest performance results in Section 4 should be read as applying to applications whose shared working set can be identified, not to arbitrary binaries with zero setup.

The primary consequence is false sharing. When unrelated variables reside on the same physical page, a write to one variable invalidates the entire page on all other nodes, even if no other node accesses that specific variable. This effect is well-known in cache-coherence literature [4, 12] and is amplified in a software DSM where invalidation costs are orders of magnitude higher than in hardware.

In hardware cache coherence, false sharing at the 64-byte cache-line level adds nanoseconds of overhead per spurious invalidation—a cost that is usually tolerable. In Stretch, the same phenomenon at 4 KB page granularity incurs microseconds per invalidation, making it a first-order performance concern rather than a secondary effect. KMeans illustrates this directly: centroid arrays are small enough that multiple centroids share the same page, and each centroid update by any node invalidates the entire page on all other nodes, producing a cascade of coherence traffic disproportionate to the actual data modified.

In practice, Stretch offers two mitigations. First, shared data structures can be allocated via `mmap` with explicit page alignment, ensuring that distinct logical structures do not share physical pages. Second, Stretch supports explicit page selection, where only memory regions that are genuinely shared across threads are registered with the DSM runtime,

avoiding coherence traffic for thread-private data that happens to reside in anonymous pages. Both techniques can significantly reduce spurious invalidation traffic.

More broadly, these observations highlight that for page-based DSM, *memory layout* is as important as algorithmic structure in determining performance. The choice of allocator, the alignment of shared data structures, and the separation of read-shared from write-shared regions all have direct impact on coherence cost—considerations that are familiar in hardware cache optimization but take on new significance at page granularity, where the penalty for poor layout is three orders of magnitude larger.

5.3 Centralized Server: Scalability and Alternatives

A recurring design question is the scalability of Stretch's centralized DSM server. By design, all ownership transitions, invalidation decisions, and page transfers are serialized through a single server process. This architecture keeps the coherence protocol simple and correct: there is exactly one authority for each page's state, and conflicting requests are resolved in a well-defined order. For small-to-medium clusters (2–8 nodes), this design is sufficient, as demonstrated by our evaluation.

However, the centralized server also represents a potential bottleneck. Figure 4 shows that server CPU provisioning directly affects coherence latency: when the server is constrained to a single core, RDMA performance degrades by 4-5× because completion-queue polling and coherence bookkeeping contend for the same core. Write-intensive workloads amplify this effect, as each write fault generates invalidation messages proportional to the number of nodes.

Several alternative designs could address this limitation. Self-invalidation protocols, as used in ArgoDSM [11], eliminate the need for explicit invalidation messages by having each node invalidate its own stale copies at synchronization points. This approach removes the server from the invalidation path entirely, but requires the application to be data-race-free, a stronger assumption than Stretch requires. Distributed directory protocols partition page ownership across multiple nodes, avoiding a single bottleneck but introducing complexity in resolving cross-partition conflicts. Hardware-assisted coherence via CXL [38] offers a complementary path, where interconnect-level support can reduce the software overhead of page-state management.

Stretch's centralized design reflects a deliberate trade-off: simplicity and correctness in exchange for scalability limits. For the target deployment scenario—extending a multithreaded application beyond a single machine using only userspace mechanisms on commodity hardware—this trade-off is appropriate. The evaluation shows that with adequate CPU provisioning (2+ cores) and RDMA transport, the server sustains coherence for the workloads and cluster sizes we consider.

5.4 Fault Tolerance

Our current prototype does not include fault-tolerance mechanisms. If a node fails during distributed execution, the entire computation must be restarted. However, CRIU checkpoints provide a natural recovery substrate: because Stretch already captures full process state at the beginning of distributed execution, periodic re-checkpointing during execution could enable rollback recovery with bounded lost work. Integrating such a mechanism would require extending the DSM protocol with epoch-based tracking to identify which pages have been modified since the last checkpoint, ensuring that recovered state is consistent across nodes. Full fault tolerance, including replication, logging, or speculative execution, is orthogonal to the current coherence design and is left as future work.

5.5 Limitations and Future Directions

Beyond the scalability and fault-tolerance considerations discussed above, several limitations merit explicit acknowledgment. First, write-intensive workloads with fine-grained sharing remain fundamentally challenging for any page-based DSM, and Stretch is no exception. The KMeans results illustrate this clearly: when the working set is repeatedly written across all nodes, coherence overhead dominates regardless of transport or CPU provisioning. Second, the optimized invalidation path in our evaluated prototype relies on `process_madvise()` with a lightweight helper, while the fully user-space Compel fallback is slower; narrowing that gap is an important deployment issue. Third, cross-ISA execution overhead is not fully characterized in our evaluation; the compilation pipeline (shared LLVM IR, per-architecture code generation) and per-thread context transformation add latency that we have not isolated from DSM costs. Fourth, Stretch currently performs static thread placement at the beginning of execution; dynamic re-placement in response to load imbalance or changing access patterns could improve performance but would require extending the checkpoint/restore mechanism to operate on running threads.

Looking forward, we identify several promising directions. Integrating self-invalidation techniques from ArgoDSM's protocol design could reduce server load for data-race-free workloads while retaining Stretch's centralized fallback for general programs. Page prefetching based on access-pattern prediction could hide fault latency for applications with regular memory access patterns. Finally, the emergence of CXL-based memory pooling offers an opportunity to combine Stretch's software coherence with hardware-assisted remote memory access, potentially narrowing the gap between local and remote page-access costs.

6 Related Work

Classic and modern DSM systems. Distributed Shared Memory (DSM) systems aim to provide a unified memory abstraction across physically separate machines [19].

Classic DSM systems like IVY [18], TreadMarks [12], and Munin [4] introduced coherence protocols and consistency models for shared-memory programs, but typically required kernel modifications or OS-level hooks, limiting deployability on commodity platforms. More recent systems leverage high-speed networks or hardware support: FaRM [7] and DrTM [5] use one-sided RDMA for low-latency key-value access, Grappa [26] provides a partitioned global address space over RDMA, DEX [17] enables distributed multithreaded execution with kernel-level DSM but is tied to a modified Linux kernel and a single ISA, and CXL-SHM [38] explores hardware-coherent shared memory over CXL interconnects. DEX is the closest prior system in overall goal. Relative to DEX, Stretch moves the coherence control path out of a modified kernel, targets a commodity-Linux deployment with explicit tracked regions, and treats selective thread placement and heterogeneity as supporting capabilities rather than the main contribution.

ArgoDSM [11] takes a fundamentally different approach to coherence by using self-invalidation and passive data-classification directories. Rather than having a central server send explicit invalidation messages, each node invalidates its own stale copies at synchronization points. This eliminates message-handler latency and enables local coherence decisions, but requires applications to be data-race-free (DRF)—a stronger assumption than Stretch requires. Where ArgoDSM trades generality for scalability, Stretch's centralized protocol supports arbitrary sharing patterns at the cost of server-mediated transitions. These two designs represent complementary points in the DSM design space.

DRust [22] achieves object-level coherence for Rust programs by leveraging the language's ownership semantics to track sharing at fine granularity, avoiding page-level false sharing. However, DRust requires applications to be written in Rust and cannot support existing C/C++ POSIX-threaded programs directly. Stretch trades coherence granularity for broader applicability: it targets conventional POSIX-threaded applications, although our best-performing configurations use explicit region registration.

Frameworks like MPI [8] and Ray [24] support scaling out across machines but follow message-passing or actor models that require applications to be written explicitly for distributed execution. In contrast, Stretch preserves the shared-memory multithreading abstraction, allowing existing POSIX-threaded programs to scale out with modest DSM-aware setup. Commercial systems such as TidalScale and ScaleMP [9] aggregate multiple physical machines into a single virtual SMP using hypervisor-level page-fault interception and transparent memory migration. While these approaches provide full transparency to the guest operating system, they require specialized hypervisor software and dedicated deployment infrastructure. Stretch pursues a similar goal but with a lighter-weight commodity-Linux design centered on user-space fault handling and explicit tracked regions, rather than a full virtualization layer.

The `userfaultfd` interface has been used for postcopy live migration, demand paging, and fast snapshotting within single-node environments. Stretch extends `userfaultfd` beyond single-node memory virtualization to drive a full cross-node coherence protocol—a qualitatively different use of the same kernel interface. CRIU has been widely used for container migration and fault tolerance [3, 25, 35, 39]; Dapper extends it to cross-ISA whole-process migration [1]. Stretch extends CRIU to thread-level selective migration and combines it with a userspace DSM, whereas prior CRIU-based systems do not provide distributed memory coherence.

Existing systems such as Popcorn Linux [2], Stramash [37], HeterSec [34], and UNIFICO [23] support execution across heterogeneous ISAs but typically require kernel modifications or specialized OS support. Stretch optionally reuses cross-ISA techniques from these systems for thread-context transformation, but its core contribution - the userspace coherence runtime - is independent of ISA translation.

7 Conclusion

This paper presented Stretch, a fault-driven distributed shared memory runtime for multithreaded applications on commodity Linux. Stretch combines CRIU-based thread placement, centralized MSI-style page coherence driven by `userfaultfd`, and a fault-driven synchronization mechanism for the barriers and coarse-grained mutexes used by our workloads.

The evaluation shows that the computation-to-page-fault ratio is the key determinant of performance: Matrix Multiply reaches up to 3.39× speedup and PCA 2.08×, while KMeans exposes the fundamental cost of fine-grained write sharing at page granularity. RDMA lowers coherence latency by up to 4× relative to TCP when the centralized server has enough CPU resources. Overall, these results suggest that modern Linux memory-management interfaces are sufficient to support a practical DSM runtime for small-to-medium clusters when the shared working set can be identified and false sharing is controlled. Future work includes distributed coherence metadata, better fully user-space invalidation, and page prefetching for regular access patterns.

Acknowledgments

We thank the anonymous reviewers for their insightful comments, which have greatly improved the paper. This work is partly supported by the US Office of Naval Research under grants N00014-18-1-2022, N00014-19-1-2493, and N00014-22-1-2672, and by the US National Science Foundation (NSF) under grant CNS 2127491. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

References

[1] Abhishek Bapat, Jaidev Shastri, Xiaoguang Wang, Abilesh Sundarasamy, and Binoy Ravindran. 2024. Dapper: A Lightweight and Extensible Framework for Live Program State Rewriting. In *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*. 738–749. doi:10.1109/ICDCS60910.2024.00074

[2] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. 2017. Breaking the boundaries in heterogeneous-ISA datacenters. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 645–659.

[3] Christopher Blackburn, Xiaoguang Wang, and Binoy Ravindran. 2022. Rave: A Modular and Extensible Framework for Program State Re-Randomization. In *Proceedings of the 9th ACM Workshop on Moving Target Defense (MTD'22)*. Association for Computing Machinery, New York, NY, USA, 3–10. doi:10.1145/3560828.3564008

[4] John B. Carter, John K. Bennett, and Willy Zwaenepoel. 1991. Implementation and performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (Pacific Grove, California, USA) *(SOSP '91)*. Association for Computing Machinery, New York, NY, USA, 152–164. doi:10.1145/121132.121159

[5] Haibo Chen, Rong Chen, Xingda Wei, Jiaxin Shi, Yanzhe Chen, Zhaoguo Wang, Binyu Zang, and Haibing Guan. 2017. Fast in-memory transaction processing using RDMA and HTM. *ACM Transactions on Computer Systems (TOCS)* 35, 1 (2017), 1–37.

[6] Christos Kozyrakis. 2018. phoenix. https://github.com/kozyraki/phoenix, Online.

[7] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. {FaRM}: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 401–414.

[8] Message P Forum. 1994. MPI: A message-passing interface standard.

[9] Future Grid. 2026. ScaleMP vSMP. https://futuregrid.github.io/manual/scalemp.html, Online.

[10] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European parallel virtual machine/message passing interface users' group meeting*. Springer, 97–104.

[11] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. 2015. 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)*. Association for Computing Machinery, 3–14. doi:10.1145/2749246.2749250

[12] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. 1994. TreadMarks: distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference* (San Francisco, California) *(WTEC'94)*. USENIX Association, USA, 10.

[13] kernel.org. 2023. *Userfaultfd*. https://docs.kernel.org/admin-guide/mm/userfaultfd.html.

[14] Michael Kerrisk. 2024. *madvise(2) — Linux manual page*. https://man7.org/linux/man-pages/man2/madvise.2.html.

[15] Michael Kerrisk. 2025. process_madvise - give advice about use of memory to a process . *https://man7.org/linux/man-pages/man2/process_madvise.2.html* (2025).

[16] Michael Kerrisk. 2025. process_vm_readv(2) — Linux manual page. *https://man7.org/linux/man-pages/man2/process_vm_readv.2.html* (2025).

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

[18] Kai Li. 1988. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the International Conference on Parallel Processing, ICPP '88, The Pennsylvania State University, University Park, PA, USA, August 1988. Volume 2: Software*. Pennsylvania State University Press, 94–101.

[19] Kai Li and Paul Hudak. 1989. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.* 7, 4 (Nov. 1989), 321–359. doi:10.1145/75104.75105

[20] Liran Liss, Yitzhak Birk, and Assaf Schuster. 2005. In-kernel integration of operating system and infiniband functions for high performance computing clusters: a DSM example. *IEEE Transactions on Parallel and Distributed Systems* 16, 9 (2005), 830–840.

[21] Google LLC. 2023. *Protocol Buffers - Google's data interchange format*. https://github.com/protocolbuffers/protobuf.

[22] Haoran Ma, Yifan Qiao, Shi Liu, Shan Yu, Yuanjiang Ni, Qingda Lu, Jiesheng Wu, Yiying Zhang, Miryung Kim, and Harry Xu. 2024. 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 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 97–115. https://www.usenix.org/conference/osdi24/presentation/ma-haoran

[23] Nikolaos Mavrogeorgis, Christos Vasiladiotis, Pei Mu, Amir Khordadi, Björn Franke, and Antonio Barbalace. 2024. UNIFICO: Thread Migration in Heterogeneous-ISA CPUs without State Transformation. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*. 86–99.

[24] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 561–577.

[25] Shripad Nadgowda, Sahil Suneja, Nilton Bila, and Canturk Isci. 2017. Voyager: Complete container state migration. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2137–2142.

[26] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. {Latency-Tolerant} software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 291–305.

[27] Rob Lyerly. 2018. Popcorn Linux compiler toolchain for heterogeneous-ISA execution. https://github.com/ssrg-vt/popcorn-compiler, Online.

[28] Paul Sweazey and Alan Jay Smith. 1986. A class of compatible cache consistency protocols and their support by the IEEE futurebus. *ACM SIGARCH Computer Architecture News* 14, 2 (1986), 414–423.

[29] Justin Talbot, Richard M Yoo, and Christos Kozyrakis. 2011. Phoenix++ modular mapreduce for shared-memory systems. In *Proceedings of the second international workshop on MapReduce and its applications*. 9–16.

[30] criu.org. 2025. Checkpoint/Restore In Userspace CRIU. https://criu.org/Main_Page

[31] criu.org. 2025. Compel. https://criu.org/Compel

[32] criu.org. 2025. CRIT. https://criu.org/CRIT

[33] criu.org. 2025. CRIU Images. https://criu.org/Images

[34] Xiaoguang Wang, SengMing Yeoh, Robert Lyerly, Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. 2020. A Framework for Software Diversification with ISA Heterogeneity. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 427–442. https://www.usenix.org/conference/raid2020/presentation/wang-xiaoguang

[35] Adityas Widjajarto, Deden Witarsyah Jacob, and Muharman Lubis. 2021. Live migration using checkpoint and restore in userspace (CRIU): Usage analysis of network, memory and CPU. *Bulletin of Electrical Engineering and Informatics* 10, 2 (April 2021), 837–847. doi:10.11591/

eei.v10i2.2742 Number: 2.

[36] Bruce Wile. 2014. Coherent accelerator processor interface (CAPI) for POWER8 systems. *IBM White Paper* (2014).

[37] Tong Xing, Cong Xiong, Tianrui Wei, April Sanchez, Binoy Ravindran, Jonathan Balkind, and Antonio Barbalace. 2025. Stramash: A Fused-Kernel Operating System For Cache-Coherent, Heterogeneous-ISA Platforms. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1172–1188.

[38] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. 2023. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 658–674.

[39] Diyu Zhou and Yuval Tamir. 2020. Fault-tolerant containers using nilicon. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1082–1091.

Received 2026-04-03; accepted 2026-05-04