CRIU-RTX: Remote Thread eXecution using Checkpoint/Restore in Userspace

Mohamed Husain Noor Mohamed

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Binoy Ravindran, Chair
Kendall Giles
Xiaoguang Wang

June 20, 2023
Blacksburg, Virginia

Keywords: Heterogeneous Systems, Distributed Execution, Energy Efficiency

CRIU-RTX: Remote Thread eXecution using Checkpoint/Restore in Userspace

Mohamed Husain Noor Mohamed

(ABSTRACT)

Scaling up application performance on single high-end machines is increasingly becoming difficult due to scalability challenges of processor interconnects, cache coherence protocols, and memory bandwidth. Significant prior work has addressed this problem by scaling-out application threads across multiple nodes to exploit resources outside the single machine boundary. Prior works have also leveraged heterogeneous instruction set architecture (ISA) systems to improve application performance as well as energy-efficiency, a major cost driver in datacenters, by augmenting high-end servers with power-efficient embedded boards. Existing works, however, suffer from deployability challenges due to dependencies on the operating system or programming models that require non-trivial application modifications. We introduce CRIU-RTX, a userspace framework to scale-out multi-threaded applications across multiple nodes. Integrated with HetMigrate, a prior work on migrating processes across heterogeneous-ISA systems, CRIU-RTX can suspend a subset of threads in a process and resume their execution on different nodes, including, but not limited to heterogeneous-ISA nodes. CRIU-RTX implements distributed shared memory in userspace, thereby allowing application threads to access distributed memory transparently without any operating system dependency. Our experimental evaluations show 21% to 43% performance gains while scaling-out applications across x86-64 servers, and energy efficiency gains of up to 18% while scaling-out across a cluster of x86-64 server and ARM64 embedded boards. Since CRIU-RTX does not depend on operating system modifications, it can be easily deployed on a

diverse set of machines, including, but not limited to ISA-different machines running the stock Linux operating system.

CRIU-RTX: Remote Thread eXecution using Checkpoint/Restore in Userspace

Mohamed Husain Noor Mohamed

(GENERAL AUDIENCE ABSTRACT)

Commonly referred to as "Moore's Law", Gordan Moore predicted that the number of transistors on a chip would double every two years. However, this law no longer holds true, leading to a shift in computer research and development. To meet the increasing demands for faster and cheaper servers, researchers began exploring alternative computer designs. Data centers have started adopting servers with diverse architectures to enhance the cost-to-performance ratio, resulting in heterogeneous environments. Distributed execution refers to the process of running computational tasks or executing software across multiple interconnected systems or nodes. Instead of relying on a single machine or processor, the workload is distributed among a network of computers, allowing for parallel processing and improved performance. Prior works in this direction had difficulty in adoption due to customized hardware or operating system requirements. This thesis introduces CRIU-RTX, a userspace framework to scale-out application threads without operating system dependency. We implemented a distributed shared memory system in userspace to allow application threads running in scaled-out execution to access distributed memory as if they are running on the same machine. Our evaluations of CRIU-RTX show significant improvement in performance and energy-efficiency.

Dedication

This thesis is dedicated to my parents

Acknowledgments

First, I would like to thank my advisor, Dr. Binoy Ravindran, for his immense support throughout my master's program. His course Compiler Optimizations holds a special place in my heart. I am indebted to Dr. Ravindran for his continuous encouragement and mentorship. I am thankful for the opportunity to be a part of SSRG under his guidance, and working alongside my colleagues at SSRG has been a tremendous learning experience.

I would also like to thank to Dr. Xiaoguang Wang, who has been a great mentor and imparted invaluable knowledge to me. I am deeply grateful for his willingness to dedicate time and effort to address my queries and concerns.

I would like to thank Dr. Kendall Giles for accepting to be part of my thesis committee and dedicating his valuable time.

I am grateful to my brother Ansari for his guidance and inspiration throughout all stages of my life. I am also grateful to my friend Hemanth, whose support has meant a great deal to me. Finally, I want to thank my parents and siblings for their constant love and for being there for me.

This work is supported in part by the US Office of Naval Research (ONR) under grants N00014-18-1-2022, N00014-19-1-2493, and N00014-22-1-2672 and US National Science Foundation (NSF) under grant CNS 2127491. Any opinions, findings, conclusions, or recommendations expressed in this thesis are those of the author and do not necessarily reflect the views of ONR or NSF.

Contents

List of Figures

Chapter 1

Introduction

Based on a prediction by the International Data Corporation (IDC) [57], it is anticipated that the worldwide production of data will surpass 180 zettabytes by 2025. With large volumes of data, more computational power is required to process them in a reasonable amount of time. For decades, computing power has consistently adhered to Moore's Law, which states that the processing power of computers approximately doubles every two years. As Moore's law started to slow down, single-threaded CPU performance started to reach its limitations. Hence chip vendors shifted to multi-core and specialized-core architectures to exploit parallelism in order to meet the ever-increasing performance demand.

The significance of in-memory databases and in-memory graph processing applications is on the rise because of the drastic increase in data collected by organizations. These applications run on high-end machines with extreme processing power and memory capacity to achieve high performance. However, such high-end machines are becoming increasingly difficult to build due to complexities in cache coherence protocols, scalable interconnects, and memory bandwidth, among other factors [53].

Since the last decade, efficient energy usage has become a significant topic globally and had a great impact on computing. Energy-efficiency concerns are now ubiquitous in all types of computing, from mobile computing to server-based computing. Heterogeneous designs provide accelerated performance at significantly high energy-efficiencies by coupling general-purpose computers with special-purpose compute units such as GPUs and FPGAs. Processor

designers started to include high energy-efficiency cores along with high-performance cores to run applications with optimal performance and energy. ARM's big.LITTLE [1] is an example of such heterogeneous architecture in which different cores with different performance and energy profiles of the same instruction set architecture (ISA) co-exist. The Linux kernel community introduced frameworks [39] to exploit the heterogeneity in cores by the process schedulers.

1.1 Motivation

The performance of multi-threaded applications can be improved by scaling out application threads across distributed memory nodes, i.e., a high-end server cluster, which allows applications to exploit remote compute cycles. Also, by scaling-out application threads across server nodes and power-efficient embedded boards, energy-efficiencies can be improved. Such scale-out requires the distributed shared memory (DSM) abstraction. DSM is a memory abstraction for distributed memory systems that provides the illusion of a "shared" virtual address space [40]. DSM ensures memory consistency across concurrent readers and writers using a consistency algorithm. The abstraction thus allows shared memory multi-threaded applications to execute on distributed memory nodes and read/write a coherent shared virtual address space. DSM can be implemented in hardware (e.g., in custom chips [31, 47], switches [51, 55]) or in software (e.g., in OS [29, 33], library [44], middleware [41]). While hardware DSM achieves higher performance, software DSM allows greater flexibility (e.g., for implementing different consistency algorithms). This thesis focuses on the software DSM.

An interesting prior work, Dex [29] shows that compute-intensive multi-threaded applications can be scaled-out across multiple nodes with a Linux kernel extension. Previous works in achieving scaled-out execution require a distributed programming model, which means that

applications need to be rewritten for these environments. Dex overcomes this challenge by implementing DSM in the Linux kernel, which allows threads to access distributed memory using normal load and store instructions. Dex's major limitation is OS dependency: to use Dex's DSM, its custom Linux kernel must be used.

Another closely related work is HetMigrate [38], a modification of Linux's CRIU [3] framework, which allows a process to migrate across heterogeneous instruction set architecture (ISA) nodes without any operating system dependencies. HetMigrate's evaluation showed up to a 39% gain in energy efficiency by offloading compute-intensive workloads to inexpensive embedded boards. HetMigrate can only migrate an entire process and does not support a subset of threads to be migrated and scaled out across nodes. This means with the HetMigrate, a process can be offloaded across one node at a time and does not allow shared execution across multiple nodes.

This thesis raises the following questions on scaling-out application threads to improve performance and energy-efficiency.

1. Can we improve application performance and energy efficiency by scaling-out threads of a process across ISAs without operating system dependency?

2. Can we implement DSM to allow threads to access shared memory without operating system dependency?

1.2 Thesis Contributions

This thesis introduces CRIU-RTX, a userspace framework based on the HetMigrate version of CRIU to transparently scale-out application threads across heterogeneous ISA machines.

1. **Framework to Scale-out Application Threads across ISAs**: CRIU-RTX implements a mechanism to suspend a subset of threads in a process and resume their execution on different nodes with the same or different ISA. CRIU-RTX uses HetMigrate to transform the CRIU images from one ISA to another.

2. **Userspace Distributed Shared Memory**: CRIU-RTX allows threads across nodes to access distributed memory without rewriting the applications. CRIU-RTX uses the Linux kernel feature, userfaulfd, which allows applications to handle page faults in userspace. An MSI-based coherency protocol is implemented at page-level granularity to allow page sharing among threads.

3. **CRIU-RTX's Evaluation**: We evaluated the performance and energy efficiency gains of CRIU-RTX on a server/embedded cluster. Our evaluation shows 21% to 43% performance gains while scaling-out applications across x86-64 servers, and energy efficiency gains of up to 18% on an x86-64 server interconnected with ARM64 embedded boards.

1.3 Thesis Organization

The rest of the thesis is organized as follows.

Chapter 2 provides the background information necessary to understand the contributions of the thesis.

Chapter 3 discusses the related works on process migration and distributed shared memory.

The design and implementation of CRIU-RTX are discussed in Chapter 4 and Chapter 5, respectively.

The evaluation of CRIU-RTX is discussed in Chapter 6.

Chapter 7 concludes the thesis and discusses the limitations of CRIU-RTX, and identifies future work.

Chapter 2

Background

CRIU-RTX is a userspace framework that allows scaling-out application threads across homogenous-ISA and heterogeneous-ISA nodes transparently.

The concepts necessary to understand CRIU-RTX are Linux's CRIU mechanism (Section 2.1), the Compel parasite tool (Section 2.2), Distributed Shared Memory (Section 2.3), Linux's Userfaultfd mechanism (Section 2.4), the Popcorn Linux software stack (Section 2.5), and the HetMigrate software system (Section 2.6).

2.1 Checkpoint and Restore in Userspace

CRIU [3] is an open-source checkpoint/restore tool for the Linux operating system, widely used for migrating userspace applications across different host machines. CRIU provides transparent snapshotting of Linux applications by suspending the process and saving the process state like memory, file descriptors, and other metadata held by the operating system as protobuf images [8] in a filesystem. It can be used to migrate applications to different nodes by transferring the protobuf images to different nodes.

While check-pointing the program, CRIU gets the process metadata via the /proc [7] entries in the system. Then CRIU uses the ptrace system call to read the process's address space and dump them into the disk. During restore, CRIU RESTORE morphs itself as the target

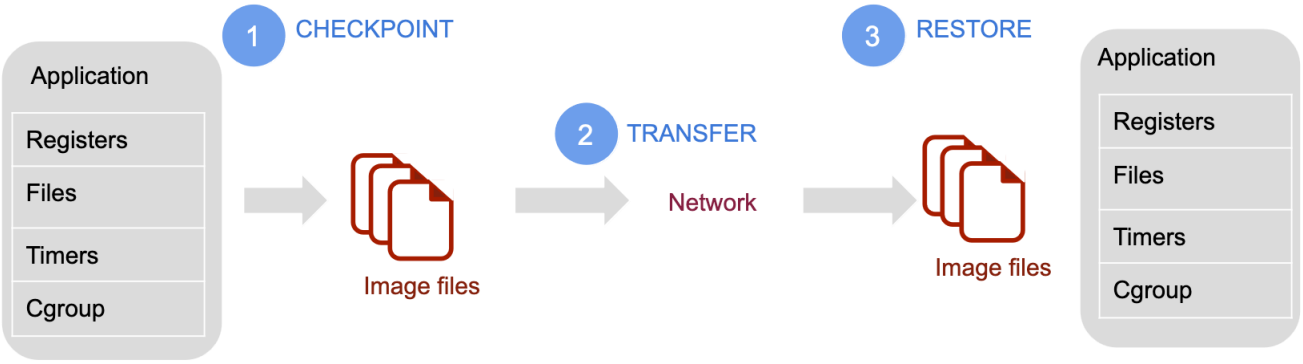

Figure 2.1: CRIU's workflow.

program and restores the pages, and file mappings, and resumes the program from the exact program point where it was suspended. Figure 2.1 shows the overview of CRIU's workflow.

One of the image files dumped by CRIU is the core image file, which contains the values of the registers, including the instruction pointer of the suspended process. For each thread running in the process, CRIU dumps separate core images for those threads, and each core image contains registers, sigmask, and timers data of the respective thread.

CRIU also provides a tool called CRIT (CRiu Image Tool), which allows editing the CRIU images before restoring them as a process. By editing the image files, the state of the program can be modified. For example, the pstree.img file holds the process and thread IDs of the suspended process. Using the CRIT tool, one could change the PID of the program from the one originally allocated at the origin node.

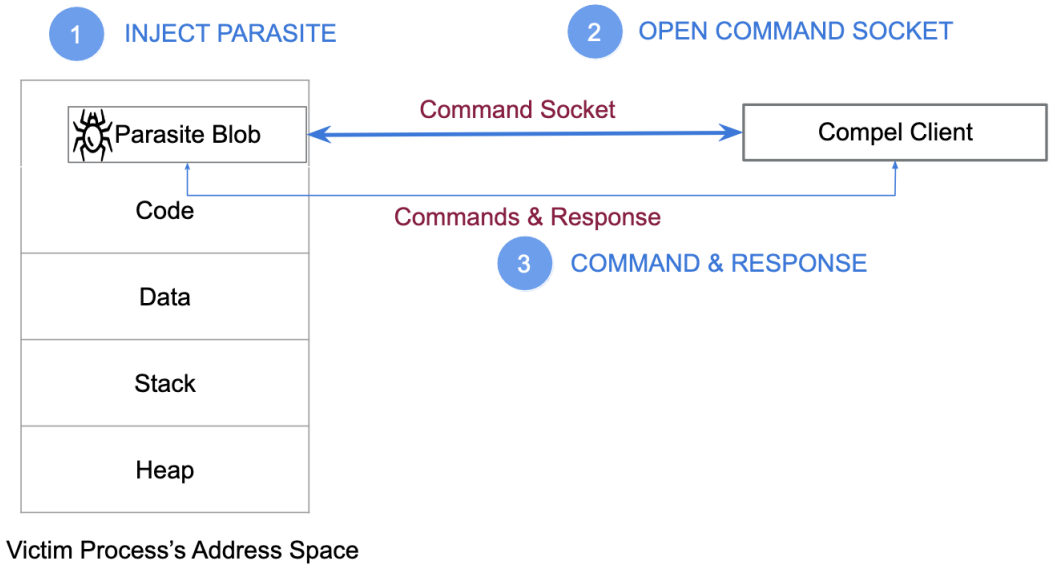

Figure 2.2: Compel and parasite communication.

2.2 Compel

Compel [2] is a tool to execute arbitrary code in the context of another process controlled via ptrace system call. The code that is executed in the foreign process context is called parasite code. The parasite code needs to be compiled with compel flags for it to be executed in the victim process's context.

In order to execute the parasite code, compel saves the context of the target process main threads (registers, signal mask). Then it scans the code section of the victim process and injects the parasite code into the victim's executable region. The parasite opens the control socket and executes the commands sent by the compel through the socket. Once the parasite completes the commands, it finishes at a predefined exit point. Then compel restores the victim process registers and signal mask, allowing the victim program to resume its execution. Figure 2.2 shows the communication between Compel and parasite code.

For executing system calls, compel also provides support without injecting the parasite code. In order to accomplish that, compel stops the victim process and injects the instructions for the desired system call. After executing the system call as the victim process, it restores the process execution.

2.3 Distributed Shared Memory Model

Distributed Shared Memory (DSM) [43] is a memory architecture where two or more physically separate memories can be aggregated to a single shared address space. The nodes are usually unicore or multicore machines connected by a high-speed network interface. The granularity of the memory blocks is usually the size of a page, so it can either be 4KB, 16KB, or higher.

From a page-level perspective, DSM can be implemented through page replication and page migration. Replication allows multiple copies of the same data to reside in different local memories, which allows read concurrency. Migration ensures a single copy of data exists at a given time so that only one machine can access the data at a time. Taken together, they satisfy the Singe-Writer/Multiple-Read (SWMR) and Data-value Invariants [40].

The implementation of DSM can be done either in hardware or software. While hardware-based implementations [11, 30, 31, 47, 48, 49] provide low latency, software-based implementations [12, 17, 18, 22, 24, 27, 28, 29, 34, 42, 52, 55] provide portability and flexibility on algorithms used in the consistency model.

One of the main characteristics of DSM is to maintain consistency and coherency among the connected nodes. MSI protocol [5], a cache coherency protocol, can be used to achieve sequential consistency in the DSM model but at the page granularity. Figure 2.3 and figure

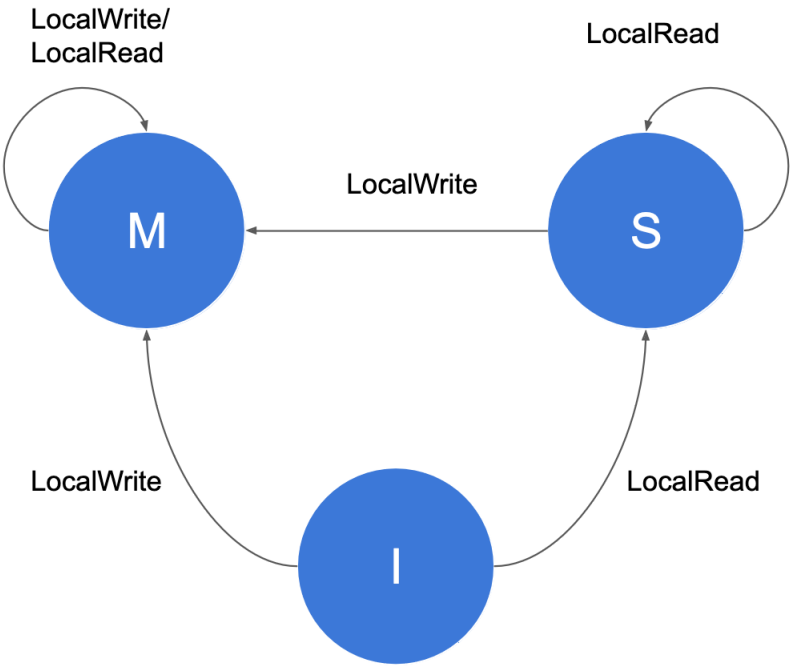

Figure 2.3: MSI state diagram for processor messages.

2.4 show the state diagram of page status based on processor and bus messages, respectively.

In the MSI protocol, each page can be in any one of the following states.

1. Modified (M): The page is modified and owned by only one node (page owner), and all the other nodes have stale data. The node that owns the page has exclusive write access to modify the page.

2. Shared (S): The page is shared among multiple nodes and owns the same data (share page owners). This state indicates that nodes only have read access to the page.

3. Invalid (I): The page owned by the current node is not a valid page and has been invalidated by other nodes. For read and write operations, the node needs to obtain the updated page.

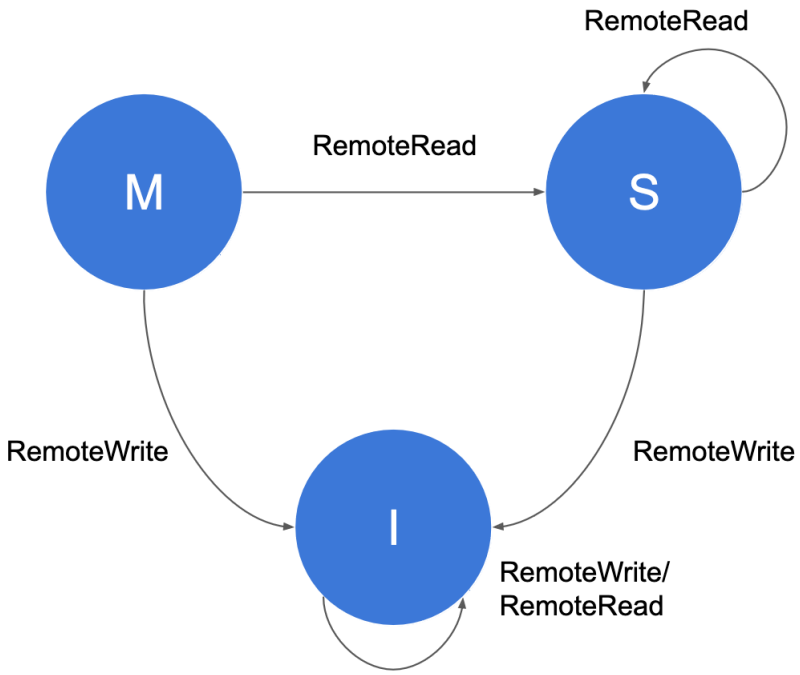

Figure 2.4: MSI state diagram for bus messages.

2.4 Userfaultfd

When a program tries to read or write a page that is not loaded in the memory or when the process does not have proper permission to access a page, the memory management unit (MMU) raises an exception called page fault [6]. Before accessing the page, it must be added to the process's virtual address space, and the actual page data should be loaded from a backing store like a disk or a network. When a page fault is raised by the MMU, the operating system resolves the fault by loading the page from the disk or denying access if it is not a valid request.

From kernel version v4.19, Linux introduced the userfaultfd [9] system call that allows applications to handle page faults for registered memory in userspace. The page fault of the registered memory can be handled in user space either through SIGBUS (bus error) fault

handler or via a file descriptor returned during the userfaultfd system call. When a page fault happens (e.g., when some memory has not been loaded into the current address space), the fault can be handled in another thread in the same process or in a different process by passing the userfaultfd file descriptor through IPC sockets. Listing 2.4 shows the definition of `struct uffd_msg` sent by the kernel to userfaulfd file descriptor.

```
1
2 struct uffd_msg {
3 ...
4     __u8  event,          /* Type of event */
5     __u64 flags,          /* Flags describing fault */
6     __u64 address,        /* Faulting address */
7 ...
8 }
```

Listing 2.1: Definition of userfaultfd message emitted during page fault.

While copying the page, the fault handler can set the page as read-only in order to get a notification when a write operation happens on the read-only page. This can be achieved by registering the memory region in write-protect mode. When a write operation is executed on a write-protected page, the fault handler receives a notification and updates the permissions to allow writing on the page.

2.5 Popcorn Linux

Popcorn Linux [37] is a system software stack for heterogeneous instruction-set architecture (ISA) hardware. Popcorn Linux contains a compiler, an operating system kernel, and a run-time system, which allows migrating live applications between heterogeneous-ISA hardware.

The Popcorn compiler toolchain includes a modified LLVM compiler and GNU GOLD linker. The popcorn generates multi-ISA binaries, which contain modified code and data sections with state transformation metadata that enables them to be migrated between different ISAs running Popcorn Linux OS kernels. After converting the C/C++ source file to LLVM IR, the migration points are inserted by the compiler at the beginning and end of functions. Migration points are semantically equivalent program points where cross-ISA execution migration is possible.

Since execution migration is possible only at migration points, threads must check with the Popcorn scheduler for migration, which is based on a scheduling policy. After reaching the migration point, the state transformation runtime built into the multi-ISA binary transforms the thread state to the destination ISA format. The registers, stack, and pointers are transformed through stack unwinding and rewriting.

2.6 HetMigrate

HetMigrate [38] is a framework, integrated with CRIU, to migrate Linux processes between heterogeneous ISA systems. HetMigrate runs on stock Linux and requires the program to be instrumented with equivalence points [13]. The equivalence points are function boundaries in the process, and for each equivalence point, the Popcorn compiler generates the live variable information and stores it in the DWARF [19] format in a custom section of the compiled binaries. HetMigrate extends the Popcorn compiler to allow external applications to suspend the process and restore the process at the equivalence points in a different ISA machine.

For dumping the programs at equivalence points, HetMigrate includes a *code-monitor* which uses the ptrace system call to indicate that the program needs to be suspended. It modifies the flag ___*indicator*___ (added by the Popcorn compiler) using ptrace. The *code-monitor*

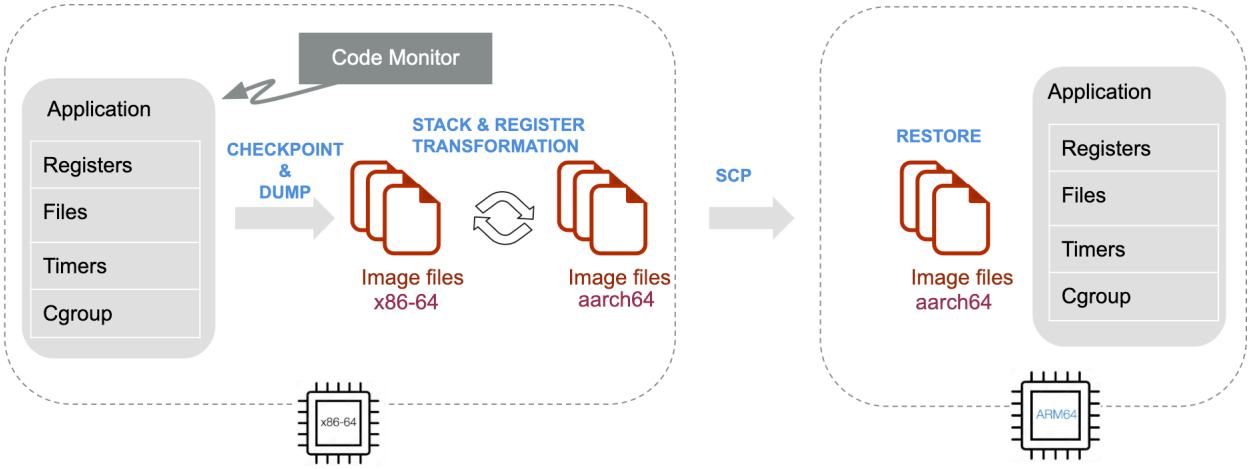

Figure 2.5: HetMigrate's workflow between x86-64 and aarch64 CPUs.

controls the execution of each thread until it reaches the equivalence points. Once all the threads reach equivalence points, the *code-monitor* suspends the process and starts dumping the CRIU images. Figure 2.5 shows the workflow of HetMigrate.

HetMigrate includes a utility to transform the dumped CRIU images into different architectures to enable cross-ISA migration. The registers and stacks of the process are transformed from source ABI format to destination ABI format by unwinding the stack and rewriting it frame-by-frame.

After transforming the images to the target ABI, HetMigrate restores the process from the transformed images. For that, HetMigrate invokes the CRIU RESTORE tool, which resumes the suspended process based on the images. While suspending the process, the monitor changes the program state from ALIVE to SLEEP. HetMigrate reverts this change and enables the program to execute immediately after restoration.

A limitation of HetMigrate is that it can only migrate the enter process (main threads and all the child threads) and does not support migrating a subset of threads of the process. CRIU-RTX supports offloading fewer threads to multiple nodes and thereby enables scale-

out. Sequential consistency is achieved by custom page management using userfaultfd.

Chapter 3

Related Work

Significant research work has explored migrating execution contexts across nodes to achieve improved performance and energy efficiency. These works aim to utilize the resources outside the machine boundaries like processors, memory, and disk.

This chapter explains the prior work done on process migration and distributed execution. Section 3.1 explores existing works in distributed shared memory frameworks. Section 3.2 describes the works related to container migration in heterogeneous-ISA systems. Section 3.3 explains previous works on live-application migration. Section 3.4 describes other closely related works of CRIU-RTX.

3.1 Distributed Shared Memory

DSM systems have been extensively examined in the past and are designed to offer a coherent memory view to distributed execution contexts, such as processes and threads, across multiple machines. Some of the existing works [14, 25, 41] rely heavily on custom APIs to explicitly manage the memory. Those APIs often limit what data can be shared among the threads (e.g., heap or data requested through mmap() call). This requires developers to write applications limited by the capabilities of these APIs. There are also DSM frameworks [23, 29, 50] that provide transparent shared execution without rewriting existing applications.

Based on the implementation, DSM systems can be categorized into software and hardware DSM. Software DSM works [12, 17, 18, 22, 24, 27, 28, 29, 34, 52, 55] are implemented in the userspace library or as modifications in operating systems, or as compiler-level modifications. They are slower compared to hardware implementations but can use high-speed interconnects to reduce page transfer and message-passing latency.

Hardware DSMs [11, 30, 31, 47, 48, 49] have low latency, but they are time-consuming to implement and do not scale for a larger number of nodes [45]. In CRIU-RTX, DSM is implemented in the software layer and works on stock Linux OS without any hardware/OS modification over any TCP/IP network.

Grappa [41] is a DSM framework for running data-intensive applications on commodity clusters. Grappa leverages parallelism to efficiently use processor resources and overcomes the cost of inter-node communications. Grappa also uses fine granularity than a page, reducing the page-fault overhead and allowing compiler optimizations on global memory accesses.

Grappa follows an approach called Partitioned Global Address Space (PGAS) [20] where the remote data accesses are explicitly handled by developers. Grappa provides a programming interface implemented in C++ and requires modification in the existing applications. Compared to Grappa, CRIU-RTX supports both homogeneous-ISA and heterogeneous-ISA systems, and CRIU-RTX does not require changes in existing applications.

DEX [29] is a Linux kernel extension that allows an application to transparently share its threads across multiple nodes. Though threads are executed in different machines, threads can run as if they were on the same node, i.e., the threads can access the distributed memory via normal load and store instructions. DEX achieves this by implementing a memory consistency protocol on page-level granularity in the Linux kernel.

DEX uses RDMA for transferring bulk data, like page data, to achieve low latency. While DEX offers shared execution similar to CRIU-RTX, it is limited to homogeneous-ISA hardware and requires modification in the operating system kernel. In contrast to that, CRIU-RTX can run on the stock Linux operating system and is portable across x86-64 and ARM64 ISAs. Integrated with HetMigrate, CRIU-RTX can scale-out a set of threads in a process across x86-64 and ARM64 machines and provide distributed memory access to threads running in different nodes using userfaulfd.

3.2 Container Migration in Heterogeneous-ISA Systems

Edge computing is a computing paradigm that allows devices in remote locations to process the data (instead of cloud servers) and aims to bring cloud services closer to the client. In order to provide low latency, the devices should be physically close to the client. So when the client is mobile, cloud services live-migrate the programs to edge devices.

The challenge in migrating applications in edge computing is that the devices can be of different ISA hardware. For instance, many of the edge devices run on low-powered ARM or MIPS architecture processors, whereas cloud servers are dominated by x86-64 processors. H-Container [56] overcomes this challenge by migrating containers between heterogeneous ISAs. H-Container is based on CRIU and Popcorn compiler, integrated with docker to migrate containers. Experiments with H-Container show there is no overhead on running the applications except for a 10-100ms delay during migration of the containers.

While H-Container supports live migration of the containers, CRIU-RTX enables the migration of threads between heterogeneous-ISA nodes. H-Container requires applications to be linked with the stack transformation library, while the CRIU-RTX uses HetMigrate [38] to transform the CRIU images externally to the target ABI format.

3.3 Live Application Migration in Heterogeneous-ISA Systems

Many research works investigated scaling systems better on high-core machines. New OS designs have been proposed, and specifically, multi-kernel designs have gained traction. Popcorn is one such replicated OS model based on Linux, which boots multiple Linux kernels on multi-ISA hardware. Applications running on Popcorn can migrate between machines exploiting resources in the heterogeneous nodes.

Popcorn Linux [13] work developed a framework with an operating system and compiler toolchain that allowed processes to share memory and execute in a heterogeneous-ISA environment. Popcorn creates a single system image and runs the applications on a modified Linux kernel. For migrating applications from one ISA machine to a different ISA machine, a state transformation runtime is added as part of the multi-ISA binary during the compilation stage. Popcorn's evaluation of applications shows a 52% speedup on applications compared to homogeneous-ISA systems.

libHetMP [36] is an OpenMP runtime for transparently distributing parallel computations across CPUs with different instruction set architectures (ISA). libHetMP decides workload distribution automatically without programmers intervention. It measures the performance characteristics and invokes a loop iteration scheduler to decide if cross-node execution can improve performance. libHetMP evaluations show OpenMP benchmark suites runs 41% faster across asymmetric CPUs.

CRIU-RTX functionally differs from Popcorn and libHetMP on the shared thread execution. Both Popcorn and libHetMP rely on operating system changes for the message layer and migrating program state. CRIU-RTX handles the page management operations in the

userspace using userfaultfd and uses CRIU to scale-out the application threads across different machines.

HetMigrate [38] is a closely related work, and CRIU-RTX uses the HetMigrate framework for transforming process states across different ISAs. HetMigrate uses the Popcorn compiler to instrument the multi-ISA binaries with equivalence points (beginning and end of the user-defined functions). At the equivalence point, instrumented threads check the flag for migration and allow the monitor program to guide them to the equivalence point. HetMigrate invokes the modified CRIT tool in the CRIU framework to transform the dumped CRIU images to target ISA. The tool unwinds the stack and transforms live variables, stack, and pointers to the target ABI (Application Binary Interface) format.

HetMigrate supports migrating both single-threaded and multi-threaded applications and uses *code-monitor*, a `ptrace` based tool in the HetMigrate framework, to suspend the application threads. The *code-monitor* attaches to each thread in the process and suspends the process when all the threads reach the equivalence point. A limitation of this work is migrating a subset of threads is not supported. CRIU-RTX supports shared thread execution with its userspace page management and offers memory sharing among the threads on the stock Linux operating system.

3.4 Other Closely Related Works

Venkat et al. were among the first to demonstrate the capability of heterogeneous multi-core architectures for high performance and energy efficiency. Their work reduced the overhead of program state transformation through dynamic binary translation till the program reaches the equivalence point. They explored the diversity offered by three modern ISAs: Thumb, Alpha, and x86-64. Their evaluation shows their architecture can provide 23% energy savings

and performance improvement as much as 21% compared to single-ISA systems.

Recent works in disaggregated memory systems [12, 52, 54] leverage high-speed low-latency interconnects and expands the limitations of memory in single machines by providing a large volume of memory. But these systems do not provide resources (e.g., processors or disk) other than memory. So the applications have to be manually distributed to utilize those resources.

Chameleon [35] is a framework for continuous stack re-randomization in userspace. It uses the userfaultfd and ptrace to transform the application's stack and inject newly-rewritten code. Similarly, Rave [16] provides a runtime to dynamically update the program execution states (e.g., internal stack layout, instruction sequences, and machine node to run on). Rave also uses the userfaultfd to reload the randomized code and data pages. Like Chameleon and Rave, CRIU-RTX manages the pages of an application in userspace without any components in the kernel layer of the operating system.

Chapter 4

Design

CRIU-RTX extends CRIU to transparently scale-out threads across multi-node systems connected through TCP/IP networks. It uses userfaultfd for managing page faults in userspace and tracks page ownership to provide memory consistency across the nodes.

The chapter is organized into the following sections. Section 4.1 explains the overall design of CRIU-RTX's thread scale-out mechanism. Section 4.2 discusses how pages are selected for shared memory. Section 4.3 explains the DSM components in CRIU-RTX, which provides memory coherency. Section 4.4 describes the design choices for coherency protocol in CRIU-RTX. Section 4.5 explains different page request messages used in the DSM implementation.

4.1 Overall Design

CRIU-RTX allows multi-threaded programs in a machine to suspend a subset of threads and restore their execution in different machines. Figure 4.1 shows the overview of CRIU-RTX's thread scale-out procedure. Some of the terms necessary to understand the design are listed below.

1. Code-monitor: It is ptrace based tool that can control a target program's state (like suspending or resuming at an instruction) and modify its address space (changing a global variable or inserting an instruction).

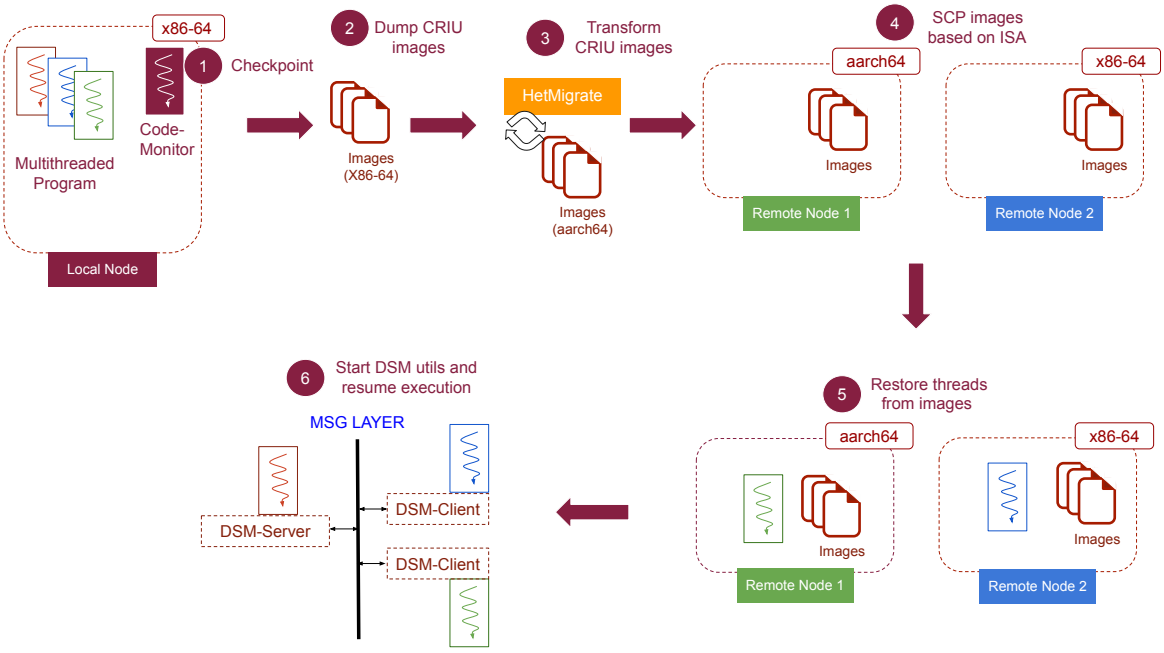

Figure 4.1: CRIU-RTX flow for scaling-out threads to remote nodes.

2. Local node: The machine where the target process is launched. *code-monitor* runs in this node and initiates scaling-out threads based on user input.

3. Remote node: This is the hardware (a server or an embedded board) connected to the *local node*, where the threads initially suspended from the *local node* are restored. This node can be of the same instruction set architecture (ISA) as the *local node* or a different ISA.

The high-level steps for the thread scale-out mechanism are:

1. With the target PID as input, the *code-monitor* attaches itself to the target multi-threaded program. For each thread, the *code-monitor* creates its own child threads to guide them to the equivalence point (program points where program state transformation is possible). For the main threads blocked at `pthread_join()`, the *code-monitor*

uses POSIX-signal based mechanism to wake the thread. This technique is explained in detail in the implementation chapter 5.

2. Once all the threads reach the equivalence point, the process is suspended, and the *code-monitor* launches CRIU DUMP to save the metadata of the process as protobuf images on disk.

3. Based on the nodes connected (homogeneous-ISA or heterogeneous-ISA), CRIU-RTX decides to transform the CRIU images for the target ISAs using HetMigrate. Using the modified CRIT tool, CRIU-RTX enables restoring a set of child threads on the remote nodes.

4. The corresponding images based on ISA are transferred to the *remote nodes*, and CRIU RESTORE is invoked to create a new task from the images.

5. CRIU RESTORE spawns a new process with the child threads of the target process. Once the threads are restored, the components for the distributed shared memory (DSM) are launched, and all threads will resume their execution.

4.2 Selecting Pages for Shared Memory Updates

Once the threads are scaled-out across machines, threads can access both the local memory (memory accessed only by the thread itself) and the shared memory (memory that is accessed by multiple threads) during their execution. To provide a consistent view of memory, CRIU-RTX allows multiple threads running in different machines to access and manipulate the same page. The main objective in the design of the CRIU-RTX is to eliminate OS dependencies. Therefore, it is necessary to handle the page faults in userspace instead of allowing the operating system to handle the page faults. CRIU-RTX relies on userfaultfd, which defers

page fault handling to userspace applications. This section explains how pages are selected to be part of the shared memory and tracked with userfaultfd.

After dumping the images with CRIU DUMP, the CRIU-RTX scans the VMA of the target process and prepares the list of the pages to be managed by the userfaultfd. The operating system utilizes virtual memory areas (VMAs) to track the memory mappings of a process. CRIU-RTX gets the VMAs of the process through procfs entries and prepares the tracking list based on each segment in the process's VMA.

1. Code: The code sections of the process are already handled by CRIU. CRIU requires binaries in the same path in source and destination nodes. Hence during restoration, CRIU RESTORE will load the code section from the binary. Therefore pages mapped to the code section are ignored.

2. Stack: Each thread maintains its own stack, and those pages are ignored. In some multi-threaded applications, the main thread might allocate memory for shared data in the stack and pass it as a pointer to the stack to the child threads. To simplify the design, those applications are not supported by CRIU-RTX.

3. Data: The data section is common to all the threads, and any thread can access those pages in this section. They are added to the tracking list.

4. Heap: Pages allocated based on dynamic memory allocation APIs like *malloc* are tracked as they can be accessed by different threads.

5. *mmap* pages: Pages requested by mmap do not reside in any of the above-mentioned sections. If the pages are anonymous, i.e., not backed by files, they are added to the tracking list.

6. Other pages: Miscellaneous VMAs like vdso, vsyscall, and vvar are also in the process

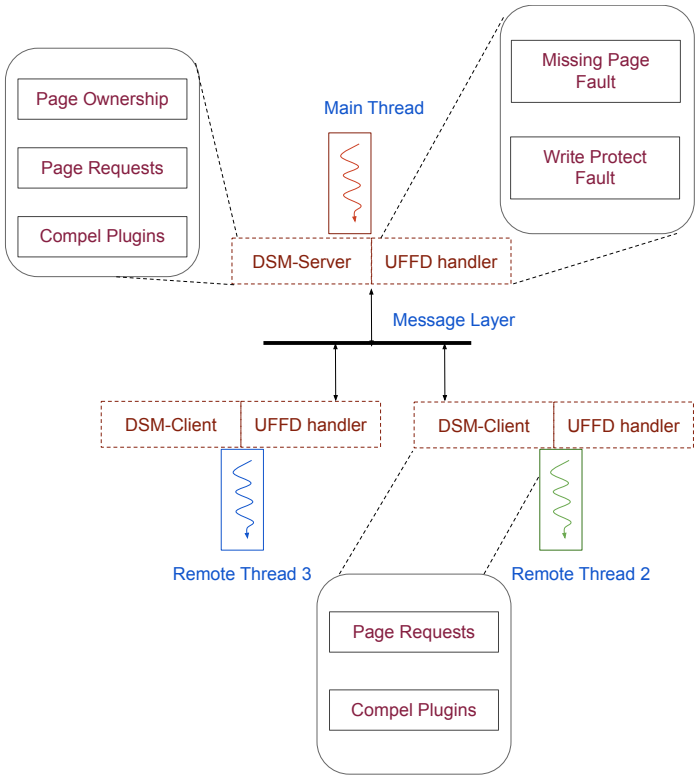

Figure 4.2: Components in CRIU-RTX for maintaining coherency.

address space. These are the mechanisms [10] used to reduce system call overhead, and those pages are not added to the tracking list.

4.3 DSM Components

CRIU-RTX provides a coherent view of the shared virtual memory transparently to threads running in different machines. CRIU-RTX allows only one node to write to an address while no other process updates or reads the data from the address at the same time. This section explains the components of DSM implementation and their responsibilities in providing distributed shared memory access across the nodes. Figure 4.2 shows an overview of these components.

1. Userfaultfd handler: Each node has one userfaultfd handler attached to the thread, and it handles the page faults raised by the thread running in the node.

2. DSM server: The local node launches the DSM server after scaling-out its child threads, and the DSM server is responsible for handling page ownership and page data across the remote nodes. Any remote node requests for page handling, like page data requests and page invalidation, will be sent to the DSM server.

3. DSM client: Each remote node has one DSM client process that connects to the DSM server running in the local node and waits for page-handling messages. Based on the message, the DSM client handles page management requests like retrieving the page data and invalidating pages using CRIU's Compel plugin.

4.4 Design Choices for Memory Coherency

To utilize the parallelism in the multi-threading paradigm, allowing the threads to own copies of a page for read-only operations is necessary. If page sharing is not enabled and only one node can access a page at a time, it will provide strict memory consistency, but it will be a huge performance bottleneck as there will not be any parallel executions.

To allow parallelism, DSM systems should allow page replication *i.e.,* multiple nodes can have a copy of the same page simultaneously. But while using a replication strategy, coherency should be maintained. For a memory to be coherent, the data returned by a read instruction should be the same as the data written by the most recent write instruction at the same address. To enforce coherency, there are two protocols that exist.

1. Write-Update protocol: In this protocol, if a node updates a shared page, all the pages are updated. The change in page data is propagated to all the shared owners of a page.

2. Write-Invalidation protocol: In this scheme, if a node writes to a shared page, all other copies of the page are invalidated. At a time, only one node will have exclusive access to the page for writing.

Li and Hudak's [32] work showed the write-invalidation protocol performs better than the write-update protocol in many applications. It is because propagating the updated page data to all the shared owners of the page will increase the network traffic as the systems scale. The writer-update protocol will be efficient in some instances, like hardware DSM implementation, where the latency can be reduced. CRIU-RTX is a software implementation of DSM in userspace, hence the write-invalidation protocol is the better choice for maintaining coherency.

4.4.1 Page Ownership

To share pages across the nodes, each node should know which node contains the latest page content and should have access to the page. One way to do that is to broadcast the page data request to all the nodes in the network, and the node that owns the page can respond to the requested node. The drawback of this method is that those nodes that do not have the page also need to process those messages. Another way is to have a centralized server that maintains the page ownership data and knows which page is owned by which node. Although this serializes the page requests, it makes the design simple and works well if multiple nodes do not compete for the same page. This case is applicable for compute-intensive workloads where the threads spend most of the cycles in computations.

CRIU-RTX implements the centralized server method by maintaining the page ownership table and processing the page requests on a first-come, first-served (FCFS) basis. Figure 4.3 shows a sample page ownership table for a set of pages. A node can be an *owner* with

Page Address	Page Status	Owners
0xffffa000	Modified	Local Node
0xffffb000	Shared	Local Node, Remote Node 1 & 2
0xffffc000	Invalid	Remote Node 1

Figure 4.3: Page ownership table maintained by the DSM server.

exclusive read/write access to a page or can be a *shared-owner* with read-only access shared with a few other nodes.

4.5 Page Request Handling

In CRIU-RTX, the page faults are handled in userspace in a fault handler thread which waits for the userfaulfd notifications. To serve the page fault, the userfaultfd handler creates page requests composed of DSM message packets and sends them to the DSM server. The DSM server, which maintains the page ownership table based on the message type, fetches the page or invalidates the page from the page *owner* node. Figure 4.4 shows the operations followed for each DSM message. There are three types of page requests that the userfaultfd thread can send to the DSM server:

1. MSG_GET_PAGE_DATA: This message is sent when the fault is due to a missing page, and the remote node needs to read the page content to resolve the fault request.

2. MSG_GET_PAGE_INVALIDATE: This message is similar to the previous message, except the requested node needs exclusive write access to the page. Hence after the page owner node sends the page contents, the page is dropped by the *owner*, and the

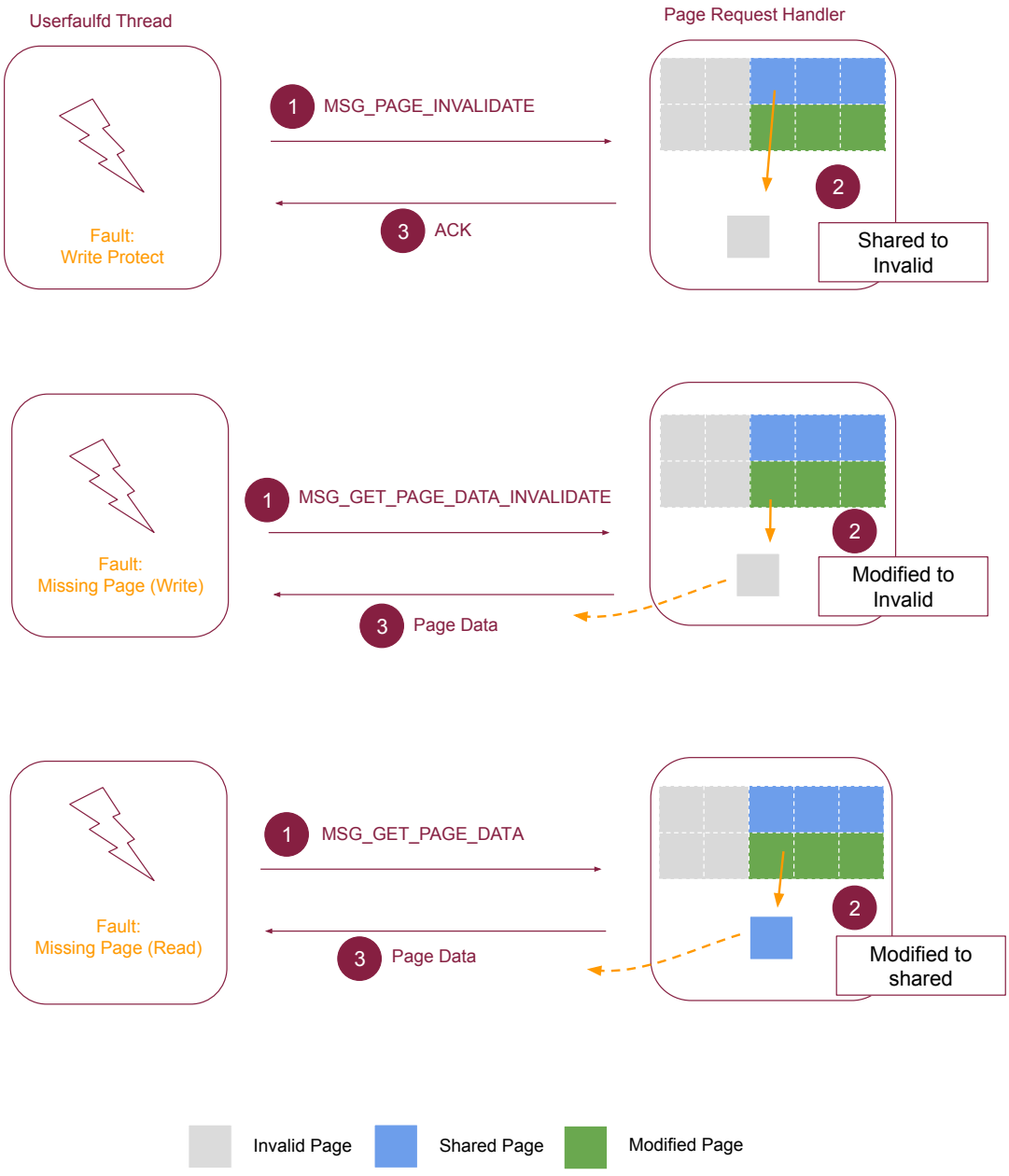

Figure 4.4: Page request operation from userfaultfd thread.

requested node is the new textitowner of that page.

3. MSG_PAGE_INVALIDATE: This message is sent when the node is a *shared-owner i.e.,* the page ownership is shared by other nodes and the node wants to write to the page. So the DSM server broadcasts this message to the other *shared-owners* and instructs them to drop the page.

Chapter 5

Implementation

The CRIU-RTX framework was implemented as an extension of the HetMigrate version of CRIU with userspace distributed shared memory (DSM). CRIU-RTX uses HetMigrate to transform the CRIU images to a different instruction set architecture (ISA) and avoids operating system dependency by using userfaultfd for userspace page fault handling. This chapter discusses the implementation details of CRIU-RTX, which includes *code-monitor*, userspace page fault handler, and Compel-based page management utility.

The chapter is organized into the following sections. Section 5.1 explains the *code-monitor* extension for POSIX multi-threaded programs. Section 5.2 discusses the thread restoring mechanism across *remote nodes*. Section 5.3 explains the userfaultfd handler's page fault workflow. Section 5.4 describes the page management operations with Compel.

5.1 Code-Monitor Extension for POSIX Multi-threaded Programs

For scaling-out application threads across heterogenous-ISAs (i.e., x86-64, ARM64), the threads in a multi-threaded program need to be stopped at equivalence points [13]. Equivalence points are semantically equivalent points in the programs where execution migration to a different ISA is possible. These points are instrumented by the Popcorn compiler.

CRIU-RTX reuses the *code-monitor* tool from the HetMigrate framework with modifications to support multi-threaded programs. *Code-monitor* is a ptrace-based tool that suspends the process at an equivalence point. The challenge with multi-threaded programs is that, in most POSIX programs, the main thread will be waiting at the `pthread_join()` library call, and it is not possible for the *code-monitor* to suspend the main thread without allowing child threads to finish their execution. To overcome this, CRIU-RTX instruments the code at `pthread_join()` to alter its program flow upon a signal from the *code-monitor*. Listing 5.1 shows the instrumented code before `pthread_join()`.

Before calling `pthread_join()`, the instrumented main thread registers a signal handler for the `SIGUSR1` signal and sets the flag `__pthread_join_enter`. *code-monitor* reads this flag with the `ptrace()` system call and sends the SIGUSR1 signal to the main thread. The signal handler then invokes the `longjmp()` call, which changes the instruction pointer to the return statement of `setjmp()`. Then the main thread reaches the equivalence point and executes the trap instruction. The code change for this mechanism is described in Listing 5.1.

```
1 signal(SIGUSR1, sig_handler);
2 __rollback_flag = setjmp(buf);
3 if(__rollback_flag)
4     function_with_equivalence_pt();
5 __pthread_join_enter=1;
6 pthread_join(thread, NULL);
```

Listing 5.1: Code changes to suspend main thread.

The following code is the signal handler for the SIGUSR1 signal sent by the *code-monitor*.

```
1 void sig_handler() {
2   /* set __rollback_flag to 1 */
3   longjmp(buf, 1);
4 }
```

Listing 5.2: Signal hanlder definition.

5.2 Restoring Threads on Remote Nodes

After suspending the threads at equivalence points, CRIU DUMP stores the program state as protobuf images in the disk. If one needs to restore the process on a different node, the images need to be copied to the *remote node*, and CRIU restore will restore the program from the images. For a multi-threaded program, CRIU will restore and resume all the threads. But for CRIU-RTX, only selected threads need to be restored at a node. To achieve this, CRIU-RTX uses the CRIT tool from the CRIU framework to modify the dumped pages.

```
{
    "magic": "PSTREE",
    "entries": [
        {
            "pid": 121875,
            "threads": [
                121875,
                121876
            ]
        }
    ]
}
```

Listing 5.3: JSON output of sample pstree image.

CRIU stores the process IDs and threads IDs in an image called the pstree image. For a multi-threaded process, a list of the thread IDs is stored in a list named `threads`. CRIU-

RTX converts the pstree.img to a JSON [4] file and edits the `threads` member in the JSON file. Listing 5.3 shows the JSON output of the sample pstree image with PID 121875.

Based on the details in the pstree image, CRIU RESTORE loads the core image file from the disk. For each thread in the process, CRIU will look for a core image with its thread ID (TID), i.e., core-<TID>.img. The contents in the core image file differ for main threads and child threads. Each core image contains the registers of the respective threads during the checkpoint. But for the main thread's core image, it also has an additional field called `tc` (Task Context). In order to restore the child threads without the main thread, the core images should contain the `tc` field. CRIU-RTX copies the `tc` field from the main thread's core image and copies it to the child thread's core image. With the modified core image with `tc` data, CRIU can restore these child threads alone as standalone processes.

5.3 Userfaultfd Thread Workflow

CRIU-RTX invokes the page fault handlers on the *local* and *remote nodes* to maintain coherency in shared memory. There are two types of fault events. One is for a missing page, which is triggered when the thread tries to access the content of a page owned by a different node. Another fault event is the write-protect fault when a page is shared among two or more threads and the thread needs to write to that page. For both of these faults, the page fault handler on the *remote nodes* and the handler on the *local node* send a DSM message to the DSM server. Figure 5.1 shows the structure of the DSM Messages.

In order to receive the DSM messages from the fault handlers on the *local node* and *remote node*, the DSM server uses the poll() system call. The *local node*'s fault handler communicates with the DSM server using UNIX pipes and with the *remote node*'s fault handlers using network sockets. The file descriptor for the *remote node*'s sockets and the write-end of the

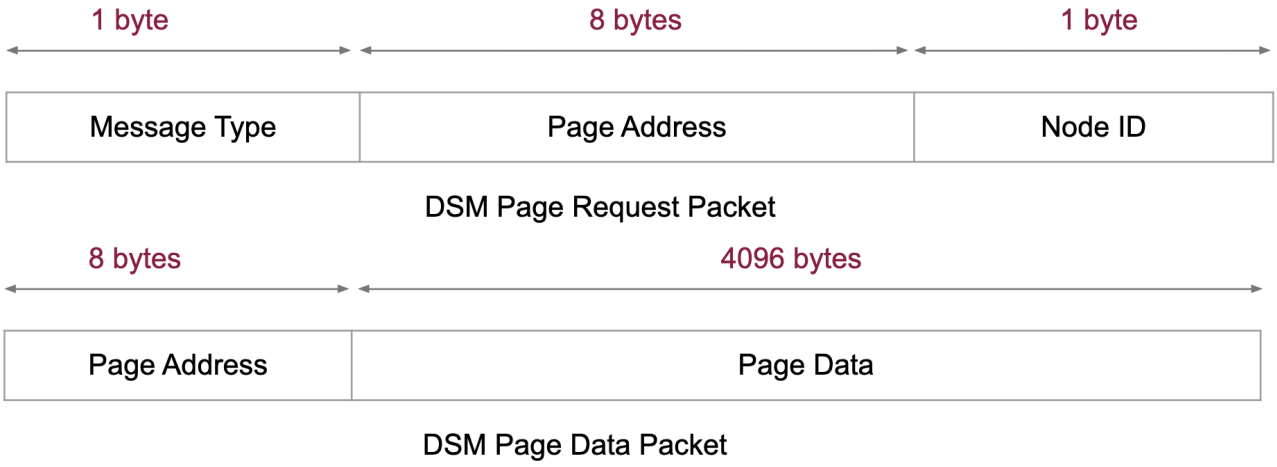

Figure 5.1: DSM message structure.

local node's pipe are polled by the DSM server.

It is the responsibility of the DSM server to maintain the page ownership table based on the DSM messages received by the server.

```
1 struct page_data {
2         long addr;
3         short owner;
4         unsigned char shared_owners;
5         int page_status;
6 }
```

Listing 5.4: `struct page_data` definition.

This data is maintained in a list of `struct page_data` data structures. It contains details of the current *page owner*, *shared owners* (as a bit vector, if it is a shared page), and page status with respect to the *local node*. For each page fault, the userfaultfd threads send one DSM message to resolve the page fault. Figure 5.2 shows an overview of the userfaultfd thread's operations. Listing 5.4 shows the definition of `struct page_data`.

5.4 Page Management with Compel

CRIU-RTX implements the write-invalidate [40] protocol at page-level granularity to maintain coherency. This allows multiple nodes to own the same copy of the page for read-only operations and a single node to own the page for write operations. With userfaultfd, CRIU-RTX is notified when a node wants to access a page when it is owned by a different thread (missing-page fault) and also when a node wants to write to a shared page (write-protect fault). When these events happen, userfaultfd sends a DSM message to the DSM server, which handles the page ownership and responds to the DSM messages.

If the *local node* owns the page, it will respond to the request; else it will forward the request to the *remote node* that owns the page. In both cases, CRIU's Compel plugin is used to manage page data by running parasite code in the target thread's context.

There are three different types of DSM messages. For each message, Compel modifies the thread's page data based on the message type.

1. MSG_PAGE_INVALIDATE: To invalidate the page, the Compel runs the `madvise()` system call with `MADV_DONTNEED` flag in the victim thread context to drop the page.

2. MSG_GET_PAGE_DATA: To get the page data from the victim thread, Compel runs the `vmsplice()` system call. `vmsplice()` maps the address of the thread to a pipe and returns the file descriptor of the pipe. Compel sends the read-end file descriptor of the pipe to the DSM server, which reads the page data from the descriptor and forwards it to the userfaultfd handler thread.

3. MSG_GET_PAGE_DATA_INVALIDATE: For this message, Compel performs both the operations from the previous messages, i.e., it fetches page date with `vmsplice()` and after that, drops the page using the `madvise()` call.

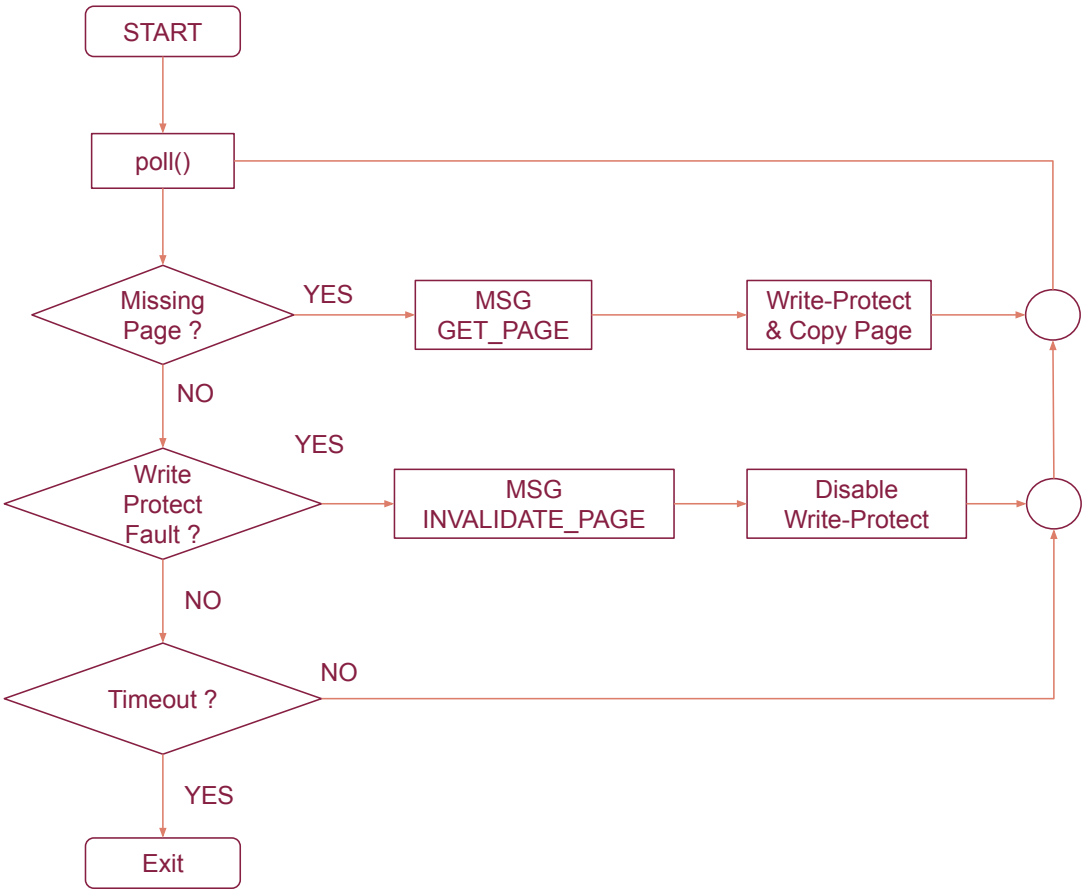

Figure 5.2: Userfaulfd fault handling thread workflow.

Chapter 6

Evaluation

CRIU-RTX aims to improve the performance and energy efficiency of servers by scaling out processes across machines. We evaluated the benefits of CRIU-RTX by measuring throughput and energy efficiency using compute-intensive applications. Our evaluations show 21% to 43% performance gains while scaling-out applications across servers with congestion and improvement in the energy efficiency of up to 18% while scaling-out applications across the server and embedded boards.

The chapter is organized into the following sections. Section 6.1 explains the experimental setup and the applications used for evaluating CRIU-RTX. Section 6.2 discusses the improvement in energy efficiency with CRIU-RTX. Section 6.3 explains the performance gain with CRIU-RTX while running applications on a server with co-existing applications. Section 6.4 explores the optimizations for false page-sharing and results after applying the optimizations. Section 6.5 explains the experimental results of CRIU-RTX on servers without resource congestion.

6.1 Workloads and Experimental Setup

For measuring energy efficiency, we utilized an x86-64 Xeon server as the *local node* and three Raspberry Pi 4 boards (hereafter addressed as RPi4) as *remote nodes*. And for performance measurement, we used the Xeon servers (m510) from Cloudlab's [21] Utah cluster, which

demonstrates the easy deployability of CRIU-RTX. All these machines run on Ubuntu Server 22.04 operating system with a v5.14 kernel.

For an application to be scaled-out across heterogeneous-ISA nodes, applications need to be compiled with the Popcorn compiler, which instruments the compiled binaries with state transformation metadata. Due to the limitations of the Popcorn compiler, only a few applications can be compiled with CRIU-RTX. We evaluated three multi-threaded micro-benchmarks with different page-sharing patterns and three multi-threaded macro-benchmarks to demonstrate the benefits of CRIU-RTX.

The applications used for evaluation are listed below.

1. Micro-MultiWriter: Multiple threads will *write* to the same page simultaneously.

2. Micro-MultiReader: Multiple threads will *read* from the same page simultaneously.

3. Micro-SingleWriter-MultiReader (Micro-SW-MR): Single thread will write to a page, and the remaining threads will read from the same page.

4. Blackscholes: Multi-threaded financial analysis application from the Parsec benchmark suite [15].

5. Grep: Command-line tool for searching plain-text data. We used an in-house multi-threaded version of grep for evaluating CRIU-RTX.

6. Matrix Multiplication: Multi-threaded matrix multiplication program to demonstrate the effectiveness of CRIU-RTX over compute-intensive workloads.

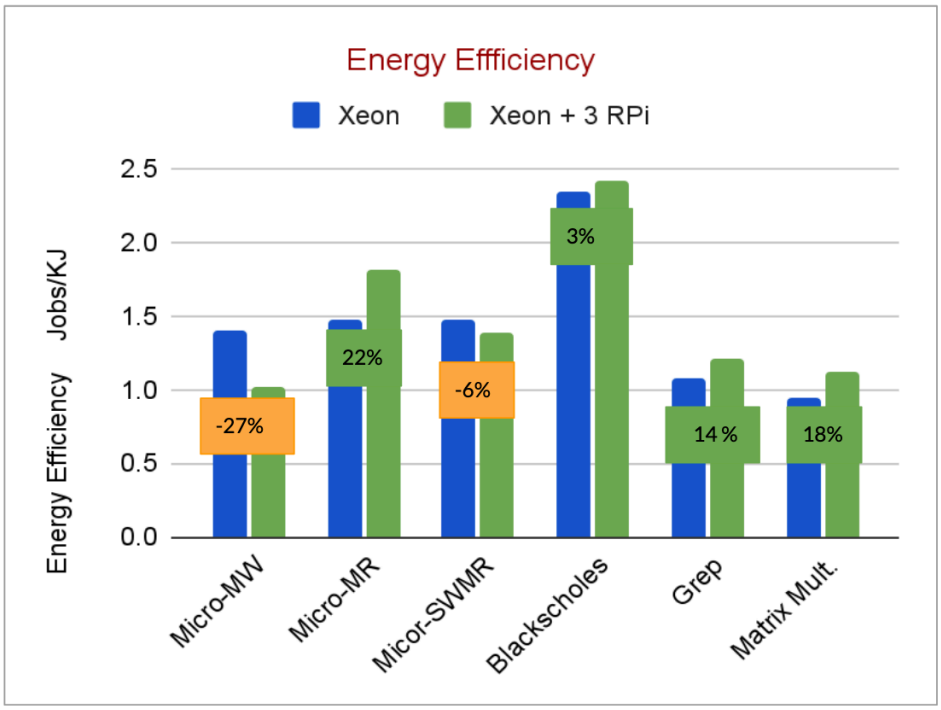

Figure 6.1: Energy efficiency gains of CRIU-RTX.

6.2 Energy Efficiency

The energy efficiency is evaluated using a Xeon server equipped with 3 RPi4 boards connected by a 100 Mbps Ethernet network. The power consumption of these systems was measured using a SURAIELEC Energy Watt Meter. The multi-threaded programs are launched in the Xeon server, and once the child threads are created by the programs, CRIU-RTX will scale-out the child threads across the RPi4 boards. Each benchmark was executed for 30 mins in the Xeon server (local execution) and Xeon with RPi4 (scaled-out execution) combination. Figure 6.1 shows the jobs executed per Kilo-Joule in both setups. For micro-benchmarks, Micor-MultiReader has the highest energy efficiency (22%) while running in *scaled-out execution* mode. Similarly, Matrix Multiplication gains 18% energy efficiency on *scaled-out execution* compared to *local execution*.

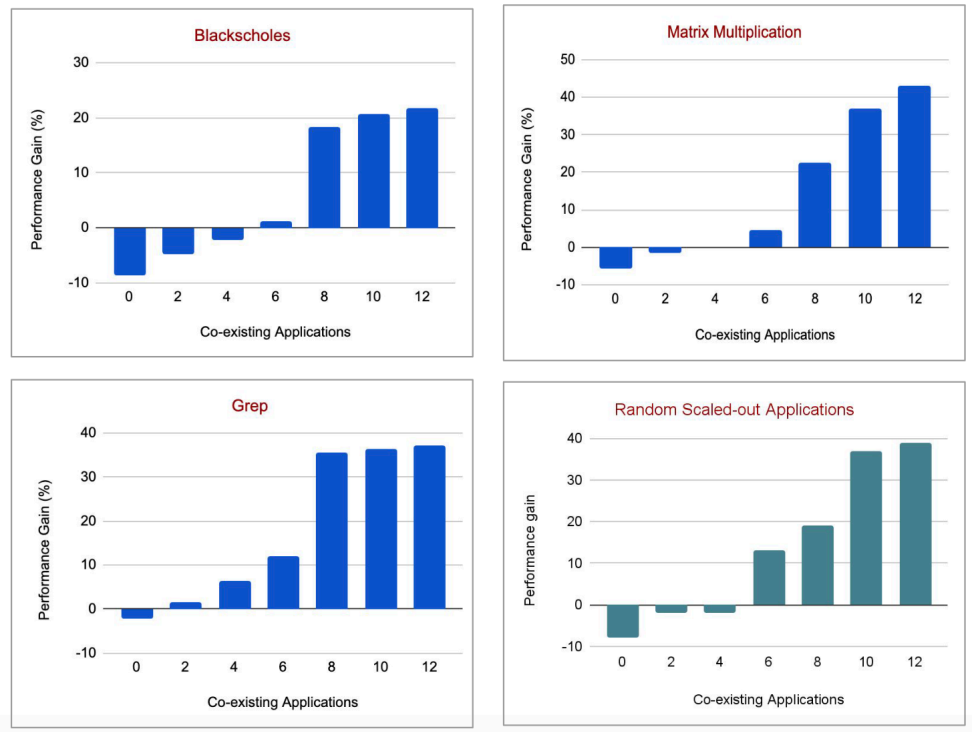

Figure 6.2: Performance gain with scaled-out execution during server congestion.

6.3 Performance Gain on Server Congestion

This section discusses the performance gain on macro-benchmarks when the server is congested with other co-existing applications using the server resources. We launched NAS Parallel Benchmarks (NPB) as co-existing applications to introduce resource congestion in the server. The experiments are done in Cloudlab's Xeon servers, running benchmarks with 4 threads scaled-out across four servers of the same configuration. Figure 6.2 shows the performance gain while increasing co-existing applications on *local execution* vs. *scaled-out execution*.

The performance of benchmarks in *scaled-out execution* with less than six co-existing applications is less compared to *local execution*. This is because the performance improvement

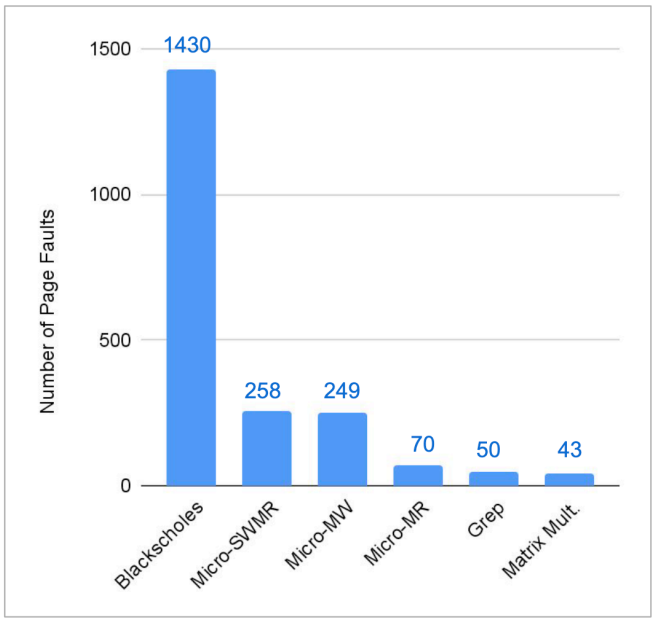

Figure 6.3: Number of page faults during scaled-out execution across 4 nodes.

with *scaled-out execution* exceeds the overhead of CRIU-RTX (due to thread scale-out time and time spent on page faults) only after six co-existing applications in the server. Matrix Multiplication gains 43% improvement in throughput with 12 co-existing applications, followed by Grep with 37% and Blackscholes with 21% gain, respectively. We also selected the applications randomly with uniform distribution to scale out across 4 servers, gaining a 38% improvement in performance.

We also measured the page faults triggered by these applications, which shows the distributed memory access pattern of the threads during *scaled-out execution*. With more page faults, there will be more overhead during *scaled-out execution*. Blackscholes, Micro-SingleWriter-MultiReader, and Micro-MultiWriter applications incur higher page faults during their execution.

Figure 6.4 shows the amount of time spent on the page fault operations with respect to network transfer and compel operations. For page faults due to missing pages, up to 17%

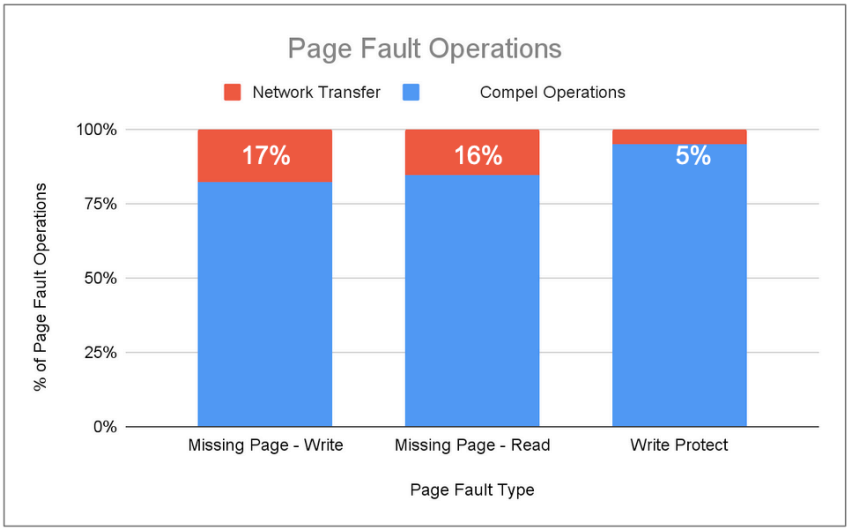

Figure 6.4: Comparison of page fault operations based on computational and network transfer cost.

time is spent on the DSM message and page content transfer.

6.4 Optimization for False Page Sharing

False page-sharing is a memory access pattern when multiple threads try to access different memory within the same page. This pattern degrades the performance since multiple threads accessing the same page will lead to a higher page fault count. We tested the performance loss due to false page sharing in two microbenchmarks, Micro-MultiWriter and Micro-SingleWriter-MultiReader. We also modified these applications to avoid false page-sharing by padding the data structures accessed by the threads such that each data structure is on a different page.

Figure 6.3 shows the execution time difference for applications with false page-sharing optimization over an unoptimized version. Micro-MultiWriter gains an 18% improvement in

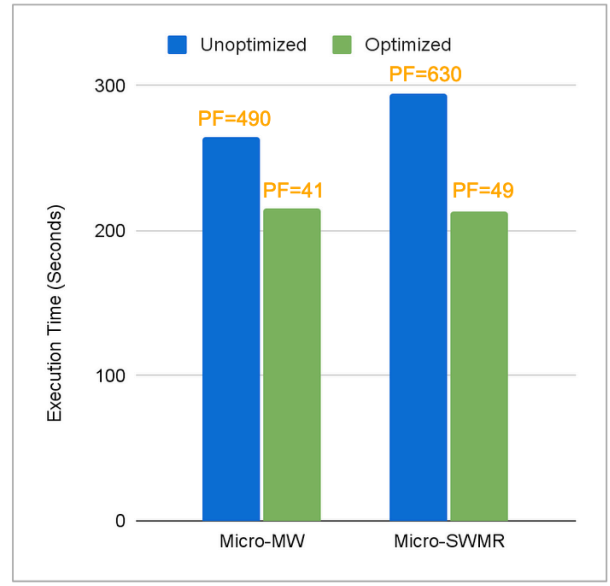

Figure 6.5: Execution time of micro-benchmarks with and without false page sharing optimization.

performance, and Micro-SingleWriter-MultiReader gains a 27% improvement when false-page sharing is removed. Padding the data structures to place them on different pages will increase the memory usage of the programs. We modified these benchmarks manually by identifying memory access patterns. This is one of the limitations of CRIU-RTX since optimizing false page-sharing for complex applications will be time-consuming work.

6.5 Performance on Servers without Congestion

This section shows the performance of CRIU-RTX when there is no resource congestion in servers (i.e., no co-existing applications). Figure 6.6 shows the execution time of applications with *local execution* and *scaled-out execution*.

For *local execution*, we measured the performance on two configurations, a Xeon server with

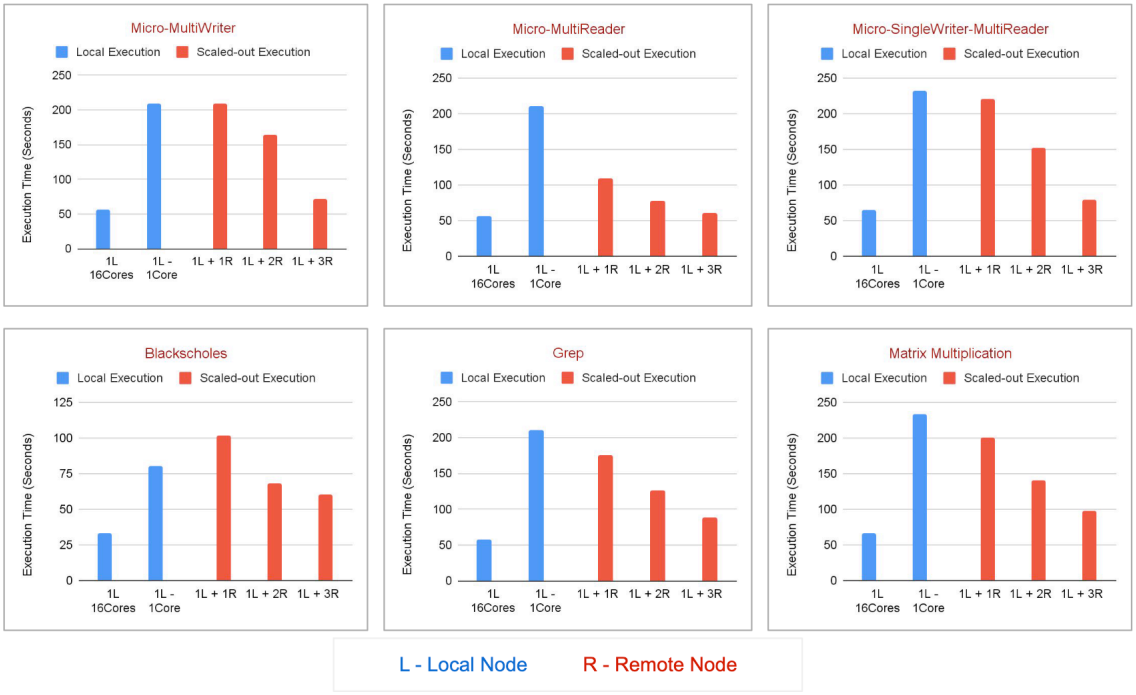

Figure 6.6: Execution time of applications running on a server without congestion.

16 cores and the same server with only one core enabled. In both cases, applications have four threads each and during *scaled-out execution*, applications thread count is the sum of a number of local and remote nodes. Grep, Matrix Multiplication, and Micro-MultiReader applications perform better compared to their single-core *local execution*. This is due to the number of page faults in these applications being less compared to the other three applications.

Chapter 7

Conclusion

7.1 Contributions revisited

In this thesis, we presented the design, implementation, and evaluation of a userspace framework, CRIU-RTX, to transparently scale-out application threads across machines. CRIU-RTX helps to reduce resource congestion in data centers by allowing scaled-out execution of applications with distributed memory access. CRIU-RTX implements a distributed shared memory model in userspace that can run on stock Linux operating systems providing easy deployability.

This thesis shows that it is possible to develop an efficient DSM system in userspace utilizing mechanisms like userfaultfd and Compel in the Linux Operating system. The bottleneck for CRIU-RTX's DSM efficiency is communication cost, not computational cost (such as due to any heavy code paths in the OS or application or mode/context switch cost). Hence with faster networks, CRIU-RTX's performance and energy gains will be even better.

The major contributions of CRIU-RTX can be summarized as follows:

1. With HetMigrate's transformation runtime, CRIU-RTX enables the programs to run in a scaled-out execution mode leveraging resources of multiple nodes of different instruction section architecture (ISA) connected through an Ethernet network.

2. CRIU-RTX allows threads in the applications to access distributed memory transparently during scaled-out execution without any operating system dependency. With Linux's userfaulfd feature, an MSI-based coherency protocol is implemented at page-level granularity in userspace to enable page-sharing among the threads running on different nodes.

3. We evaluated the performance of CRIU-RTX in server and embedded board configurations. Our evaluations show 21% to 43% performance gains while scaling-out applications across servers with congestion and energy efficiency gains up to 18% while scaling-out applications on a server equipped with low-cost embedded boards.

The following subsections discuss the limitations and future works that can enhance CRIU-RTX.

7.2 Limitations

The limitations of CRIU-RTX are partly inherited from HetMigrate and CRIU, in addition to its own limitations. Some of them are discussed in this section.

1. One of the major limitations of CRIU-RTX is that the applications need to be compiled by HetMigrate's Popcorn compiler in-order to transform the CRIU images to different instruction set architecture. This introduces the following limitations.

   (a) Application written only in C language can be compiled.

   (b) Applications need to be compiled with the musl-C library and need static linking. Without these changes, applications must be analyzed dynamically to correct the

offset in the memory organization of heap and stack objects. This will increase the performance overhead while scaling-out threads to different ISAs.

2. Another limitation is due to performance degradation from false page-sharing. False page-sharing is a memory access pattern where two threads try to access the same page but have different memory addresses within the page. Since CRIU-RTX uses the write-invalidate coherency protocol, only one thread can have exclusive write access to a page. In the evaluation section 6.4, we showed performance gain when false page-sharing is eliminated by modifying the application. But analyzing the memory access pattern of each application and modifying them will be time-consuming for developers.

3. Bandwidth of Ethernet networks connecting the nodes with each other also limits the performance of CRIU-RTX. Although servers are usually connected through 10 Gbps Ethernet, embedded boards are still manufactured with 1 Gbps Ethernet NICs. Improving the network speed between the nodes will improve the performance of CRIU-RTX.

4. Possible security concerns in CRIU-RTX are similar to those in virtual machine environments, like denial of service, side-channel attacks, etc., CRIU-RTX allows applications to consume resources outside the machine boundaries. Thus CRIU-RTX enables resource sharing by allowing applications to use idle computing nodes. Without proper security policies and configurations, this might provide unlimited access to the remote node's resources (like the filesystem and network) to a malicious process leading to denial of service attacks. This can be mitigated by limiting resources consumed for migrated threads.

5. CRIU-RTX does not fully support POSIX pthread synchronization APIs. This is because the pthread library calls rely on stateful kernel features implemented using

`futex` system calls. In scaled-out execution mode, the threads run on nodes with different kernel instances. One way to overcome this challenge is to back-migrate the thread to the *local node* and complete the pthread library call on the *local node's* kernel.

7.3 Future Work

This section explores the future work that can improve the design and usability of CRIU-RTX.

One future direction can be eliminating the dependency on the Popcorn compiler, which inserts stack map metadata for transforming stack and registers. A runtime algorithm can be developed to transform stack and registers without the stack map metadata. Also, supporting other programming languages like C++ will allow more applications to use the CRIU-RTX framework.

Another direction is reducing false page-sharing by memory access pattern analysis. Identifying the memory access pattern of each application manually is a laborious task. Raptor [46] is a related work in this direction that identifies false page-sharing in unified CPU-GPU memory systems. Jeremiassen [26] et al. developed a compile-time transformation technique to identify false page-sharing and mitigate them by restructuring the shared data.

To identify the performance and energy efficiency of using CRIU-RTX for new hardware other than the setup we used for evaluation, the same set of experiments must be reproduced in the new setup. Although CRIU-RTX can be deployed easily, it is time-consuming to test in diverse sets of machines. An analytical model can be developed where the benefits of CRIU-RTX can be predicted without actual testing. The analytical model can give the anticipated performance of CRIU-RTX on new hardware, which depends on processing power, memory,

network speed, and memory access pattern of target applications.

Bibliography

[1] Processing Architecture for Power Efficiency and Performance . https://www.arm.com/technologies/big-little.

[2] CRIU compel. https://criu.org/Compel.

[3] CRIU - Checkpoint Restore in Userspace. https://criu.org/Main_Page.

[4] JavaScript Object Notation. https://www.json.org/json-en.html.

[5] MSI Protocol. https://en.wikipedia.org/wiki/MSI_protocol.

[6] Page Fault. https://en.wikipedia.org/wiki/Page_fault.

[7] The /proc filesystem, . https://www.kernel.org/doc/html/latest/filesystems/proc.html.

[8] Protocol buffers - google's data interchange format, . https://github.com/protocolbuffers/protobuf.

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

[10] On vsyscalls and the vdso. https://lwn.net/Articles/446528/.

[11] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The mit alewife machine: Architecture and performance. *ACM SIGARCH Computer Architecture News*, 23(2):2–13, 1995.

[12] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, et al. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 775–787, 2018.

[13] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. Breaking the boundaries in heterogeneous-isa datacenters. *ACM SIGARCH Computer Architecture News*, 45(1): 645–659, 2017.

[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] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[16] 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*, pages 3–10, 2022.

[17] 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. *Proceedings of the VLDB Endowment*, 11(11): 1604–1617, 2018.

[18] Ho-Ren Chuang, Robert Lyerly, Stefan Lankes, and Binoy Ravindran. Scaling shared memory multiprocessing applications in non-cache-coherent domains. In *Proceedings of the 13th ACM International Systems and Storage Conference*, pages 13–24, 2020.

[19] Eager Consulting. The dwarf debugging standard. https://dwarfstd.org/.

[20] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned global address space languages. *ACM Computing Surveys (CSUR)*, 47(4):1–27, 2015.

[21] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019. URL https://www.flux.utah.edu/paper/duplyakin-atc19.

[22] 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. IEEE, 2020.

[23] Mark S Gordon, D Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, and Xu Chen. COMET: Code offload by migrating execution transparently. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 93–106, 2012.

[24] 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*, pages 417–433, 2022.

[25] Ayal Itzkovitz, Assaf Schuster, and Lea Shalev. Thread migration and its applications

in distributed shared memory systems. *Journal of Systems and Software*, 42(1):71–87, 1998.

[26] Tor E Jeremiassen and Susan J Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. *ACM SIGPLAN Notices*, 30(8):179–188, 1995.

[27] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *OSDI*, volume 16, pages 185–201, 2016.

[28] 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*, pages 3–14, 2015.

[29] 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.

[30] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, et al. The stanford flash multiprocessor. In *Proceedings of the 21ST annual international symposium on Computer architecture*, pages 302–313, 1994.

[31] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The dash prototype: Implementation and performance. *ACM SIGARCH Computer Architecture News*, 20(2):92–103, 1992.

[32] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[33] Liran Liss, Yitzhak Birk, and Assaf Schuster. Efficient exploitation of kernel access to infiniband: a software dsm example. In *11th Symposium on High Performance Interconnects, 2003. Proceedings.*, pages 130–135. IEEE, 2003.

[34] Liran Liss, Yitzhak Birk, and Assaf Schuster. 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):830–840, 2005.

[35] Robert Lyerly, Xiaoguang Wang, and Binoy Ravindran. Dynamic and secure memory transformation in userspace. In *Computer Security–ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part I 25*, pages 237–256. Springer, 2020.

[36] Robert Lyerly, Carlos Bilbao, Changwoo Min, Christopher J Rossbach, and Binoy Ravindran. An openmp runtime for transparent work sharing across cache-incoherent heterogeneous nodes. *ACM Transactions on Computer Systems (TOCS)*, 39(1-4):1–30, 2022.

[37] Robert F Lyerly. *Popcorn linux: A compiler and runtime for state transformation between heterogeneous-ISA architectures*. PhD thesis, Ph.D. dissertation, 2016.[Online]. Available: http://www. ssrg.ece.vt.edu, 2016.

[38] Abhishek Mandar Bapat, Jaidev Shastri, Xiaoguang Wang, Mohamed Husain, Abilesh Sundarasamy, and Binoy Ravindran. Dapper: Lightweight architecture and program state diversification via process rewriting. In *Under Submission*, 2023.

[39] Agostino Mascitti, Tommaso Cucinotta, and Mauro Marinoni. An adaptive, utilization-based approach to schedule real-time tasks for arm big. little architectures. *ACM SIGBED Review*, 17(1):18–23, 2020.

[40] Vijay Nagarajan, Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 15(1): 1–294, 2020.

[41] 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 (USENIXATC 15)*, pages 291–305, 2015.

[42] 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 (USENIXATC 15)*, pages 291–305, 2015.

[43] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[44] Ranjit Noronha and Dhabaleswar K Panda. Can high performance software dsm systems designed with infiniband features benefit from pci-express? In *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, volume 2, pages 945–952. IEEE, 2005.

[45] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel & Distributed Technology: Systems & Applications*, 4(2):63–71, 1996.

[46] Md Erfanul Haque Rafi, Kaylee Williams, and Apan Qasem. Raptor: Mitigating cpu-

gpu false sharing under unified memory systems. In *2022 IEEE 13th International Green and Sustainable Computing Conference (IGSC)*, pages 1–8. IEEE, 2022.

[47] Steven K Reinhardt, James R Larus, and David A Wood. Tempest and typhoon: User-level shared memory. In *Proceedings of the 21st annual international symposium on Computer architecture*, pages 325–336, 1994.

[48] Steven K Reinhardt, Robert W Pfile, and David A Wood. Decoupled hardware support for distributed shared memory. *ACM SIGARCH Computer Architecture News*, 24(2): 34–43, 1996.

[49] Xiaowei Ren and Mieszko Lis. Efficient sequential consistency in gpus via relativistic cache coherence. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 625–636. IEEE, 2017.

[50] Daniel J Scales and Kourosh Gharachorloo. Towards transparent and efficient software distributed shared memory. *ACM SIGOPS Operating Systems Review*, 31(5):157–169, 1997.

[51] Martin Schoeberl, Luca Pezzarossa, and Jens Sparsø. S4noc: a minimalistic network-on-chip for real-time multicores. In *Proceedings of the 12th International Workshop on Network on Chip Architectures*, pages 1–6, 2019.

[52] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 323–337, 2017.

[53] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, 2018.

[54] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 306–324, 2017.

[55] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with in-network cache coherence. In *FAST*, pages 277–292, 2021.

[56] Tong Xing, Antonio Barbalace, Pierre Olivier, Mohamed L Karaoui, Wei Wang, and Binoy Ravindran. H-container: Enabling heterogeneous-isa container migration in edge computing. *ACM Transactions on Computer Systems (TOCS)*, 39(1-4):1–36, 2022.

[57] M. Zwolenski and L. Weatherill. The digital universe: Rich data and the increasing value of the internet of things. *J. Telecommun. Digit. Economy*, 2(3):1–47, 2014.