MEMORY SUBSYSTEMS FOR SECURITY, CONSISTENCY, AND SCALABILITY

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Terry Ching-Hsiang Hsu

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

March 2018

Purdue University

West Lafayette, Indiana

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Hsu, Terry C.-H. PhD, Purdue University, March 2018. Memory Subsystems for Security, Consistency, and Scalability. Major Professors: Patrick Eugster and Mathias Payer.

Memory is the fundamental element of computer systems. Traditional memory subsystems provide basic functionalities for multiple programs to share physical memory using the concept of virtual memory. However, today's computer systems have become too sophisticated for traditional memory subsystems to handle many design requirements. This dissertation advances the design and implementation of existing memory subsystems to provide *security*, *consistency*, and *scalability* guarantees through a software approach.

This dissertation has three main parts. The first part describes a memory subsystem for the security of multi-threaded applications. The second part describes a memory subsystem for the consistency of multi-threaded applications running on non-volatile memory systems. The third part describes a memory subsystem for the scalability of data-intensive applications with isolation and crash recovery support.

This dissertation makes four high-level contributions. First, the concise descriptions show a memory subsystem that provides threads with fine-grained control over privileges for a partially shared address space. Second, the precise descriptions show another memory subsystem that allows programmers to easily leverage non-volatile memory. Third, the detailed descriptions show the third memory subsystem that enables memory-centric applications to freely access memory beyond the traditional process boundary with support for isolation and crash recovery. Finally, this dissertation provides details and experience for designing practical memory subsystems from high-level concepts, design logic, to low-level implementation techniques.

1 INTRODUCTION

This dissertation advances the design and implementation of traditional memory subsystems. Specifically, the proposed memory subsystems address three important issues in modern software: *security*, *consistency*, and *scalability*.

1.1 Background

This dissertation is mainly about achieving three things using memory subsystems: security, the techniques for efficient memory compartmentalization; consistency, the techniques for automatic persistence of consistent program states; and scalability, the techniques for fast access to large amount of memory beyond the traditional process boundary with support for memory isolation and crash recovery.

1.1.1 Security

Coordinating software components with different privilege levels is a key requirement for the stability and security of today's computer systems. However, failing to properly isolate components in the same address space has resulted in a substantial amount of vulnerabilities. Enforcing the least privilege principle for memory accesses can selectively isolate software components to restrict attack surface and prevent unintended cross-component memory corruption. However, the boundaries and interactions between software components are hard to reason about and existing approaches have failed to stop attackers from exploiting vulnerabilities caused by poor isolation.

To address the challenges in selective memory isolation, we present the *secure memory views* (SMV): a practical and efficient and memory system for secure and

selective memory isolation in monolithic multithreaded applications in Chapter 2. SMV offers explicit access control of memory and allows concurrent threads within the same process to partially share or fully isolate their memory space in a controlled and parallel manner following application requirements. An evaluation of our prototype in the Linux kernel (TCB < 1,800 LOC) shows negligible runtime performance overhead in real-world applications including Cherokee web server (<0.69%), Apache httpd web server (<0.93%), and Mozilla Firefox web browser (<1.89%) with at most 12 LOC changes.

1.1.2 Consistency

Non-volatile memory technologies, such as memristor [1] and phase-change memory [2], will allow programs to persist data with regular memory instructions. Liberated from the overhead to serialize and deserialize data to storage devices, programs can aim for high performance and still be crash fault-tolerant. Unfortunately, to leverage non-volatile memory, existing systems require hardware changes or extensive program modifications.

To help programmers to easily leverage non-volatile memory, we present NVthreads, a memory subsystem and runtime that adds persistence to existing multithreaded C/C++ programs in Chapter 3. NVthreads is a drop-in replacement for the pthread library and requires only tens of lines of program changes to leverage non-volatile memory. NVthreads infers consistent states via synchronization points, uses the process memory to buffer uncommitted changes, and logs writes to ensure a program's data is recoverable even after a crash. NVthreads' page level mechanisms result in good performance: applications that use NVthreads can be more than 2× faster than state-of-the-art systems that favor fine-grained tracking of writes. After a failure, iterative applications that use NVthreads gain speedups by resuming execution.

1.1.3 Scalability

Data intensive applications — enabled by emerging non-volatile memory technologies and motivated by modern memory-driven computations — demands scalable memory management for accessing vast amounts of main memory. A large memory pool is only useful if applications can efficiently access and protect data without expensive context switches. No existing framework currently allows applications to decouple memory from the process abstraction with privilege separation, thereby limiting the scalability and practicability of memory-driven applications in the big data era.

In Chapter 4, we present the design and implementation of *PetaMem*: a memory subsystem that enables memory-centric applications to freely access memory beyond the traditional process boundary with support for isolation and crash recovery. PetaMem completely decouples the traditionally fused abstractions of processes and memory from space and time. The novel kernel-level pager and reference monitor in PetaMem organize virtual memory in different address spaces efficiently and securely. In addition, PetaMem provides a crash recovery engine to make applications running on non-volatile memory systems fault tolerant. Our evaluation shows that applications using PetaMem can outperform the traditional process-centric design by two orders of magnitude when accessing memory in multiple address spaces, and the recovery engine provides three orders of magnitude speedup when recovering from crash failures.

1.2 Problem Statement

No practical memory subsystems currently allow applications to (1) achieve selective intra-process isolation with negligible overheads, (2) guarantee program consistent states automatically with low overheads, or (3) access large amount of memory beyond the traditional process boundary with memory isolation and crash recovery support. Current solutions impose significant programming burdens and performance

overheads on applications when isolating, persisting, and accessing large amount of memory.

1.3 Thesis Statement

The thesis of this dissertation is that *we can advance the design and implementation of existing memory subsystems to enhance the security, consistency, and scalability of applications through a software approach.* This thesis will introduce memory subsystems to achieve strong intra-process isolation for security, automatic data persistence for consistency, and fast access to large amount of memory for scalability.

1.4 Contributions

This dissertation makes the following contributions.

First, the design and implementation of our memory subsystem which provides threads with fine-grained control over privileges for a partially shared address space.

Second, the specification of an application programming interface for programmers that facilitates porting existing legacy software for intra-process isolation.

Third, an evaluation of our memory subsystem prototype showing a practical and efficient memory subsystem to achieve intra-process isolation for multithreaded applications.

Fourth, the design and implementation of our memory subsystem and runtime that infers when data structures are consistent and adds durability semantics with very few program modifications.

Fifth, an extensive evaluation of our memory subsystem demonstrating the ease of use for programmers to leverage non-volatile memory in practice.

Sixth, the design and implementation of our memory subsystem that enables applications to efficiently access large amount of memory without requiring any hardware modifications.

Seventh, the specification of an application programming interface for programmers to access large amount of memory in memory-centric computing.

Eighth, an evaluation of our memory subsystem showing the performance improvement over the traditional process-centric architecture when accessing large amount of memory.

Finally, this dissertation provides details and experience for designing practical memory subsystems from high-level concepts, design logic, to low-level implementation techniques.

1.5 Organization

This dissertation has five chapters. Chapter 2 presents our memory subsystem that enforces strict intra-process isolation for security. Chapter 3 describes our memory subsystem that persists consistent program states in non-volatile memory for consistency. Chapter 4 illustrates our memory subsystem that efficiently addresses large amount of memory for scalability. Finally, Chapter 5 concludes the dissertation.

2 MEMORY SUBSYSTEM FOR SECURITY

In this chapter we present a memory subsystem [3] for selective intra-process isolation along with an extensive evaluation of the system prototype. We propose a corresponding application programming interface with concrete application examples.

2.1 Overview

Ideally, software components are separated logically into small fault compartments, so that a defect in one component cannot compromise the others. This concept of *privilege separation* [4, 5] protects confidentiality and integrity of data (and code) that should only be accessible from small trusted components. However, most applications use a *single address space*, shared among all components and threads. Redesigning all legacy multithreaded applications to use processes for isolation is impractical. Today's software, such as web servers and browsers, enhances its functionality through libraries, modules, and plugins that are developed independently by various *third-parties*. Failing to properly separate privileges in applications and confine software components in terms of their memory spaces leaves a system vulnerable to attacks such as privilege escalation, denial-of-service, buffer overflows, and control-flow hijacking, jeopardizing both the stability and the security of the system.

This chapter develops and evaluates a new memory subsystem known as *secure memory views* (SMVs) to enable strict thread isolation for C/C++ multithreaded applications. SMV is a programming abstraction to support selective memory isolation with kernel-level implementation. The extensive evaluation suggests that SMV is robust and practical for large-scale production software.

Many proposals exist for privilege separation in a monolithic application. The first generation privilege separation techniques focus on splitting a process into different

single-process compartments. Provos et al. [6] presented an intra-process privilege separation case study by manually partitioning OpenSSH components into privileged master processes and unprivileged slave processes. Privtrans [7] automated the partitioning procedure. Wedge [8] then introduced capabilities to privilege separation and Salus [9] enabled a dynamic security policy. Unfortunately, all these techniques cannot support multithreaded compartments.

Second generation privilege separation techniques like Arbiter [10] aimed to support multithreaded applications by allowing concurrent thread execution. However, Arbiter's implementation for separating memory space and its serialized user-level memory management impose prohibitive runtime overhead (200% – 400% for memory operations). As a result, the thread execution is not fully concurrent since all threads must wait on a global barrier to tag memory pages for capabilities. In addition, the required retrofitting efforts for legacy software are non-trivial, as the case studies showed that at least hundreds of lines of code (LOC) changes are required to separate software components even for applications that have small code base sizes (8K LOC).

We postulate that a third generation privilege separation technique for achieving intra-process isolation in monolithic multithreaded applications such as the Cherokee web server, Apache httpd, and the Mozilla Firefox web browser, needs to fulfill the following requirements (which are only partially addressed by existing solutions) for wide adoption:

- *Genericity and flexibility* (GF): Implementing privilege separation in different types of applications requires programmers to employ completely different abstractions and concepts. A general model with a universal interface and isolation concept that supports *both* client- and server-side multithreaded applications is needed (e.g., compartments in the Firefox browser, worker buffers in Cherokee web server, worker pools in Apache httpd web server).
- *Ease of use* (EU): Programmers prefer to realize their desired security policy in a model with a high-level API rather than through low-level error-prone memory

management tools without intra-process capabilities (e.g., mmap, shmem.). In particular, porting legacy software to a new model has to be easy despite the complexity of component interweaving and the underlying assumption of shared memory (e.g., Firefox, which contains 13M LOC), and should be possible with minimal code refactoring efforts.

- *No hardware modifications* (NH): Over-privileged multithreaded applications are pervasive. A model that is ready to run on today's commodity hardware (even regardless of the CPU brands/models) is necessary for wide deployment.
- *Low runtime overhead* (LO): Monitoring application memory accesses at high frequency is unrealistic for practical systems. A practical model must be implemented in a way that incurs only negligible runtime overheads. In particular, enhanced security should not sacrifice the parallelism in multithreaded applications. A model has to support selective memory isolation for multiple computing entities (i.e., multiple threads can exercise the same privilege to parallelize a given workload and perform highly parallel memory operations).

To address the above challenges, we propose a third generation privilege separation solution for monolithic applications: *secure memory views* – a model and architecture that efficiently enforces differential security and fault isolation policies in monolithic multithreaded applications at negligible overheads. SMV protects applications from negligent or malicious memory accesses between software components. In short, the intrinsically shared process address space is divided into a dynamic set of *memory protection domains*. A thread container SMV maintains a collection of memory protection domains that define the *memory view* for its associated threads. Access privileges to the memory protection domains are explicitly defined in the SMVs and the associated SMVthreads must strictly follow the defined security policies. The SMV model provides a well-defined interface for programmers to exercise the least privilege principle for arbitrary software objects "inside" a multithreaded process. For example, a server's worker thread can be configured to allow access to

its thread stack and part of the global server configuration but not to the private key that resides within the same process address space.

With the SMV model, the programmer can enforce different access permissions for different components in a single address space (GF). New software can leverage the full API and can be designed to only share data along a well-defined API, and existing software can be retrofitted (with minimal code changes) by instrumenting calls across component boundaries to change the underlying memory view (EU). Moreover, the privilege enforcement relies on OS kernel level page table manipulation and standard hardware virtual memory protection mechanisms (NH). Therefore, the SMV model does not suffer from the performance overheads (LO) imposed by IPC (vs in-memory communication), user level memory management (vs kernel level), or per-instruction reference monitors (vs hardware trap). The SMV model's programmability and efficient privilege enforcement mechanism allow it to protect *both* client- and server-side multithreaded applications with low overhead.

We implemented a prototype of the SMV model in the Linux kernel. Our evaluation demonstrates (a) its negligible runtime overhead in the presence of high concurrency using multithreaded benchmarks that employ the general producer-consumer pattern, and (b) the immediate benefit of efficient software component isolation by compartmentalizing client connections for the popular Cherokee and Apache httpd web servers and the compartments in the Firefox web browser. SMVs incur only around 2% runtime overhead overall with 2 LOC changes for the multithreaded benchmark PARSEC, 0.69% throughput overhead with 2 LOC changes for Cherokee, 0.93% throughput overhead with 2 LOC changes for Apache httpd, and 1.89% runtime overhead with only 12 LOC changes for the Firefox web browser. Note that SMV focuses on restricting memory views for individual threads, access permissions for kernel APIs is an orthogonal problem that is well covered by, e.g., AppArmor [11], SELinux [12], or the seccomp framework [13].

In summary, this chapter makes the following contributions:

	1st gen technique				2nd gen technique	3rd gen technique
Problem tackled	Non-parallel privilege separation				Concurrent execution and dynamic security policy	Concurrent memory operations and high performance
Issue vs solution	**OpenSSH** [6]	**Privtrans** [7]	**Wedge** [8]	**Salus** [9]	**Arbiter** [10]	**SMV** (This work)
Security principal	Process	Process	Single thread	Single thread	Multiple threads	Multiple threads
Parallel execution	Not handled	Not handled	Not handled	Not handled	Partially handled	Yes
Parallel tagging	Not handled	Not handled	Not handled	Not handled	Not handled	Yes
Security policy	Static	Static	Static	Dynamic	Dynamic	Dynamic
TCB	OS	Compiler, OS	OS	Library, OS	Library, OS	OS (< 1800 LOC)
Refactoring efforts	Fully manual	Annotations	Tool assisted	Tool assisted	Fully manual (> 100 Δ LOC)	Library assisted (< 20 Δ LOC)
Use cases	OpenSSH	OpenSSH etc.	OpenSSH etc.	PolarSSL	FUSE (8K LOC) etc.	Firefox (13M LOC) etc.

Table 2.1.: Issues and solutions for intra-process privilege separation techniques.

Design of the SMV model which provides threads with fine-grained control over privileges for a shared address space.

Specification of an SMV API for programmers that facilitates porting existing pthread applications.

Implementation of the SMV model that consists of a trusted Linux kernel component (implementing enforcement) and the corresponding untrusted user-space library that implement the SMV API, which is publicly available along with our benchmarks and test suite[1].

[1] `https://github.com/terry-hsu/smv`

Evaluation of our prototype implementation showing that SMVs achieve all four desired requirements as a practical and efficient model for enforcing least privilege memory views for multithreaded applications in practice.

2.2 Related Work

Techniques for achieving intra-process isolation have been studied for decades. In this section, we summarize and compare the related work following a more detailed breakdown.

Memory safety is the goal of many proposals, as memory corruption is the root cause of various well-known software vulnerabilities. We refer the reader to Nagarakatte et al. [14] and Szekeres et al. [15] for two surveys on memory safety. In short, solutions for complete memory safety do not handle intra-process privilege separation problem and impose significant cost for practical systems (cf. LO).

The first generation privilege separation techniques focus on partitioning a process into single-process components. Provos et al. [6] were the first to manually partition OpenSSH by running components in different processes and coordinating them through inter-process communication (IPC). Privtrans [7] automated the retrofitting procedure for legacy software by partitioning one program into a privileged monitor process and an unprivileged slave process with just few programmer-added annotations. Wedge [8] extended the idea of privilege separation to provide fine-grained privilege separation with static capabilities, which was improved by Dune [16] through the Intel VT-x technology (cf. NH) for better performance and by Salus [9] for dynamic security policy. The disadvantage of these first generation techniques is that they lack support for multiple computing entities within the same compartment (cf. LO). This limitation hurts performance of multithreaded programs and restricts the usability of these solutions in practice.

The second generation privilege separation technique Arbiter [10] allowed multiple threads to run in the same compartment. However, Arbiter still faces similar

limitations on parallel memory operations and their evaluation does not use multi-threaded benchmarks that have intensive memory operations to demonstrate the system's parallelism, even though the design aims at concurrent execution for threads (cf. LO). We identify two major causes of the limitation on Arbiter's parallelism. First, the highly serialized memory management in their user-space library incurs inevitable runtime overhead of up to 400%. Second, the design choice of separating mm_structs forces their kernel to aggressively synchronize the global process address space for every thread's memory descriptors. The synchronization costs increase when an application performs intensive memory operations or generates a huge amount of page faults. These limitations on parallelism manifest themselves when running real-world applications (e.g., Firefox) with large inputs (e.g., web server hosting a 100MB file). However, the largest input in the authors' evaluation is only 1MB. In addition, programmers are on their own to partition applications as the solution does not provide assistance in retrofitting applications (cf. EU). As we will show in this chapter, the SMV model addresses the limitations of the first and second generation privilege separation techniques without sacrificing security or parallelism.

OS-level abstraction mandatory access control solutions such as SELinux [12], AppArmor [11], and Capsicum [17] protect sensitive data at process/thread granularity. However, fine-grained privilege separation for software objects (e.g., arrays) within a process is not supported in these techniques. On the other hand, SMVs tackle issues for intra-process data protection with capabilities. PerspicuOS [18] separates privileges between trusted and untrusted kernel components defined by kernel developers using an additional layer of MMU. Such an intra-kernel design does not facilitate intra-process privilege separation as SMV does for user-space applications (cf. GF and EU). These security policies are orthogonal to the SMV memory policies and SMVs can be used in conjunction with these techniques to gain additional inter-process protection.

Decentralized information flow control (DIFC) systems allow programmers to associate secrecy labels with data and enforce information flow to follow security

policies. HiStar [19] is an OS that fundamentally enforces DIFC that could likely address intra-process isolation. However, HiStar is not based on a general OS kernel such as Linux and thus cannot be incrementally deployed to commodity systems. Moreover, the applications have to be completely rewritten in order to use HiStar (cf. GF and LO). Thus, the solution is infeasible for legacy software (e.g., Firefox) in practice. Flume [20] focuses on process-level DIFC for OS-level abstractions (e.g., files, processes) in UNIX but it does not handle intra-process privilege separation within a multithreaded application. Laminar [21] supports multithreaded application running in its specialized Java virtual machine (cf. GF). However, the additional layer in the software stack and its dynamic checker incur significant runtime overhead (cf. LO). As noted in Section 2.7.9, byte-granularity checkers in DIFC systems incur high performance overhead in practical applications that have intensive memory operations (cf. LO).

Software-based fault isolation (SFI) [22,23] isolates software components within the same address space by constructing distinct fault domains for code and data. SFI prevents code from modifying data or jumping to code outside of a fault domain. Native client [24, 25] utilizes SFI with x86 hardware segmentation for efficient reads, writes, control-flow integrity [26], and component isolation. However, the untrusted code is statically associated with a specific fault domain as the approach does not provide simple means of implementing a dynamic and flexible security policy for practical multithreaded applications (cf. EU and LO). In contrast, SMVs offers solutions for programmers to structure the protected memory regions in a dynamic and non-hierarchical manner.

Language-based techniques utilize safe language semantics to provide isolation for applications written in type-safe languages (e.g. [27, 28]) and implement information flow control for objects within a process (e.g. [29–31]). However, the vast majority of legacy software are still written in an unsafe language for efficiency. As a result, programmers need to completely rewrite their legacy software using safe languages (cf. GF). Ribbons [32] is a programming model developed entirely for user space that

provides fine-grained heap isolation for multithreaded applications. While the access privileges of threads are tracked pair-wise between domains hierarchically in user space in Ribbons, the SMV model leverages the OS memory management subsystem to organize the access privileges of threads systematically in kernel space at negligible overhead (cf. EU and LO).

Special hardware support and virtualization technologies is another line of research that seeks for strong isolation of program secrets. Flicker's [33] significant overhead due to its intensive use of the TPM chip (cf. NH) makes it impractical for performance-critical applications (cf. LO). Although TrustVisor [34] mitigates the overhead by a hypervisor and a software-based TPM chip, the system is impractical for applications that require multiple compartments with different capabilities (cf. GF). Fides [35] points out the limitations in TrustVisor and improves it by supporting more flexible secure modules with a dual VM architecture on top of its special hypervisor. Hypervisors can be used for guest OSs (e.g. SMV OS kernel) on a shared host while SMVs (providing a richer API) directly run on bare metal at full speed (cf. GF and LO). The additional software level in the hypervisor introduces overheads as the VMM intervenes for the guest OSs page tables, causing TLB cache misses. Recent studies [36–38] by Intel indicate that hardware support for secure computing will become available on mainstream X86 environments in the near future. Intel Software Guard Extensions (SGX) is a mechanism to ensure confidentiality and integrity (but not availability) of a trusted unprivileged software module under an untrusted OS with limited trusted hardware. SGX protects one component from possible interaction using an "enclave" enforced by hardware. Although the goal of Intel SGX is similar to SMVs, our pure-software solution allows SMVs to be adopted by any OSs that have MMU subsystems with commodity hardware (cf. NH). Loki's [39] tagged memory architecture, CODOMs' [40] tagged pages, and CHERI's [41] capability registers can isolate modules into separate domains with efficient access protection check logic. But these approaches require hypothetical hardware support which make them incompatible with commodity systems (cf. NH).

2.3 Threat Model

We assume that the attacker, an unprivileged user without root permissions, can control a thread in a vulnerable multithreaded program, allocate memory, and fork more threads up to resource limits on a trusted kernel with sound hardware. The adversary will try to escalate privileges through the attacker-controlled threads or gain control of another thread, e.g., by reading or writing data of another module or executing code of another module. In this model, the adversary may read or write any data that the controlled thread has access to. The adversary may also attempt to bypass protection domains by exploiting race conditions between threads or by leveraging confused deputy attacks, e.g., through the API exported by other threads. We assume that the OS kernel is not compromised (OS kernel security is an orthogonal topic [42]) and user-space libraries installed by root users are trusted. We assume that the access permissions both of the memory views (enforced through SMV) and for the kernel (enforced through AppArmor, SELinux, or seccomp) are set correctly.

2.4 Objectives

The key objective of the SMV model is to efficiently protect memory references of threads to prevent unintentional or malicious accesses to privileged memory areas during the lifetime of a program. Threads may communicate with other threads through mutually shared memory areas set up by the programmer through SMVs. The SMV model restricts the memory boundaries and memory access permissions for each thread. Without SMVs, an untrusted thread (e.g., a compromised worker thread) may access *arbitrary* software objects (e.g., the private key) within its process (e.g., a web server). Existing programs assume a shared memory space for threads and SMVs must therefore validate that all threads follow the memory rules defined by the programmer (cf. Section 2.5.2). Threads that deviate from these memory reference rules are killed by the system.

The SMV model aims to strictly confine the memory access boundaries for multithreaded programs while *preserving all four desired requirements* for intra-process isolation. We argue in Section 2.5 that the SMV model along with the memory access enforcement can constrain threads within the programmer-defined memory boundaries.

2.5 SMV Model Design

The SMV model consists of three abstractions: *memory protection domains*, *secure memory views*, and *SMVthreads*. The SMV model uses user-defined security policies to enforce the threads' privileges in accessing the shared memory space. The flexibility and programmability of the model allows a programmer to specify the protection domains using high-level abstractions while enforcing the security policy at the lowest level of the software stack (page tables) with acceptable runtime overhead.

2.5.1 Memory Protection Domains

We define a *memory protection domain* as a contiguous range of virtual memory. Any memory address can only belong to one memory protection domain. In this way, a large shared memory space such as the heap can be divided into several distinct sets of memory protection domains. For example, a process can create a private memory protection domain that is only accessible by *one* thread, or a *partially* shared memory protection domain such that only threads with explicit privileges can access it. In addition, an in-memory communication domain can be allocated with global access privileges so that all threads can exchange data without relying on expensive IPC. In general, an unprivileged thread cannot tamper with a memory protection domain even if there exists a defect in the code of the thread. We use the term *memory domain* to refer to the memory protection domain in the rest of this chapter.

2.5.2 Secure Memory Views

We define a *secure memory view* (SMV) to be a thread container with a collection of memory domains. The memory blocks covered by a memory view can only be accessed by threads explicitly given permission to run in the corresponding privileged SMV. Therefore, we consider a memory view to be *secure*.

We define three abstract operations for defining the composition of an SMV:

- *Register(SMV,MD)*: registers memory domain *MD* as part of *SMV*'s memory view.
- *Grant(SMV,MD,P)*: grants *SMV* the capability to access memory domain *MD* with access privilege *P*.
- *Revoke(SMV,MD)*: revokes *SMV*'s capabilities to access memory domain *MD*.

We categorize the privileges P of an SMV to access a memory domain into four operations:

- *Read*: An SMV can read from the memory domain.
- *Write*: An SMV can write to the memory domain.
- *Execute*: An SMV can execute in the memory domain.
- *Allocate*: An SMV can allocate/deallocate memory space in the memory domain.

The access privileges to each of the memory domains for an SMV can be different. Two SMVs can reference the same memory domain but the access privileges can differ. The programmer can set up the SMV's privileges to access memory domains in the way needed for the application at hand. For example, multiple threads sharing the same security context can be assigned to the same SMV to parallelize the workload (LO). To minimize an application's attack surface, the programmer can assume the main parent thread to be the master thread of a program. All the permission modifications must be done by the master thread and are immutable by child threads. The SMV model considers any access to a memory domain without proper privileges to be an *SMV invalid* memory access. We implemented the privilege enforcement at the OS kernel level and detail the design in Section 2.6.5.

2.5.3 SMVthread

An SMVthread is a thread that strictly follows the privileges defined by an SMV to access memory domains. SMVthreads run in the memory view defined by an SMV and cannot change to other SMVs. While the popular pthreads have to trust all pthreads running in the same memory space, SMVthreads distrust other SMVthreads by default. SMVthreads – unlike pthreads – must *explicitly* share access to the intrinsically shared memory space with other SMVthreads. We designed SMVthreads to partially share the memory space with other SMVthreads according to the policy specified by the programmer through the API. Section 2.6.3 explains the implementation of the partially shared page tables for SMVthreads. SMVthreads are glibc-compatible, meaning that our SMVthreads can directly invoke the library functions in glibc. SMVthreads can cooperate with pthreads through all the synchronization primitives defined by the pthreads API. For SMV management, privileged SMVthreads have to invoke the SMV API to set up the memory boundaries for least privilege enforcement. pthreads can access the whole process address space. Changing such accesses would hamper the correctness of legacy programs that do not require any memory segregation (backward compatibility). While possible, programmers are advised *against* mixing SMVthreads and pthreads in one process when an application requires isolation as pthreads will have unrestricted access to all memory of the process.

2.5.4 SMV API: User Space Library

We implemented our SMV model as a user-space library that offers an API to support partially shared memory multithreading programming in C and C++. Table 2.2 summarizes the primary SMV API with descriptions of the main functions. For instance, a programmer can use memdom_create to create a memory domain and memdom_alloc to allocate memory blocks that are only accessible by SMVthreads running in the privileged SMVs. Each memory domain and SMV has a unique ID assigned by the SMV model in the system. SMVthreads are integrated with pthreads

Table 2.2.: List of primary SMV API.

SMV API	Description
int smv_main_init (**bool** allow_global)	Initialize the main process to use the SMV model. If allow_global is true, allow child threads to access global memory domains. Otherwise distrust all threads by default.
int memdom_create (**void**)	Creates a new memory domain, initializes the memory region, and returns the kernel-assigned memory domain ID.
int smv_create (**void**)	Creates a new SMV and returns the kernel-assigned smv_id.
pthread_t smvthread_create (**int** smv_id, (**void***)func_ptr, **struct** smv_data* args)	Creates an SMVthread to run in the SMV specified by smv_id and returns a glibc-compatible pthread_t identifier.
void* memdom_alloc (**int** memdom_id, **unsigned long** size)	Allocates a memory block of size bytes in memory domain memdom_id.
void memdom_free (**void*** data)	Deallocates a memory block previously allocated by memdom_alloc.
int memdom_priv_grant (**int** memdom_id, **int** smv_id, **int** privs)	Grants the privileges privs to access memory domain memdom_id for SMV svm_id and returns new privileges.
int memdom_priv_revoke (**int** memdom_id, **int** smv_id, **int** privs)	Revokes the privileges privs to access memory domain memdom_id from SMV svm_id and returns new privileges.
int memdom_kill (**int** memdom_id)	Deletes the memory domain with memdom_id from the process.
int smv_kill (**int** smv_id)	Deletes the SMV with smv_id from the process.

for easier synchronization and every SMVthread thus also has an associated pthread_t identifier. Note that casting an SMVthread to a pthread does not bypass the privilege checks. The SMV interface allows programmers to structure the process memory space into distinct memory domains with different privileges for SMVthreads and to manage the desired security policy. Furthermore, our library provides options for programmers to automatically override related function calls to significantly reduce the porting efforts. For example, pthread_create can be automatically replaced by smvthread_create, which internally allocates a private memory domain for the newly created SMVthread. Similarly, when an SMVthread calls malloc, the library allocates memory in the calling thread's private memory domain.

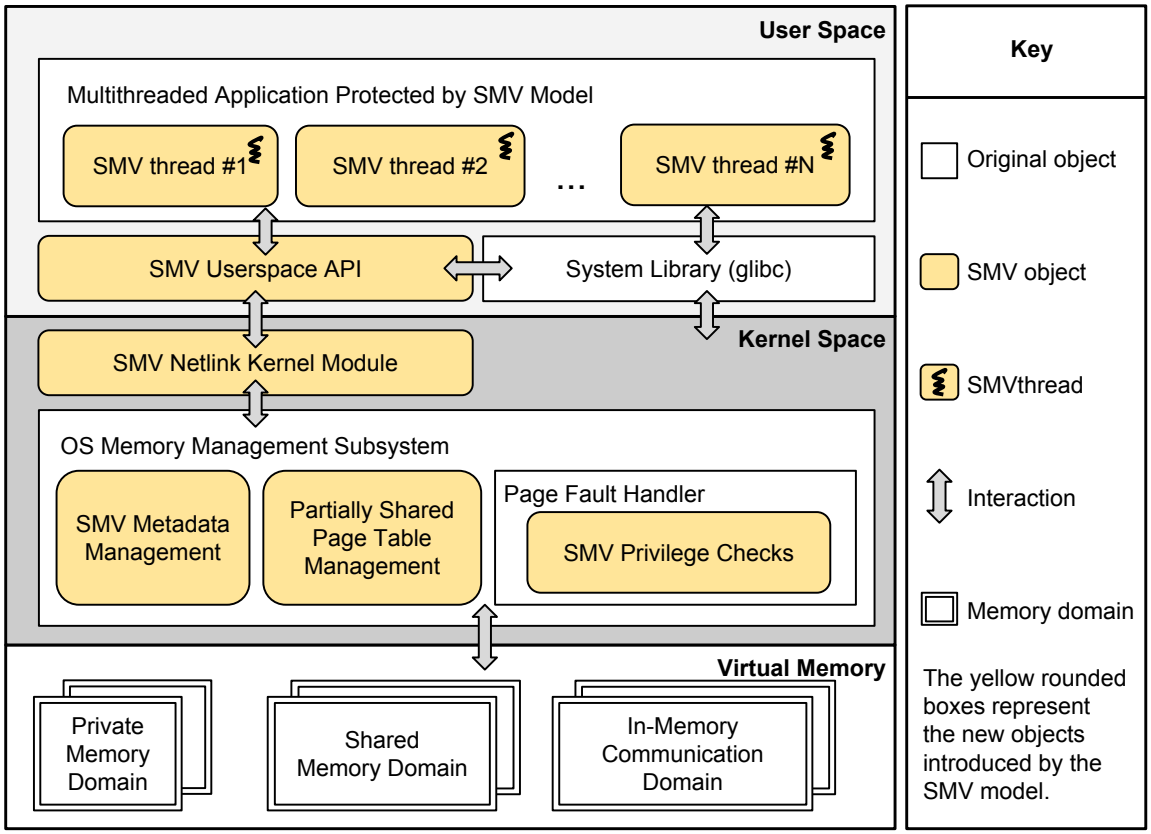

Figure 2.1.: SMV architecture.

2.5.5 SMV Architecture

The SMV architecture consists of two parts: a user space programming interface and a kernel space privilege enforcement mechanism. Figure 2.1 gives an overview. In short, a user space application can call the SMV API to use the SMV model. In the OS kernel, the SMV kernel module is responsible for exchanging the messages between the user space component and the kernel memory management subsystem. We added SMV metadata management to the OS memory management subsystem to record the memory access privileges for the SMVs. We modified the page table management logic to support partially shared page tables and added the SMV privilege checks to the page fault handler that enforces the memory access control.

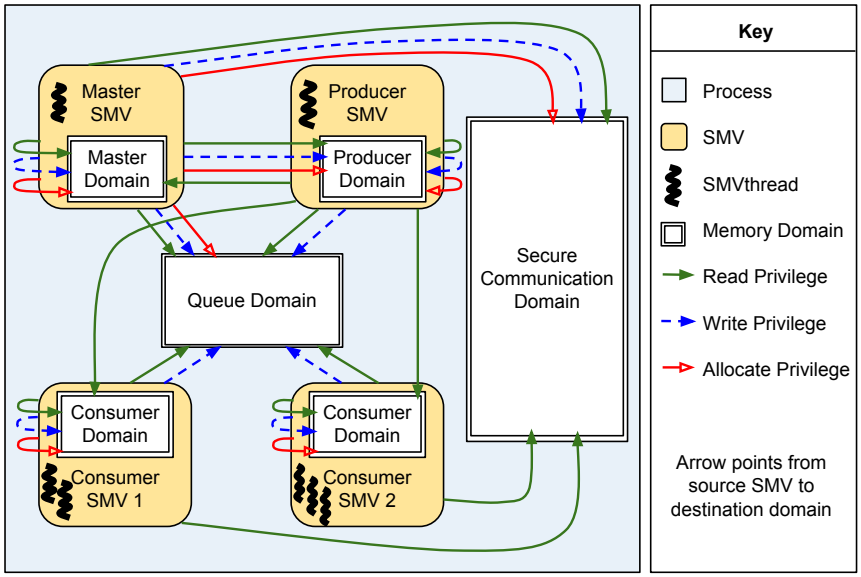

Figure 2.2.: Security-enhanced producer/consumer model with fine-grained memory protection domains.

With the user space interface and the support from the OS kernel, applications can explicitly structure the intrinsically shared process memory space into distinct memory domains with different access privileges without any hardware modifications. Therefore, our approach can be run directly on today's commodity hardware (NH).

2.5.6 Application Examples

The SMV model allows privilege separation of individual components and data regions in an application. We present one example of the popular design model in general multithreaded applications and two concrete application examples of how the SMV model can protect applications by organizing the process address space with different privileges for threads (GF and EU).

Producer-Consumer Model

First, the SMV model can support the common producer-consumer model with strict memory isolation while maintaining efficient data sharing. Figure 2.2 illustrates how the SMV model can secure interacting components according to a *generic* producer-consumer model that is employed by all the applications in PARSEC we evaluated. In this example, the SMVthreads run in the process address space that contains four SMVs and six memory domains. The SMVs confine the memory access privileges of SMVthreads according to the security policy. In this case, the queue domain is the shared memory domain for *all* SMVthreads to cooperate with each other, but any write or allocate request from the producer SMV to the consumer domains is prohibited; only reads are permitted. The secure communication domain works as a one-way communication channel for the master SMVthread to transmit data to the consumer SMVthreads and is inaccessible to the producer SMVthread due to the restricted privileges of the producer SMV. In this case, the SMV model strictly enforces memory access boundaries, constraining memory safety bugs to the current component's memory view.

Case Study: Cherokee Web Server

Cherokee [43] is a high-performance and light-weight multithreaded web server. To isolate connections, Cherokee uses worker threads to handle incoming requests stored in per-thread connection queues. One worker thread handles all the requests coming from the same connection. However, only one worker thread on the server needs to be compromised to leak sensitive information. To provide an alternative for isolating server workers in different processes, we show how the SMV model can compartmentalize the process memory into memory domains and provide reasonable isolation for the multithreaded Cherokee web server. As shown in Figure 2.3, the SMV model defines the memory boundaries for worker SMVthreads and enforces the memory access privileges to protect the server. The SSL connections are handled

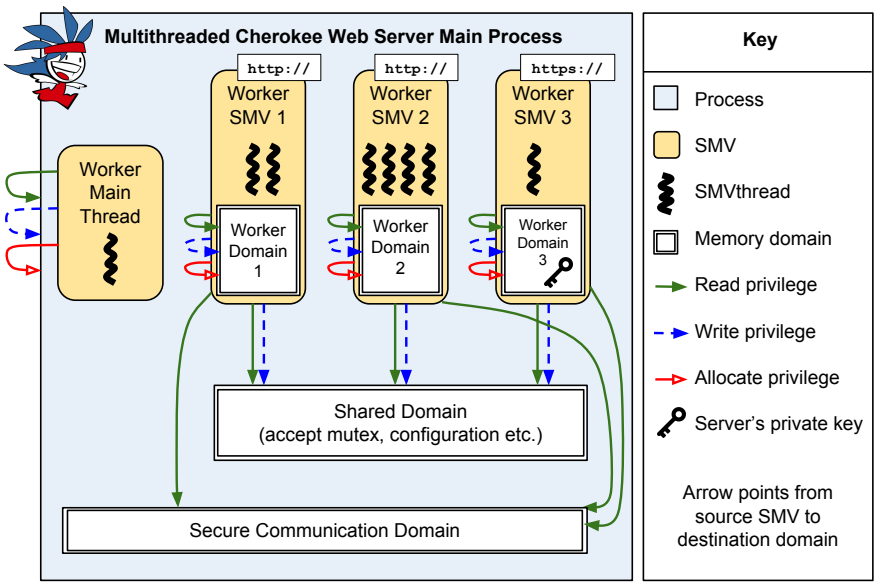

Figure 2.3.: Security-enhanced Cherokee web server.

only by the SMVthreads running in SMV 3 that have the privilege to access worker domain 3, which contains the server's private key. If SMVthreads in SMV 2 (handling only HTTP requests) make any attempt to access the private key, the SMV model will reject such invalid memory accesses because of insufficient privileges. In this way, when an exploited worker thread attempts to access memory in an invalid domain, the SMV model detects such invalid accesses and stops further attacks triggered by the memory bugs (e.g., CVE-2004-1097). The original pthread Cherokee server does not have this security guarantee since all the threads can access the complete process address space (with unanimously shared permission). We show how accessing invalid memory domains is prevented by the SMV model in Section 2.7.5.

Case Study: Mozilla Firefox Web Browser

The SMV model allows multithreaded web browsers such as Firefox and its JavaScript engine SpiderMonkey to achieve strict compartment isolation enforced by hardware protection, preventing one malicious origin from accessing sensitive data such as bank

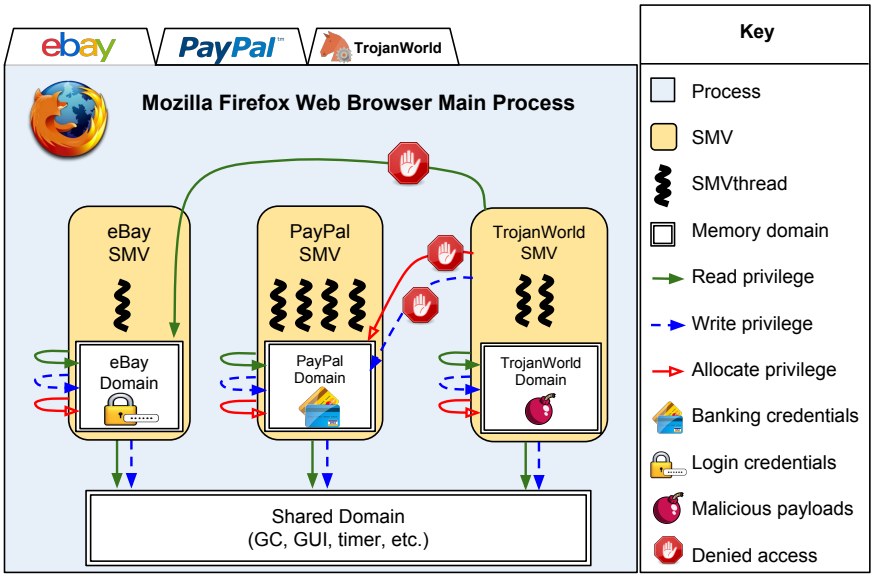

Figure 2.4.: Security-enhanced Firefox.

accounts hosted by another origin. Figure 2.4 presents an example of how the SMV model can isolate browser tabs in SMVs based on the same-origin policy [44]. With SMVs, the malicious origin TrojanWorld cannot escape from its compartment to access the PayPal banking account (add recipient account by allocating memory or transfer money to attacker's account by writing to memory) or read the user credentials hosted by eBay. Such strong isolation guarantees inspired Google to design Chrome to use process isolation for its rendering process.

2.6 Implementation

This section details the OS kernel level implementation of the SMV model and discusses its security guarantees. We modified the Linux kernel version 4.4.5 for the x64 architectures to support the SMV model. Table 2.3 summarizes the component sizes in our prototype.

2.6.1 SMV Communication Channel

We developed an SMV loadable kernel module (LKM) that allows the user-space SMV API to communicate with our kernel using the Netlink socket family. Once loaded, the SMV LKM is effectively part of the kernel. The SMV LKM works as a dispatcher in the SMV model that sanitizes the messages from the user space SMV API and invokes SMV-related kernel functions.

Security guarantee. The attacker cannot replace our SMV LKM with a malicious SMV LKM to perform a man-in-the-middle attack and escalate permissions for a given SMVthread. Such a system-wide change requires the attacker to have root privilege on the system.

2.6.2 Metadata Management

To efficiently maintain the state of the processes that have SMVthreads, we added two major objects to the OS kernel. (1) memdom_struct: memory domain metadata for tracing the virtual memory area and the memory domains mappings. (2) SMV_struct: the SMV privilege metadata for accessing memory domains. These kernel

Table 2.3.: Summary of component sizes in SMV.

	LOC†	**Source files**	**Protection level**
SMV API	781	6	user space
SMV LKM	443	2	kernel
SMV MM‡	1,717	24	kernel

† Lines of code computed by cloc.

‡ SMV MM stands for SMV memory management, which is integrated into the OS memory management subsystem as we show in Figure 2.1.

objects cooperate with each other to maintain the fine-grained privilege information of each SMV in a process.

Security guarantee. The metadata is allocated in kernel memory space and is not mutable by any user space programs without proper privileges through our API. Memory bugs in user space programs cannot affect the integrity of the metadata stored in kernel memory. One of the main sources of kernel 0-day attacks is the use of uninitialized bytes in kernel memory (e.g., CVE-2010-4158) that allows local users to read sensitive information. The SMV model sanitizes the metadata by initializing objects to avoid any potential information leakage from this added attack surface. Our kernel inherits the original kernel's garbage collection system using reference counting to ensure that the additional metadata does not create any dangling pointers.

2.6.3 Partially Shared Memory Space

In the SMV model, SMVthreads can be perceived as *untrusted* tasks by default. Therefore, our kernel has to *partially* separate the kernel objects; it also maintains the consistent process address space for the SMV model. Overall, our kernel: (1) separates the memory space of SMVs by using a page global directory (pgd_t) for each SMV; (2) frees memory for all SMVs when one SMVthread frees the process memory; (3) loads thread-private pgd_t into the CR3 register during a context switch.

All SMVthreads in a process share the *same* mm_struct that describes the process address space. Our kernel allocates one pgd_t for each SMV in a process and stores all pgd_ts in a process's mm_struct. SMVthreads use their private page tables to locate memory pages, yet their permissions to the same page might differ. Note that we designed SMVs to protect thread stacks as well. To ensure the integrity of the process memory space, the page tables of all SMVs need to be updated when the kernel frees the process page tables or when kswapd reclaims page frames. The original kernel avoids reloading page tables during a context switch if two tasks belong to the same process (thus using the same mm_struct). We modified our kernel to reload page

tables and flush all TLB entries if one of the switching threads is an SMVthread. Note that processors equipped with tagged TLBs could mitigate the flushing overhead. However, SMVs do not rely on this hardware optimization feature in order to function correctly (NH).

Based on our extensive experiments, we found that using different mm_structs to separate the address space for threads is overkill and could significantly impact the performance for practical applications (LO). This is because all the memory operations related to mm_struct need to be synchronized in an aggressive manner in order to maintain the consistent process address space for all threads (e.g., rotating the vm_area_struct red-black tree). Using the clone syscall without CLONE_VM flag to isolate a thread's address space from its parent is another approach. However, this approach has two main drawbacks. First, the kernel creates a new mm_struct for the new thread if CLONE_VM is not set. This leads to frequent synchronization and imposes overhead. Second, debugging (e.g., GDB [45]) and tracing memory activity (e.g., Valgrind [46]) become extremely difficult: GDB has to be constantly detached from one process and then attached to another in order to debug a parallel program; Valgrind does not support programs with clone calls. In contrast, using the same mm_struct preserves the system-wide process address space assumption and allows the kernel to separate process address space for threads efficiently.

Security guarantee. The security features of the partially shared memory space rely on the protection guaranteed by the original kernel. The memory management subsystem in the kernel space is completely unknown to user space programs. The attacker has to exploit the permission bits of the page table entries (PTEs) for a thread to break the security features provided by our kernel. We argue that this kind of exploit is highly unlikely without serious DRAM bugs such as rowhammer [47].

2.6.4 Forking SMVthreads

The SMV API uses pthread_create to create a regular pthread and signals the kernel to convert the pthread to an SMVthread before the SMVthread starts execution. The kernel instructs the SMVthread to use the private page tables defined by the SMV that the SMVthread runs in. Once an SMVthread is created, the kernel turns on the using_smv flag stored in the process's mm_struct so that future memory operations must go through additional privilege checks.

To simplify porting efforts, the SMV API provides an option to override all pthread_create calls and automatically allocate private memory domains for each SMVthread.

Security guarantee. The mm_struct of a thread is allocated in kernel space and used solely by the kernel. There are no interfaces that allow user space programs to directly or indirectly modify the memory descriptor. This strong isolation between user and kernel space is guaranteed by the trusted OS kernel. In addition, the atomic fork procedure ensures that the attacker cannot intercept the fork procedure and steal the memory descriptor for the malicious thread.

2.6.5 Page Fault Handler

Figure 2.5 shows the flow chart of the page fault handler in our kernel. The additional checks are surrounded by the gray box with a dotted line. Our kernel kills the SMVthread that triggers an SMV invalid (cf. Section 2.5.2) page fault by sending a segmentation fault signal. For the privileged SMVthreads, our kernel performs *SMV demand paging* to efficiently handle the page faults.

Indeed, since the SMVs use private page tables to separate SMVs' memory views, using the original demand paging routine for SMVthreads is insufficient as the page fault handler only updates the page tables for the current SMV, which causes inconsistent process address space. To solve this problem, our kernel tracks all the faulted pages of a process in the *SMV shadow page tables*. The page fault handler deals with

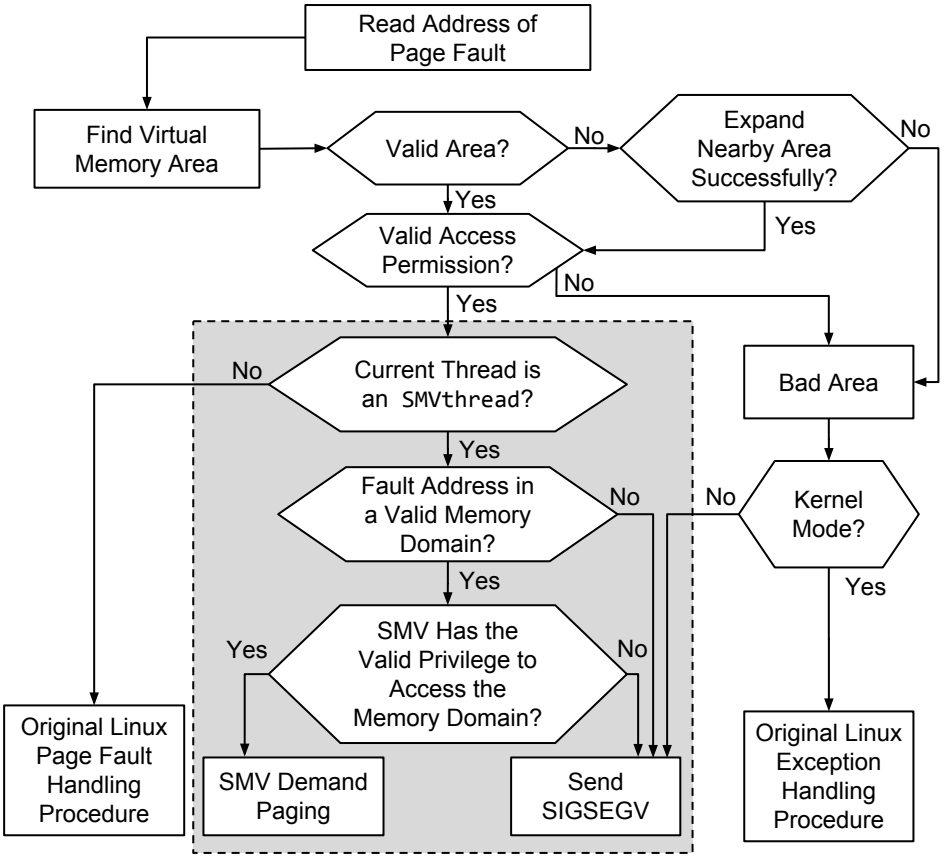

Figure 2.5.: Page fault handler flow chart. The SMV kernel performs additional privilege checks (marked in the gray box).

faults by using the SMV shadow page tables and then copies the page table entry of the fault from the shadow page tables to the running SMVthread's page tables. Note that one process has only one set of shadow page tables, which only serve as quick reference with no permission implications when SMVthreads locate a memory page.

Security guarantee. The page fault handler cannot be accessed, changed, or abused by the attacker as it resides in the lowest level of the software stack. The PTE bits force invalid memory accesses to be trapped to the kernel for the additional privilege checks. To access a privileged memory region, the attacker must first get around the page fault handler. However, such a scenario is infeasible because the

kernel memory management subsystem must intervene and prepare the data page before the attacker can access the privileged memory region.

2.7 Evaluation

The goal of our evaluation is to demonstrate that *the SMV model has all four desired requirements* when enforcing least privilege memory views for multithreaded applications in practice. We show that the SMV model supports different types of multithreaded programs with flexible policies (GF), requires minimal code changes for legacy software (EU), requires no hardware modifications (NH), and incurs negligible runtime overheads while supporting complex thread interactions and extremely intensive memory allocation/free calls in parallel (LO).

2.7.1 Experiment Setup

We measured the performance of our SMV model on a system with Intel i7-4790 CPU with 4 cores clocked at 2.8GHz and 16GB of RAM for our modified x86 64-bit Linux kernel 4.4.5 Ubuntu 14.04.2 SMP (NH). The benchmarks are compiled into two versions: pthread and SMVthread.

2.7.2 Example Policy

SMVthreads cannot access privileged memory domains without being explicitly granted the proper privilege. To test this security guarantee in all of our experiments, the number of domains was set to $N+1$, where N is the number of worker threads and the additional domain serves as a global pool for threads to securely share data. Each worker has its own private memory domain that can only be accessed by itself. We do not claim that the proposed policy is optimal but instead focus on the mechanics to enforce the policy. Setting up alternative policies is possible (GF).

2.7.3 Robustness Test

To examine the robustness, we tested our modified Linux kernel with the Linux Test Project (LTP) [48] developed and maintained by IBM, Cisco, Fujitsu, SUSE Red Hat, Oracle and others. Specifically, we used the runltp script in the LTP package to test the memory management, filesystem, disk I/O, scheduler, and IPC. All stress tests completed without error. We did not observe any system crashes.

2.7.4 Inspecting Isolation

The SMV model treats invalid memory accesses as segmentation faults. Suppose an attacker's thread triggers a segmentation fault by accessing an invalid memory domain on purpose. The main process will crash to prevent further information leakage. Our SMV library provides detailed memory logs to the programmer. Listing 2.1 shows an example of the memory activity log. For crashes due to wrong isolation setup, the logs can help the programmer immediately identify the SMVthread that accessed the invalid protection domain and subsequently rectify the object compartmentalization. In addition, our library provides detailed stack traces for debugging. The logs and stack traces are unreadable by the attacker when debugging mode is disabled. A binary compiled without debugging option makes it impossible for an attacker to learn about memory activity.

2.7.5 Security Evaluation

To further understand how the SMV model offers strong intra-process isolation, we systematically discuss the security guarantees described in Section 2.6.

Trusted computing base. The TCB of the SMV model contains the SMV LKM and SMV MM with kernel level protection (cf. Table 2.3). The SMV API is untrusted and resides in user space as system library. The attacker may try to perform an SMV API call with a malicious intent to escalate permissions for an

SMVthread. The SMV LKM sanitizes all user space messages sent into the kernel and verifies that the SMVthread executing the API call has the correct permissions for the requested change. The attacker may attempt to leverage the misuse of the SMV API to invalidate the memory isolation guarantee provided by the SMV model. Therefore, the security of the application relies on the correctness of the memory isolation setup. Once the memory boundaries are defined, all SMVthreads must follow the memory access rules defined by the programmer. Note that unprivileged users without root permission cannot compromise the SMV LKM (cf. Section 2.6.1 security guarantee).

The SMV model also relies on the privilege level enforcement imposed by the original Linux kernel to make sure that the attacker cannot tamper with the SMV model operating in the kernel space. To bypass the kernel protection, the attacker must hijack the page tables of a privileged thread or modify the metadata stored in the kernel space. The original Linux kernel ensures the integrity of the metadata and memory descriptors for all threads in the system. Using wrong page tables or metadata will cause a thread to be killed once the kernel detects the tainted kernel data structures. Thus, it is impossible for the attacker to exploit the metadata of any thread without kernel 0-day vulnerabilities (cf. Section 2.6.2 security guarantee).

In addition to the software TCB, the SMV model also relies on the hardware's correctness. The hardware vendors perform significant correctness validation. We believe that the security features offered by sound hardware are unlikely for the attacker to subvert (cf. Section 2.6.3 security guarantee). Given the extremely small source code base (less than 2000 LOC), we believe that the SMV's TCB could be formally verified.

TOCTTOU attack: stealing page tables. The attacker may attempt to steal the page tables of a privileged thread by hijacking its memory descriptor. We consider an oracle attacker who knows precisely when and how to launch a time of check to time of use (TOCTTOU) attack to steal the page tables of a privileged thread. If the attack succeeds, the attacker's malicious thread will use the hijacked page tables and read sensitive data in the privileged memory domain before the

Listing 2.1: Kernel log obtained by dmesg command.

```
1 [smv]Created memdom 2, start addr: 0x00f0f000, end addr: 0x00f10000
2 [smv]SMVthread pid 11157 attempt to access addr 0x00f0f0e0 in memdom 2
3 [smv]Addr 0x00f0f0e0 is protected by memdom 2
4 [smv]Read permission granted to SMVthread pid 11157 in SMV 2
5 [smv]SMVthread pid 11155 attempt to access addr 0x00f0f260 in memdom 2
6 [smv]SMV 1 is not in memdom 2
7 [smv]Detected INVALID memory reference to: 0x00f0f260
8 [smv]INVALID memory request issued by SMVthread pid 11155 in SMV 1
9 [smv]<6>chorekee[11155]: segfault at f0f260 ip 00007f09ba7d6656 error 4
```

thread crashes. Assume the attacker can fork threads up to the system limit with the objective to hijack the page tables of an about-to-run privileged thread in the fork procedure, which is the only point for the attacker to exploit the pgd_t pointer. However, the malicious thread has to wait until the privileged SMVthread finishes the page tables setup in order to request the kernel to prepare its unprivileged page tables. Therefore, the attacker cannot intercept the fork procedure and steal the page tables. We conducted an experiment where 1,023 malicious SMVthreads tried to hijack the page tables of a privilege SMVthread. During the one million runs of the security test, every SMVthread used the correct page tables for its memory view (cf. Section 2.6.4 security guarantee).

Effectiveness of the SMV model. Listing 2.1 shows the kernel log when an invalid memory access is detected by the SMV model. In this example, the unprivileged SMVthread pid 11155 in SMV 1 tries to access memory in the privilege memory domain that stores the server's private key, which is only accessible by SMVthread pid 11157 in SMV 2. At line 5, the attempt to read the invalid memory domain triggers the page fault. The kernel rejects the invalid memory request by sending a segmentation fault signal to the unprivileged SMVthread pid 11155 at line 9, stopping the unprivileged SMVthread from accessing the server's private key. The privilege checks

cannot be bypassed because the reference monitor is implemented entirely in the page fault handler, and arbitrary page table manipulation is beyond the attacker's scope (cf. Section 2.6.5 security guarantee).

2.7.6 PARSEC 3.0 Benchmarks

Overview. The multithreaded PARSEC benchmarks include several emerging workloads with non-trivial thread interaction. Both data-parallel and pipeline parallelization models are covered in the benchmarks with coarse to fine granularity. We used all benchmarks in [49], covering all application domains that were originally multithreaded using the standard pthreads. The evaluated benchmarks all employ the producer-consumer pattern (cf. Section 2.5.6) that is pervasive in systems programs and parallelization models. We used the *parsecmgnt* tool in the PARSEC package to run the benchmarks with minimum number of threads set to four for the large inputs as defined by the benchmarks.

Assessment of porting effort. We ported the PARSEC benchmarks by replacing each pthread with an SMVthread running in its own SMV with a private memory domain. In each program, the main program allocates a shared memory domain to store the working set for SMVthreads. The porting procedure consisted of three parts: (1) including the header files to use the SMV API, (2) setting up memory domains and SMVs in the main program, and (3) replacing the pthreads with SMVthreads. All these changes required *only 2 LOC* changes as the SMV API eliminates the refactoring burden (EU). We needed to add only 1 line to include the header file and another to initialize the main process to use the SMV model. The SMV API automatically intercepts pthread_create and malloc and replaces them with smvthread_create and memdom_alloc calls. Therefore, each SMVthread could automatically allocate memory in its private memory domain (cf. Section 2.5.4).

The security-enhanced PARSEC benchmarks demonstrate a general case that could be applied to any multithreaded programs written in C/C++ (GF), even with

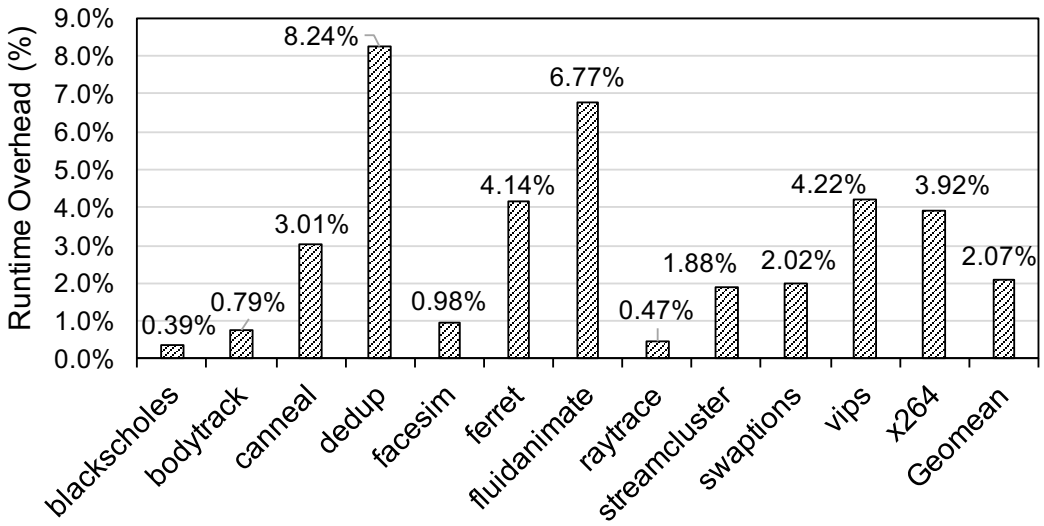

Figure 2.6.: Runtime overhead of the SMV model for the multithreaded applications in the PARSEC benchmark suite.

the presence of extremely intensive memory allocation/free calls in parallel. In addition, the intra-process isolation can help prevent attacks that arbitrarily modify data on the stack using malicious threads, e.g., ROP-based attacks. One study has shown that an attacker can perform ROP and use gadgets (16 payloads is enough for >80% of GNU coreutils [50]) to achieve Turing completeness. Note that ASLR/stack canaries have been proven ineffective to protect against information leakage [51]. With SMV, programmers can secure the system with few changed lines of source code while also handling nontrivial thread interaction, if needed.

Performance. Figure 2.6 shows the runtime overhead of the ported PARSEC benchmarks with 10 runs for each program. The results show that the SMV model incurs negligible runtime overhead. The overall geometric mean of the runtime overhead is only 2.07% (LO) and the maximum of the runtime overhead occurs for dedup due to the huge amount of page faults and the highly intensive parallel memory operations.

2.7.7 Cherokee Web Server

Overview. The original Cherokee server uses a per-thread memory buffer system for resource management to isolate threads from remote connections. We leveraged the SMV model to provide Cherokee with the OS level privilege enforcement for different server components. Then we compared the throughput of our security-enhanced Cherokee with the original Cherokee.

Assessment of porting effort. We enhanced the security of the Cherokee version 1.2.104 server as illustrated in Section 2.5.6. The user-space SMV library automatically replaced pthread_create with smvthread_create to create SMVthreads for the workers to handle client requests. Each SMVthread worker ran in its own SMV with a private memory domain which is inaccessible by other workers. All other shared objects such as the mutex are allocated in a shared memory domain and accessible by all workers. We modified *only 2 LOC* of Cherokee to enforce the least privilege memory access with the SMV model (EU). We believe that the negligible porting effort demonstrates the practicability of the SMV model to protect real-world applications.

Performance. We used ApacheBench to measure the server throughput for the original and security-enhanced Cherokee. Both versions of Cherokee hosted two kinds of web content: (a) social networking web pages, and (b) large streaming files. Based on the total transfer size per page reported in Alexa top one million websites [52], we tested web page sizes from moderate amount of content to abundant media objects (100KB to 8MB). We also evaluated the performance of both servers hosting large streaming files (50MB and 100MB) to show the practicability of our security-enhanced Cherokee. The Cherokee server process created 40 worker threads by default to handle client requests. The client initiated the ApacheBench for 100,000 requests with concurrency level set to four (matches the number of cores). We conducted the experiment 20 times for each object size and present the results in Figure 2.7. Overall,

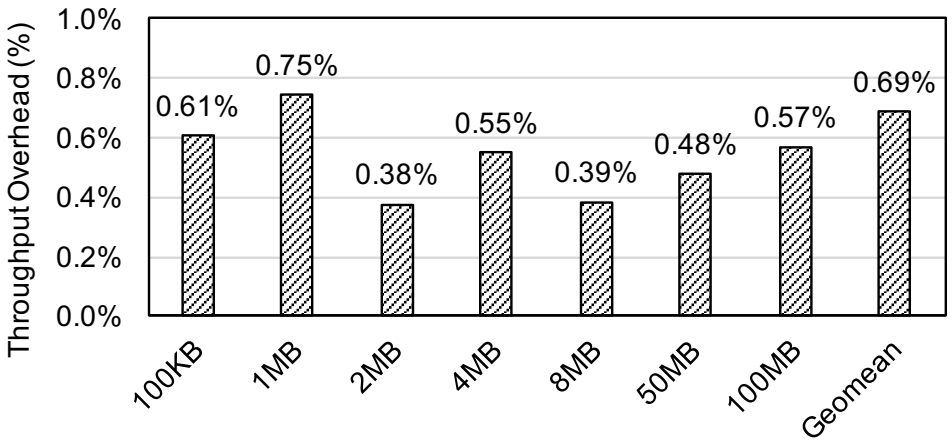

Figure 2.7.: Throughput overhead of Cherokee server.

the SMV model reduced throughput by only 0.69% in exchange for strictly enforcing a least privilege security policy (LO).

We also ported the popular Apache httpd-2.4.18 with only 2 LOC (EU). Using Apache as a file sharing server (GF) to host large objects with size of 10MB, 50MB, 100MB, and 1GB we conducted the same experiment. Overall, SMVs reduced the throughput of the httpd server by only 0.93% (LO). As Cherokee already presents the case for web servers, we exclude the details for Apache httpd due to space limitation.

2.7.8 Mozilla Firefox Web Browser

Overview. The developers of modern web browsers have made tremendous efforts to ensure resource isolation. In 2011, Firefox introduced an abstraction called "compartments" for its JavaScript engine SpiderMonkey to manage JavaScript heaps with security in mind [53]. However, the isolation is not enforced by any mechanism stronger than the compartments' logical boundaries. As a result, any memory corruption can still lead to serious attacks. Here we demonstrate that the SMV model can be easily deployed to protect Firefox's JavaScript engine from memory corruption by confining each compartment to access only its private and the system compartments.

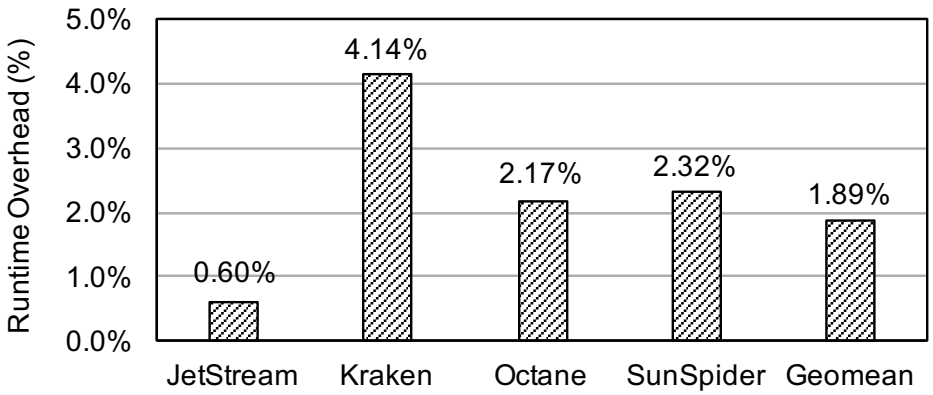

Figure 2.8.: Runtime overhead of security-enhanced Mozilla Firefox web browser.

Assessment of porting effort. Firefox uses threads for UI rendering, processing network packets, monitoring browser status, handling JavaScript jobs, etc. In our evaluation, we replaced Firefox 45.0 SpiderMonkey's NSPR (Netscape Portable Runtime) threads with SMVthreads running in a private memory domain by adding a new thread type named PR_SMV_THREAD to the NSPR library. SpiderMonkey creates 8 threads (1 thread per core + 4 excess threads) in total. We modified *only 12 LOC* of the entire Firefox source to use the SMV model. Although we designed a per-tab isolation policy, the workloads of individual JavaScript benchmark suites are run in the same tab. For example, JetStream executes 40 benchmark programs in the same tab. The performance numbers faithfully report the overhead for privilege checks as each memory page reference is monitored by our page fault handler.

Performance. We evaluated our security-enhanced Firefox with four popular JavaScript benchmarks and report the numbers in Figure 2.8. The overall geometric mean for all benchmarks is only 1.89%. The performance numbers report the overhead when Firefox performs additional privilege checks for SpiderMonkey's helper threads. We believe that such negligible overhead numbers allow efficient and strong isolation for multithreaded browsers to be used in practice and provide the Mozilla team an alternative to the ongoing multi-process Firefox *e10s* project [54].

2.7.9 Limitations

The low performance overhead for checking privileges in the SMV model builds on the virtual memory protection at page granularity. At this point, the SMV model does not guard against unprivileged memory references within the same page as the kernel relies on page table entry (PTE) permission bits. However, poorly organized data structures mixing privileged and non-privileged data within a region are intrinsically insecure and avoided by real-world software (e.g., Hoard [55] memory allocator, connection buffers in Cherokee, worker pools in Apache httpd, compartments in Firefox). Therefore, SMVs can be seamlessly integrated into modern software, eliminating the chances for threads to unintentionally access the same page while enforcing memory boundaries at kernel level. Software monitors for byte-granularity protection has inevitably high overhead (e.g., decentralized information flow control systems) since the memory boundary is neither supported by hardware nor the kernel subsystem, making every memory load/store instruction a candidate for a privilege check. In contrast, page-granularity offers strong memory isolation and superior performance with hardware/kernel support.

Although SMVs cannot protect against malicious library threads once they are installed on the system (requires root privilege, which is out of scope), a user can compile any third-party threading libraries to use SMVs, as we demonstrated in the PARSEC benchmark with GThread in vips and RTThread in raytrace.

2.8 Conclusion

In this chapter we have presented the design, implementation, and evaluation of SMVs, a memory subsystem that allows efficient memory compartmentalization across concurrent threads. SMVs yield a comprehensive architecture with all four desired requirements – *genericity and flexibility*, *ease of use*, *no hardware modifications*, and *low runtime overhead* – for efficient fine-grained intra-process memory separation in multithreaded applications. Our performance evaluation demonstrates that

the SMV model imposes negligible overhead in exchange for greatly improved security guarantees, enforcing intra-process isolation for concurrent threads. The runtime overhead of the multithreaded benchmark PARSEC for using the SMV model is only 2.07% overall with only 2 LOC changes. For popular web servers, the reduction in throughput is only 0.69% overall for Cherokee and 0.93% overall for Apache httpd. Both applications required only 2 LOC changes. We also showed that the real-world web browser Firefox can be easily ported to the SMV model with only 1.89% runtime overhead overall, requiring only 12 LOC modifications to the large code base (13M LOC). The simplicity of the porting effort allows legacy software to be quickly adapted to the SMV model. In summary, we believe that the SMV model can greatly reduce the vulnerabilities caused by improper software component isolation and encourages more research on the efficient and practical intra-process isolation for general multithreaded applications.

3 MEMORY SUBSYSTEM FOR CONSISTENCY

This chapter presents a memory subsystem [56] for persisting program memory for existing multithreaded C/C++ programs. Our extensive evaluation shows that our memory system yields good performance with strong consistency guarantees.

3.1 Overview

Memristor [1], phase-change memory [2], and other emerging non-volatile memory (NVM) technologies will provide disk-like persistence but at latency as low as main-memory (DRAM) devices [57]. These NVM devices will be accessible by memory instructions, and will result in high performance, fault tolerant programs that avoid the overheads of traditional persistent media, such as deep software layers and the cost of serializing and storing data. Programs may even eliminate the distinction between in-memory versus on-disk representations of data. Recent partnership announcements from Intel-Micron [57] and HPE-SanDisk [58] aim to bring this memory-centric computing to consumers. Unfortunately, to fully benefit from low latency persistence, developers need to re-architect both system and application software [59].

When manipulating persistent data directly, applications need to ensure that failures during updates do not end up corrupting data. As an example, a failure during insertion of an element to a persistent linked list should not result in dangling pointers or other corruptions. A safe way to manipulate persistent data is to ensure that data structure updates are failure atomic, i.e., even in the presence of failures, either all or none of the updates are reflected in the NVM. The challenge in implementing failure atomicity is to correctly handle partial updates even in the presence of multithreading, volatile caches, and reordering of NVM writes by the processor. Managing persistent data structures is costly due to these challenges (e.g., frequently flushing

cache lines to NVM is very costly [60]). For any persistent programming system to be practical, its overheads should be low enough that applications actually benefit from using NVM.

Recently proposed frameworks provide multiple ways to directly manipulate NVM data structures [61–63]. Application developers can either rewrite their program to use durable transactions (NV-Heaps [62], Mnemosyne [63]) or rely on the compiler and runtime to infer failure atomic regions from locks (Atlas [61]). These systems track persistent data at a very fine-granularity, such as at the level of individual stores, and use cache flushes and write-ahead logging to correctly recover from failures. Unfortunately, the high overheads of tracking, logging, and managing volatile caches in these systems results in a huge performance gap, sometimes an order of magnitude slowdown, between unmodified DRAM based applications and their crash tolerant versions (Section 3.6.5). Certain systems propose processor modifications to ameliorate the cost of cache flushes and ordering of NVM writes but these systems do not work on today's processors [62, 64].

Our goal is to provide a simple transition path for existing C/C++ programs to leverage non-volatile memory. We want applications to use NVM with few or no program modifications, and yet have good performance on today's processors. Our key observation is that, for many applications, the high overheads of maintaining logs can be reduced substantially by using redo logs in combination with coarse-grained tracking, such as at the level of memory pages.

We propose NVthreads, a memory subsystem we implemented as a threading library that adds durability guarantees to existing multithreaded C/C++ programs. NVthreads uses two techniques to provide failure atomicity. First, NVthreads executes a multithreaded program as a multi-process program, using virtual memory to buffer intermediate changes when data structures may be inconsistent. When program data is in a consistent state, NVthreads commits modified memory pages to a durable log for recovery. By using the operating system's copy-on-write mechanism, NVthreads can efficiently buffer uncommitted writes (unlike the costly software

transactional memory approach in Mnemosyne [63] or eager cache flushes in Atlas' undo logs [61]), and requires only a redo log to recover. Second, it builds on the observation that synchronization operations, such as lock acquire and release, provide enough information to determine the boundaries of failure atomic regions [61]. Instead of requiring programs to be re-written with durable transactions, NVthreads adds durability semantics by automatically inferring when data is safe to write to persistent memory. While NVthreads' design can also be used to implement durable transactions, our current approach of using locks to infer consistency boundaries means that programs require very few modifications to start using NVM.

NVthreads uses multiple techniques to ensure good performance. Its approach of using virtual memory to track data structure modifications is in stark contrast to recent systems that track durable data at the level of individual words [63] or stores [61]. NVthreads reduces the overheads of ordering writes to NVM by eliminating the need to flush data after each program write, and requiring that only log entries be ordered. Even though NVM will be byte-addressable, our evaluation shows the importance of *coarse-grained tracking* of program writes, i.e., at the level of 4KB memory pages, for good performance. In fact, NVthreads is 2×–10× faster than Mnemosyne [63] and more than 2× faster than Atlas [61], both of which use fine-grained memory management. Additionally, NVthreads uses less space to store metadata and logs compared to these systems.

For many workloads in the PARSEC [65] and Phoenix [66] benchmarks NVthreads incurs modest overheads while making data structures durable. We use an emulator to show that NVthreads' performance results are robust even if NVM devices are much slower than DRAM. By using NVthreads and NVM, the persistent versions of programs are 15% to 22× faster than using solid disk drives to store logs. Our evaluation on a K-means clustering program shows that NVthreads helps programs converge up to 1.9× faster after a failure versus their non-durable counterparts. Finally, we integrate NVthreads with Tokyo Cabinet, a high performance key-value store, and show

that for a modest increase in overheads it provides the same durability guarantees and without the need for custom transactional code.

The contributions of this chapter are:

- A memory subsystem that infers when data structures are consistent and adds durability semantics with very few program modifications.

- Novel mechanisms that use process memory to buffer uncommitted writes and track the data at memory page level, thus avoiding undo logs and the need to instrument each program write.

- Extensive evaluation that shows that the NVthreads' approach has low overheads, and outperforms Mnemosyne and Atlas. Iterative applications require only tens of lines of recovery code and converge faster by resuming after failures.

3.2 Challenges in Using Non-volatile Memory

Non-volatile memory (NVM) devices retain data even after a power loss, yet have access characteristics similar to DRAM and higher density than DRAM. In this subsection we review basic characteristics of NVM technologies and the challenges faced by application developers.

3.2.1 Non-volatile Memory

New NVM technologies such as PCM [2], memristor [1], and STT-MRAM [67], will have access latencies similar to DRAM, which is three orders of magnitude faster than flash. Unlike flash and disks, these NVM devices will also be byte addressable, meaning they can be accessed through memory instructions, rather than requiring block-granularity read and write operations. Given these advantages, it is conceivable that NVM devices will not only be deployed as PCIe-attached devices (replacement of SSDs) but also be directly attached to the memory bus (similar to DRAM). However, even in such architectures we expect CPU caches to be volatile, and DRAM and NVM

to coexist. While current applications will continue to work on NVM devices, they will not automatically take advantage of the low latency durability.

3.2.2 Design Issues

There are multiple challenges that need to be solved before programs can use NVM for fault tolerance.

Data structure consistency. NVM-aware solutions need to ensure data consistency even in the presence of failures. Consider Figure 3.1 where a multithreaded program adds an element to a doubly linked list. If a crash occurs during insertion, it is possible that the new element's back pointer is not set correctly, thus leaving the list inconsistent. As memory is persistent, after the crash, the program will not be able to proceed correctly.

Volatile caches. Even though programs will be able to directly access NVM, the underlying hardware may cache data in the volatile CPU caches, and reorder stores to NVM. Unfortunately, there is no easy and efficient way to determine what data has been persisted. In Figure 3.1, even if a crash occurred after the program executed line 10, the updates may not have reached the NVM, and the list could be inconsistent. Therefore, NVM-aware systems need to manage the movement of data from volatile caches to NVM.

Performance and programmability. Since objects in the heap will reside in the NVM, programs no longer need to bear the overheads of serializing and storing persistent data in filesystems. Still, there are performance and programmability costs associated with keeping heap objects consistent. For example, a possible solution to correctly handle state in volatile caches is to flush cache lines after each write, but such heavy-handed approaches result in high overheads. Some prior solutions mitigate these high overheads by assuming the presence of modified processors [62, 64], while others require extensive program modifications.

```
1  // L is a persistent list
2  readFromNVM(&L);
3  ...
4  // Add element to the tail of list
5  pthread_lock(&m);
6  e = nvmalloc(sizeof(elem), 'e');
7  e->val = localVal;
8  tail->next = e;
9  e->prev = tail;
10 tail = e;
11 pthread_unlock(&m)
```

Figure 3.1.: Pseudo-code that appends to a persistent list.

3.3 Related Work

Persistent programming models. BPFS [64] and PMFS [68] are example filesystems that leverage the low latency of NVM to accelerate filesystem operations. BPFS uses optimized shadow copying to maintain consistency, but requires new hardware primitives in the form of epoch barriers. These systems do not require any application changes, but restrict applications to the block based file system interface.

Mnemosyne [63] and NV-Heaps [62] expose direct NVM access but require applications to be rewritten with transactions. NV-Heaps relies on processor changes and maintains undo logs. NVthreads has similar goals as these systems, but uses multi-process execution, and does not require applications to use transactions. Mnemosyne extends software transactional memory with durability semantics and, similar to NVthreads, uses redo logs. However, our evaluation shows that NVthreads outperforms Mnemosyne by 2-10× because of its design choices and use of operating system techniques.

Prior work on Java concurrency control has shown how transactional boundaries can be inferred from locks [69]. Atlas extends this idea to add durability semantics

Issue	**Commonly used solutions**	**NVthreads approach**	**NVthreads advantage**
Determine consistency points	Use durable transactions [62, 63]	Infer from synchronization points	Ease of programming
Handle transaction aborts	Use undo logs [61, 72], word level STM [63]	Use process memory, no undo logs	Lower overheads, simpler recovery
Handle in-flight updates	Use redo logs	Use redo logs	None
Track updates	Intercept each store [61] or word [63]	Intercept page writes	Amortized costs, good performance
Volatile caches	Flush writes, some require new hardware [62, 64, 70]	Flush log entries (memory pages)	Amortized costs, no processor changes

Table 3.1.: NVthreads design decisions.

to lock-based programs [61]. Atlas provides a compiler and runtime that instruments writes to NVM and creates an undo log for each store to aid recovery. JUSTDO logging improves performance and log management in Atlas like systems by storing the program counter and resuming execution of critical sections from exactly the same point where a crash occurred [70]. JUSTDO logging assumes caches are persistent, and has severe programming model restrictions such as volatile data cannot be used inside critical sections and compiler optimizations like register promotion have to be disabled. In fact, JUSTDO logging is 2-3× slower than Atlas on systems with volatile caches, which is the environment that NVthreads targets. Although NVthreads and Atlas both infer failure atomic regions from critical sections, NVthreads' approach is very different. NVthreads tracks data modifications at the granularity of virtual memory pages, isolates thread execution via forking processes, and uses redo logs instead of undo logs. Our evaluation shows that Atlas incurs high overheads of flushing data, and NVthreads outperforms Atlas by 2× on many applications. SoftWrAP uses cache-line combine of writes and asynchronous writes to logs to improve application performance [71]. Unlike SoftWrAP, NVthreads automatically infers failure atomic sections, uses operating system techniques to track updates, and is easy to integrate with pthreads based applications.

Table 3.1 summarizes the benefits of NVthreads' design decisions over existing persistent programming systems.

Operating system mechanisms. NVthreads' use of page-level tracking to provide crash tolerance is similar to RVM [73] and Rio-Vista [74]. Unlike RVM, NVthreads does not have the limitation that data needs to fit in memory, nor does it require DRAM to be battery backed as in Rio-Vista. Additionally, RVM and Rio-Vista don't infer consistency semantics from synchronization operations. QuickStore [75] and Texas [76] use virtual memory techniques to provide persistent object stores on disks, but are more appropriate for object oriented programs. DThreads [77] and Determinator [78] are systems that use multi-process execution to isolate threads for deterministic execution. NVthreads uses the DThreads library to manage data modifications in critical sections, but removes the approach of a global token that DThreads uses for deterministic execution, which can result in up to 9× application slowdown. NVthreads uses a per-mutex token mechanism that improves concurrency in some applications. Unlike DThreads, NVthreads also includes mechanisms to track dependence between critical sections, manage redo logs, and recover from crashes.

Whole system persistence advocates the use of residual power supply energy to flush data during a failure [79]. It relies on new hardware to provide the flush on failure feature, and its software for saving data during failure is susceptible to operating system crashes. Other transparent checkpoint-restore mechanisms rely on virtualization, which increases overheads for all applications or doesn't determine when checkpointing should occur so that data structures are consistent even in the presence of multi-threading [80–82].

Databases and transactions. Most databases use ARIES [72] write-ahead logging which was created to handle the performance difference between sequential and random disk accesses. MARS uses editable atomic writes to make NVM-specific choices such as eliminating undo logs [83]. Stasis also uses write-ahead logging and LSN-free pages to build durable data structures [84]. Similar to MARS, NVthreads does not need an undo log but it is because NVthreads buffers updates in process memory.

3.4 Programming Model

NVthreads adds durability semantics to existing multithreaded programs. We assume that both DRAM and NVM devices exist. NVM devices can be accessed by memory instructions as well as traditional filesystem interfaces. We consider processor caches to be volatile. Programs may control ordering of writes to NVM by using a combination of cache flush instructions, such as clflush or the upcoming optimized clflushopt, fences, and the pcommit instruction. We define crashes to be failures that result in the loss of all processor and DRAM state, but do not corrupt NVM state. Crashes can be caused by power failures or fail-stop software faults.

Original applications use locking primitives and multi-thread functionality provided by the pthreads programming model and library. Applications that link to the new NVthreads library become crash tolerant, and may need to incorporate recovery code to resume execution. We assume that programs modify shared persistent data within critical sections that demarcate failure atomic regions. Program data structures may be inconsistent within a critical section, but data structures are always consistent outside critical sections (when no locks are held).

NVthreads works as follows: it (1) uses critical sections to determine failure atomic regions, (2) tracks dependence between failure atomic regions to decide when to make logs permanent, (3) uses redo logs to ensure NVM data is consistent after a crash, and (4) runs the optional application specific recovery code before resuming execution.

Guarantee. NVthreads guarantees that in the event of a crash failure, and after the completion of NVthreads' recovery process, application data structures will be in a consistent state in NVM, i.e., *the program state after recovery is as if the program stopped when no locks were held*, and the implementation guarantees it by providing the appearance of stopping at the exit of a critical section.

3.4.1 Persistent Regions

A persistent region is a segment of memory, such as a memory mapped file, which is backed by non-volatile memory. NVthreads uses persistent regions to store application data durably. Data structures stored in persistent regions survive application crashes due to faults or power loss. Applications can allocate memory from persistent regions via nvmalloc, a persistent memory allocator. When allocating persistent objects, applications can install a handle, such as a variable name, that acts as an entry point to the object. After a crash, the recovery program may use the handle to determine the location of the persistent object and to traverse other reachable objects. For example, the recovery program may start from the head of a list, and traverse it to find elements in the linked list. Data not in persistent regions, such as application data in DRAM, are lost when a program terminates correctly or incorrectly, or if the system crashes. Applications can continue to allocate and access volatile memory using existing interfaces such as malloc.

3.4.2 Inferring Consistent Program Points

A key challenge in adding durability semantics to programs is to determine when data structures are consistent. Instead of using durable transactions and rewriting applications, NVthreads infers failure atomic regions from synchronization primitives. NVthreads' API also supports programmers who explicitly specify commit points, similar to manual checkpointing. However, this work focuses on how NVthreads can automatically infer failure atomic regions, thus making it easier for programmers to add durability semantics to their programs.

Multithreaded applications use synchronization primitives, such as locks, to safely modify shared data structures. NVthreads uses these synchronization points as the boundary for failure atomic regions. We assume that programs are data race-free and data structures are consistent at synchronization points. However, updates inside critical sections can leave data structures inconsistent due to an untimely crash.

NVthreads uses two rules to guarantee that data structures in persistent memory are always consistent: (1) if an application crashes while a thread is in a critical section, NVthreads ensures that updates to persistent data in the critical section are not visible in NVM, and (2) if an application crashes when a thread is outside of a critical section, NVthreads guarantees that only modifications till the last successfully executed critical section are made visible to NVM. The challenge for NVthreads is to ensure that these rules hold true even in the presence of multi-threading and volatile caches.

Returning to the example in Figure 3.1, if the program crashes at line 9, then the last element in the list will not have a back-pointer and the variable tail will no longer point to the last element. NVthreads avoids leaving the list in such an inconsistent state by making the critical section failure atomic. Let's assume that there are two threads T1 and T2 executing the critical section to append elements to the list. If T1 has successfully completed the critical section and T2 crashes in the middle of executing the critical section, then NVthreads will ensure that the persistent list, after program recovery, has only one new element.

Dependent critical sections. Locks lead to multiple kinds of critical sections. There may be cases with (1) an inner critical section completely surrounded by the outer critical section (perfect nesting), (2) two critical sections that overlap, such as lock chaining [85], or (3) critical sections that use condition variables. In all these cases, additional care is taken by NVthreads to ensure data structure consistency.

Figure 3.2 shows two examples of nested critical sections, one with perfect nesting and another with overlapping critical sections. Note that the case where critical sections don't nest perfectly does not arise in programs with transactions because transactions are scoped regions without partial overlap [62, 63]. NVthreads treats nested critical sections as a single failure atomic region. Logically, the system provides atomic durability to the smallest program region that encompasses all lock acquires and releases in a particular nested critical section. In Figure 3.2, NVthreads will guarantee that lines 2-7 (and 9-14) are atomically durable. For example, if a crash

```
1  //Well nested section
2   pthread_lock(&m1);
3     pthread_lock(&m2);
4     ...
5     pthread_unlock(&m2);
6     ...
7   pthread_unlock(&m1);
```

```
8  //Lock chaining
9   pthread_lock(&m1);
10    pthread_lock(&m2);
11    ...
12  pthread_unlock(&m1);
13    ...
14    pthread_unlock(&m2);
```

Figure 3.2.: Different types of nested critical sections.

```
1  //Thread T1
2  a = 0;
3  pthread_lock(&m1);
4    pthread_lock(&m2);
5      a = 42;
6      pthread_wait(&cv, &m2);
7      ...
8    pthread_unlock(&m2);
9    //crash
10 pthread_unlock(&m1);
```

```
1  //Thread T2
2
3
4  ...
5  b = 0;
6  pthread_lock(&m2);
7    b = a;
8    pthread_signal(&cv);
9  pthread_unlock(&m2);
10 ...
```

Figure 3.3.: Dependence between nested critical sections.

occurs in line 6 then changes made even in the internal critical section (lines 3--5) are undone. NVthreads *tracks dependence* between nested critical sections to correctly reflect updates in NVM. For example, the inner critical section in lines 3-5 is dependent on the outer critical section, and should become durable only after line 7 successfully completes.

Figure 3.3 shows another example of why NVthreads tracks dependence between nested critical sections. Line 3-10 in T1 has nested locks and a condition variable, but it is a single failure atomic region, i.e., even if a crash occurs at line 9 the value

of a should be 0 after recovery. However, in this example T2 will set b to 42 (line 7 under T2) before the crash occurs. After recovery, this will lead to an inconsistent state where a is 0 but b is 42. NVthreads ensures that such cases do not arise by tracking dependence between critical sections. In this example, the critical section in T2 is dependent on T1, and the changes in T2 will not be visible in NVM unless T1 completes the nested critical section. Section 3.5.1 describes how NVthreads implements dependence tracking.

3.4.3 Recovery Code

In the event of a crash, NVthreads guarantees that program data structures are durable in NVM up to the point of the last successfully completed critical section. The recovery component consists of two parts. First, applications invoke NVthreads' recovery function, nvrecover, to apply log entries to NVM-resident data. This component is application agnostic. Second, similar to other systems, programmers may need to write application specific recovery code to resume execution after a crash [61, 63]. The user-provided recovery code is primarily used to assign NVM data to program variables, such as reading back two separately allocated arrays that are fields of a single variable, and assigning them to the appropriate fields.

While the amount of user-provided recovery code depends upon the complexity of an application, most programs simply need to call the application agnostic nvrecover for each allocated data and assign the output to program variables. In our experience, the recovery code for many machine learning and graph algorithms is only a few lines that read core data structures from NVM and restart iterations to run until convergence (Figure 3.4).

3.4.4 Garbage Collection

Due to crashes, applications using non-volatile memory have to handle cases of persistent memory leaks and dangling pointers. As an example, if a persistent object

```
1  /* points: input 2D points
2     labels[i]: id of center closest to point i
3     Only 'labels' is persistent  */
4  // Main iterations
5  float* kmeans(float* points, float* labels){
6     centers = calculateCenters(labels);
7     while (!converged){
8        pthread_create(.., findDistance, ..);
9        pthread_join(..);
10       centers = updateCenters(labels);
11    }
12    return centers;}
13 // Calculate distance for subset of points
14 void findDistance(points, labels, centers){
15    //find closest center for points with id in [X,Y]
16    pthread_lock(&m_xy);
17    labels[X:Y] = closestCenters[..];
18    pthread_unlock(&m_xy);
19    ...
20 }
21 //Recovery code
22 void main(){
23    if (crashed())
24       nvrecover(labels, N*2*sizeof(float),'labels');
25    else
26       labels=(float*)nvmalloc(N*sizeof(float),'labels');
27       ...
28    ans = kmeans(points, labels);
29 }
```

Figure 3.4.: Pseudo-code for multithreaded K-means.

stores a pointer to volatile memory then after a crash it will point to garbage since volatile contents are lost. Such programs are discouraged, but NVthreads currently does not enforce these pointer restrictions. Similarly, after recovery if programs do not reuse persistent data that they store, the memory space will be wastage. NVthreads assumes that a garbage collector exists for the persistent regions (similar to file system checkers). This garbage collector should run after a crash, collecting unreachable memory, and flagging dangling pointers.

3.4.5 Example: K-means Clustering

Iterative machine learning algorithms, such as clustering on large datasets, can take hours to converge even with multiple CPU cores (Section 3.6.4). Therefore, after a crash it is beneficial to restart the program from the last completed iteration instead of the beginning. Figure 3.4 is an implementation of K-means algorithm that becomes crash tolerant when linked with NVthreads. K-means is a clustering technique that divides the input dataset (stored in the array points) into K groups. The algorithm proceeds in rounds, refining the centers until convergence. In an iteration, each point is first assigned to the closest center (stored in the array labels), and then centers are updated by taking the average of points assigned to them.

In lines 5-12 the centers are first initialized using the current labels. In each iteration threads calculate the closest center to a subset of points, and update the corresponding labels in a critical section (lines 16-18). When a thread exits the critical section NVthreads guarantees that the labels of all the points it was working on have been updated. Lines 23-24 depict the recovery code. After a crash, the program re-reads labels from NVM. Otherwise, it allocates memory in NVM to store the labels. The real recovery work is performed by the nvrecover function which is application agnostic and implemented in the NVthreads runtime. After a crash, the K-means algorithm will simply restart its execution by calculating the latest centers from labels.

Algorithm 1 Execution flow for each thread T

```
1: while T has not terminated do
2:     w_T ← ∅                                          ▷ Initialize write set
3:     while instruction i is not a synchronization event do
4:         if i is a memory store then
5:             w_T ← w_T ∪ {page with memory address}
6:         end if
7:         Run i on priv. copy of global state          ▷ Avoids fine-grained logs
8:     end while
9:     log(w_T)                                         ▷ Log diff of modified pages
10:    Merge diff of pages from private copy to global state
11:    Execute synchronization event
12: end while
```

3.5 Design and Implementation

Algorithm 1 shows how NVthreads implements its persistent programming model. As each thread executes, the write set tracks updates, which are initially performed on a process private copy of data. At synchronization points, modified data is first logged and then merged with the global state. For simplicity we have not shown the dependence tracking for nested critical sections.

An important challenge is to implement Algorithm 1 while ensuring good application performance. Unlike prior solutions, NVthreads uses operating system techniques to reduce application overheads. It leverages DThreads' approach [77] to execute a multithreaded program as a multi-process program, though reducing the overheads imposed by deterministic execution. NVthreads also tracks dependence between critical sections, creates redo logs, and flushes log entries to NVM for recovery.

3.5.1 From Threads to Processes

NVthreads converts threads into child processes that execute in isolation until a synchronization point is reached. Typical synchronization points are lock acquires

and releases, as well as thread creation and exit. At a synchronization point, changes made by child processes are applied to the original shared pages and made visible to all. This merge phase ensures that the behavior of the multi-process execution corresponds to a valid multithreaded execution: threads can access and modify shared data.

Tokens. Since multithreaded programs running under NVthreads become multi-process, we use the term token to denote mutexes visible across processes. Figure 3.5 shows an example of how execution proceeds in NVthreads. We assume that the application has two threads (T_1 and T_2) that become processes when executing under NVthreads. This process-based isolation is used to buffer uncommitted writes. Outside the critical sections, processes execute in parallel. Inside a critical section, such as a locked region, processes execute sequentially, i.e., one after the other. Sequential execution is enforced by making each process wait for a per-lock token. This per-lock token is logically similar to a mutex in a multithreaded program, except that it is visible across processes using shared memory. Only the process that acquires the token can enter the corresponding critical section. Others wait till the token is released. Once a process exits the critical section it writes its dirty pages to the log and flushes the log to NVM for durability. It then merges its modifications to the shared state, which makes the local updates visible to everyone, and finally passes the token to the next waiting process. It is worthwhile to point out that DThreads uses a single per-program global token to ensure there is a deterministic order of thread interleaving. This global token reduces concurrency: it serializes even those threads that access completely different mutexes. For deterministic ordering among threads, DThreads also forces all threads to wait at a barrier each time a mutex is released. This barrier can result in poor performance if there is load imbalance among threads. NVthreads reduces the overheads in some applications by admitting more thread interleavings (Section 3.6.2).

Copy-on-write. Child processes in NVthreads are created using the clone system call and each process gets a copy-on-write version of the program data. Shared mem-

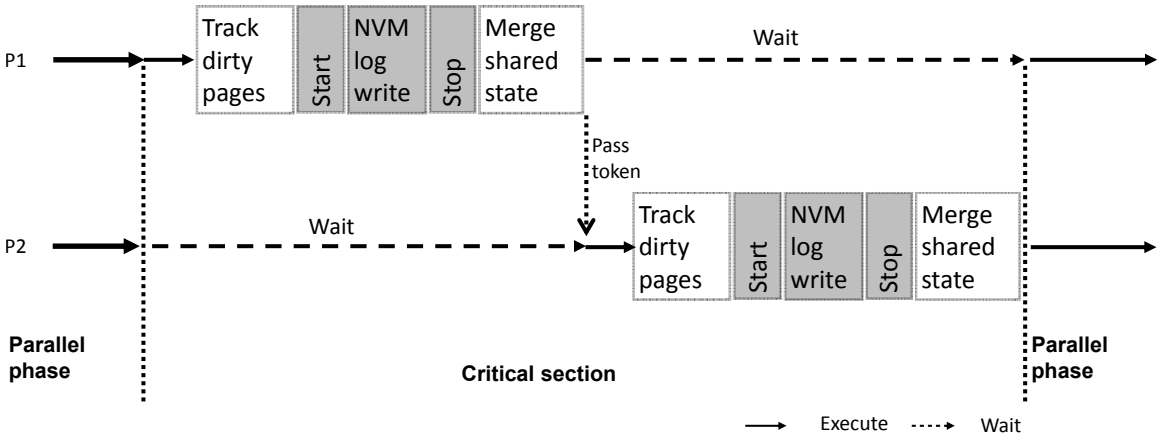

Figure 3.5.: Overview of thread execution in NVthreads.

ory regions are backed by a file, which is mapped into each process' address space and updated at synchronization points. Specifically, each process has two references of program memory: shared and process-local, which are created through two different mmap calls to the same file. Pages are initially read-only outside the critical section. Once processes write to pages, NVthreads' page fault handler uses mprotect on process-local pages with PROT_READ or PROT_WRITE and MAP_PRIVATE flags, effectively creating a copy-on-write page for local modifications. During the merge phase, the runtime compares dirty pages from a child process with the original versions of the shared pages, and applies the bytes modified by the child to the shared state. Only the dirty private bytes (i.e., diffs) are applied to the shared pages at synchronization points. At the end of a critical section, NVthreads releases the private copies and redirects references of these addresses to the shared pages with read-only permission set.

Dependent critical sections. The NVthreads runtime tracks dependence between durable regions to correctly handle nested critical sections and condition variables. NVthreads buffers the generated logs till all threads that are dependent on each other exit their critical section. Figure 3.6 shows how NVthreads tracks this dependence. If a thread T1 is inside a nested critical section, and passes a token to thread T2 for execution, then T2 is dependent on T1 if T2 touches a page that T1 modified in the

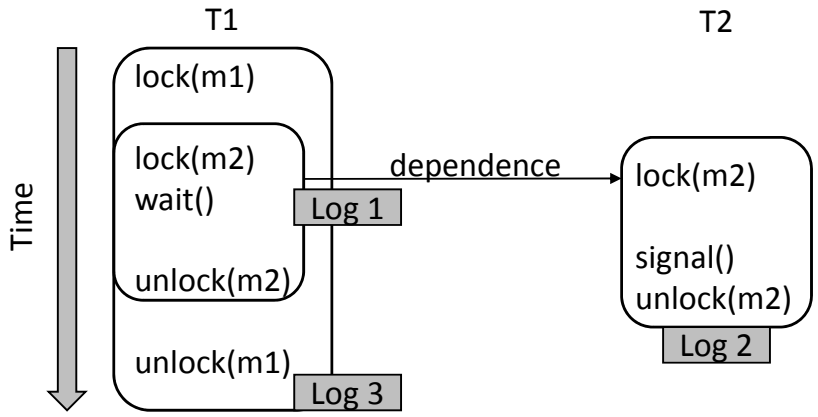

Figure 3.6.: Tracking dependence between durable regions.

current nested section. Only when T1 completely exits its nested critical region are the logs of T1 and T2 together committed to NVM. In Figure 3.6, logs 1, 2, and 3 are written to NVM only when T1 unlocks m1.

3.5.2 Logging

Figure 3.7 illustrates how logging works in NVthreads. Since NVthreads uses multi-process execution, data structure modifications are initially available only in the private copy used by each process. Only at synchronization points, and after dirty pages have been merged with the shared program state, will the changes be potentially reflected in NVM (depending upon when cache lines are evicted). NVthreads requires only logs to be durably flushed from caches to non-volatile memory. Except for the log truncation operation, which we describe later, NVthreads' correctness guarantees do not depend upon when application data is flushed from caches, which reduces the overhead of eagerly flushing cache lines.

Ordered writes to NVM. Since NVM devices appear as memory, the processor may cache data and reorder updates. NVthreads has to ensure that applications can correctly recover even in the presence of volatile caches and re-ordered writes. NVthreads requires two mechanisms from the hardware, which are available in most

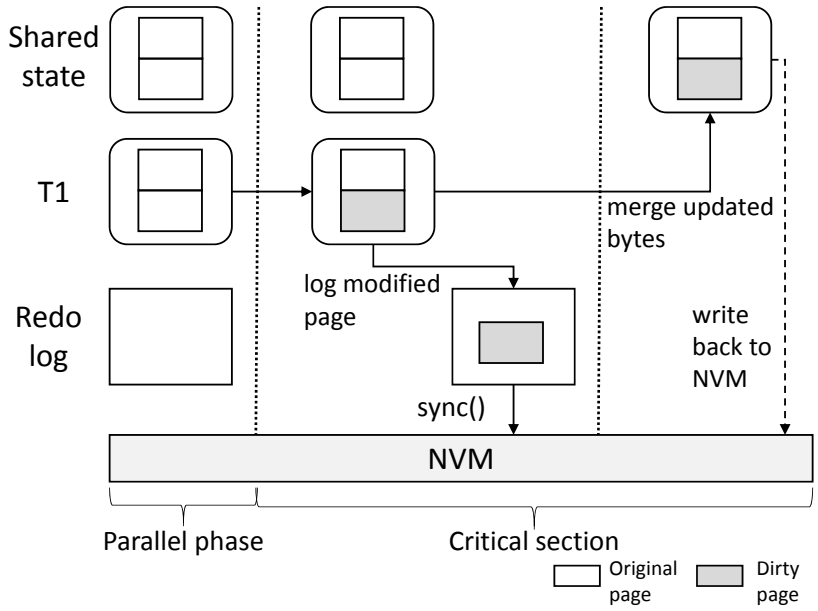

Figure 3.7.: Overview of logging.

processors, and are also expected to be present in NVM systems. First, NVthreads requires a way to evict cache lines to NVM, such as the clflush or clflushopt instruction. NVthreads uses these calls to keep its log consistent and durable. Second, NVthreads assumes that the system provides fences, such as mfence, to ensure that instructions prior to a fence complete before those after the fence. Applications should also be able to use operating system functionality, such as msync or fdatasync, to write out logs to NVM. The fdatasync call, when used as a blocking call, ensures that all writes have safely reached the device before returning.

Coarse-grained memory management. Figure 3.8 contrasts fine-grained memory management [61, 63] with NVthreads. In fine-grained memory management, first each NVM write has to be intercepted in software, then log entries are made durable followed by program data. The sync operator ensures writes reach the NVM and are ordered. It involves draining the volatile cache line using clflush followed by a barrier for instruction ordering. In contrast, NVthreads uses process local pages to buffer writes, and at the end of the critical section, logs modified pages. This approach allows NVthreads to avoid intercepting each NVM write. Additionally, only writes to

```
1  //Fine-grained
2  pthread_lock(&m1);
3   l = log(&a)
4   sync(&l);
5   a = v1;
6   sync(&a);
7   l = log(&b);
8   sync(&l);
9   b = v2;
10  sync(&b);
11 pthread_unlock(&m1);
```

```
1  //Coarse-grained:NVthreads
2  /*Writes are initially to
3  a private copy of data*/
4  pthread_lock(&m1);
5   a = v1;
6   b = v2;
7   ...
8  pthread_unlock(&m1);
9  log_pages();
10 sync();
11 merge_pages();
```

Figure 3.8.: Fine-grained [61] vs coarse-grained tracking.

the log need to reliably reach the NVM. The log provides the opportunity to batch pages and use the unordered but faster clflushopt instruction instead of clflush.

Redo log. NVthreads uses a redo log for crash recovery. Data structure modifications are first committed to the log and then made durable in the application's working space (Figure 3.7). NVthreads uses the mprotect system call and a custom page fault handler to track dirty pages. Right before dirty pages of a process are merged with the shared program state, the dirty portions of the pages (i.e., diffs) are written out to the log. For correctness, the log has to be made durable before merging changes with shared program state. Otherwise, application data may reach the NVM before the log, and a crash in between will result in a case where the system cannot undo inconsistent data structures in NVM. NVthreads uses fdatasync to write out log entries to NVM. After writing out the log entry, NVthreads also writes out a special end-of-log symbol. This symbol is used during recovery to identify if all of the items in a log append were written to NVM. The end-of-log symbol is durably written before the current process continues to merge its dirty pages to the shared program state.

Unlike the log, NVthreads does not force application data to be flushed from volatile caches after each merge phase, thus reducing cache flush overheads. The reason is because the log has sufficient information to recover data in the case of a crash. This approach of not forcing data out to durable media is similar to ARIES like protocols, i.e. *no-force* in database parlance [72].

No undo log. NVthreads does not create undo logs. In the ARIES style of database logging, the modified data of in-flight transactions may be paged out to the disk to make space in the buffer manager. In Atlas [61] updates are made in-place. Therefore, these systems need an undo log to remove the effects of uncommitted transactions or durable sections. In NVthreads each process makes its modifications on a private copy-on-write version of data, and the undo log is unnecessary.

Log truncation. Since reapplying the log from the beginning can be slow, NVthreads supports log truncation to reduce the number of log entries that need to be replayed during recovery. The redo log can be truncated up to a particular entry if the system can guarantee that all changes up to that entry are reflected in NVM. Since the application writes its data directly to NVM, the only issue is to ensure that data in the volatile caches makes it to NVM before the corresponding write entries in the log are truncated. In NVthreads, log truncation is triggered periodically in the merge phase of the synchronization points, when NVthreads has control over the whole program, and the application is quiescent. NVthreads first flushes all cache entries, and then truncates the log, using a fence to order the cache flush before log truncation.

3.5.3 Recovery

Applications initiate the recovery process if they are re-starting after a crash. NVthreads logs the modified portion of each page, i.e., the diff, before they are applied to the shared state at a synchronization point. During recovery, all the log entries are applied to recreate application data pages.

Applications invoke nvrecover, the NVthreads recovery function, which first replays the redo log, applying entries in the log to corresponding pages in the NVM (Figure 3.4). Once the recovery process is complete, it is guaranteed that the application's data structures in the NVM are consistent as of the last completed critical section. Some applications may have additional user written recovery code to read data structures from the NVM, assign them to program variables, and resume execution.

3.6 Evaluation

We evaluate NVthreads to answer the following: (1) What are the overheads of making programs durable? (2) What are the benefits of NVM over flash storage? (3) How important is fast recovery? and (4) How does NVthreads compare to Mnemosyne and Atlas?

3.6.1 Setup

All experiments were run on a Ubuntu 14.04 (Linux 3.16.7) server with two Intel Xeon X5650 processors (12 cores@2.67 GHz), 198GB RAM, and 600GB SSD.

Applications. We used 14 multithreaded benchmarks from PARSEC [65] and Phoenix [66], and ran them with the configuration of a *large* dataset. We also use PageRank, a graph algorithm, and K-means clustering.

NVM emulator. Since NVM devices are commercially unavailable, we use a simple emulator to measure the effect of different NVM latencies on performance. As NVthreads relies on a NVM filesystem to store its log, the emulator consists of a modified Linux tmpfs on DRAM in which software delays are injected to each read and write filesystem call. These delays are created by reading the processor timestamp via RDTSCP instruction and spinning in a loop until the counter reaches a certain value. In all experiments, we add 1,000ns delay to each 4KB page write, which models the expected overhead for write barriers and cache flushes with clflushopt [68].

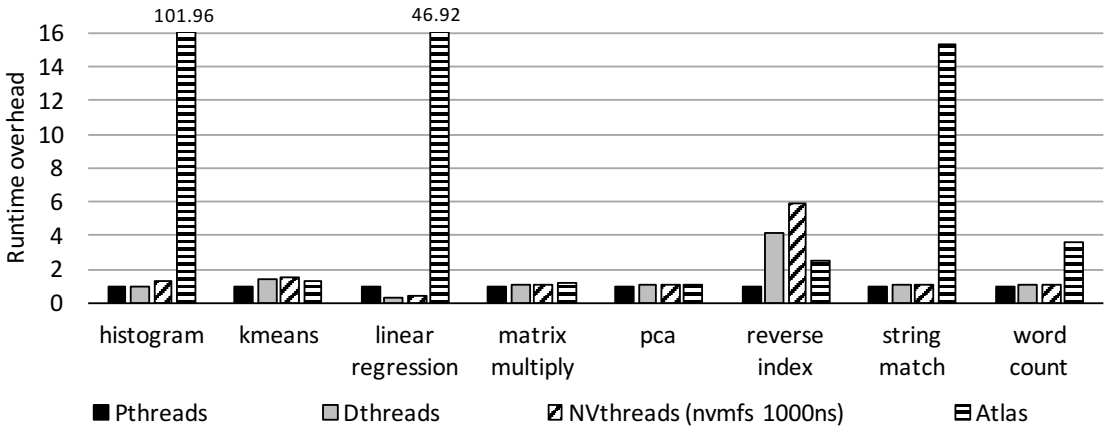

Figure 3.9.: Phoenix applications

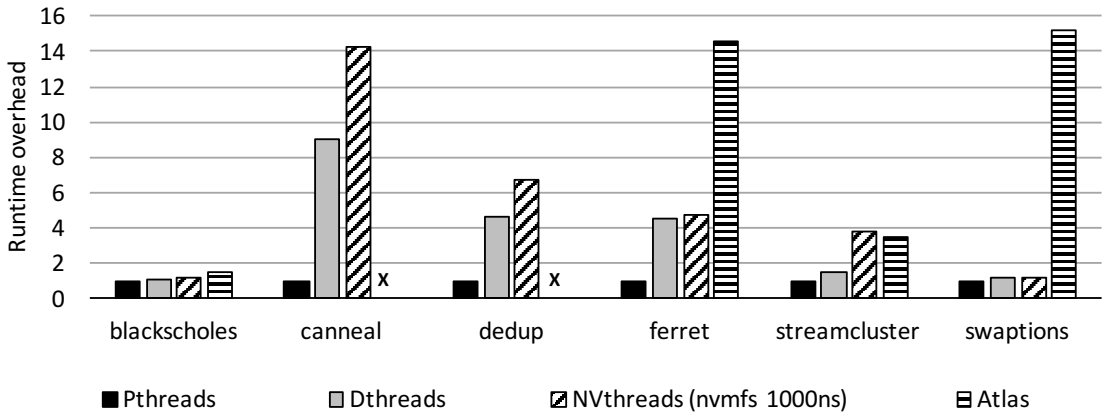

Figure 3.10.: PARSEC applications

3.6.2 Performance

Figure 3.9 and Figure 3.10 shows the overheads of using NVthreads on all 14 applications from the Phoenix and PARSEC benchmarks. Instead of modifying the applications to call nvmalloc and persist only a selective set of data structures, we made all heap allocated data persistent by ensuring each call to malloc is a persistent allocation. This setup ensures that we are measuring the worst case overhead, i.e., when all data is durable, without modifying the application at all. We did not add

restart code to applications for this experiment since the goal is to observe overheads during normal execution.

In Figure 3.9 and Figure 3.10 each program uses twelve threads. The baseline is programs running in DRAM with pthreads. The DThreads performance numbers give a sense of the overheads of converting multithreaded execution into multi-process execution. NVthreads numbers are on the modified tmpfs with a 1,000ns delay to each page write. We also show the performance of these applications when made durable using Atlas, which uses fine-grained cache line flush based logging.

Our results show that for 9 out of 14 applications, NVthreads makes the application durable with less than 28% overhead. Only reverse index, canneal, ferret, and dedup have more than 4× overhead compared to unmodified applications using pthreads. The DThreads bars in Figure 3.9 and Figure 3.10 show that these programs incur considerable overhead when using multi-process execution. However, NVthreads' can improve concurrency in some applications where DThreads would have serialized thread execution due to the global token. As an example, even though canneal with NVthreads is 4× slower than the pthreads version, it is actually 2× faster than the non-persistent DThreads version. Similarly, reverse index would have been 7× slower, instead of 5.5×, if the global token approach was used. ferret and dedup use the global token version of NVthreads, and we expect their overheads to decrease with the per-mutex token version of NVthreads.

Figure 3.9 and Figure 3.10 shows that NVthreads performs as well, and in most cases significantly better, than Atlas. We were not able to get Atlas to run on the two challenging workloads of canneal and dedup in the PARSEC benchmark. In 10 out of the 12 workloads on which Atlas ran, NVthreads is from 7% to 100× faster depending upon the workload. NVthreads is about 7% and 50% slower than Atlas for streamcluster and reverse index respectively. In reverse index, most of the overheads in NVthreads are due to the multi-process execution as shown by the DThreads bar. Section 3.6.5 has more in-depth comparison with Atlas.

Table 3.2.: Application characteristics of PARSEC and Phoenix.

Program	Critical sections	Logging (%tot. time)	Logged pages	Log size	Slowdown over pthread
histogram	25	12.56%	44	0.3MB	1.2
kmeans	1845	20.91%	9763	46MB	1.5
linear reg.	25	12.87%	27	0.2MB	0.3
matrix mult.	37	2.32%	3955	16MB	1.1
pca	25	18.11%	11463	45MB	1.1
reverse index	137113	37.88%	2691474	11GB	5.5
string match	37	3.24%	39	0.3MB	1.1
word count	145	1.72%	12476	50MB	1.1
blackscholes	25	6.76%	89	39MB	1.2
canneal	1475	39.34%	7440183	29GB	4.1
dedup	320492	33.72%	2314600	11GB	6.7
ferret	8010	4.93%	149963	618MB	4.8
streamcluster	95581	47.07%	176054	1.1GB	3.7
swaptions	25	4.76%	483	2.0MB	1.2

Application characteristics. Table 3.2 shows the characteristics of all benchmarks by measuring the number of critical section invocations and logging overheads. The Phoenix benchmarks primarily include data mining applications. Except for reverse index, NVthreads has to log only 27 to 12K dirty pages in Phoenix applications, and results in only 20% slowdown over the pthreads version. In comparison, for some of the PARSEC applications such as dedup, NVthreads has to manage and log about 2.3 million dirty pages and spends 34% of the total time in logging. This high logging overhead, in addition to multi-process execution, is a reason for dedup's 6.7× slowdown. The application linear regression has better performance than pthreads because multi-process execution reduces false sharing.

Our experiments also reveal that NVthreads has low memory footprint and uses at most 400MB more DRAM space versus the pthreads version.

Page-level tracking. Figure 3.11 shows what percentage of each page is modified by the application. It can help us understand whether page-based tracking is a good option compared to byte level tracking. There are two interesting observations. First,

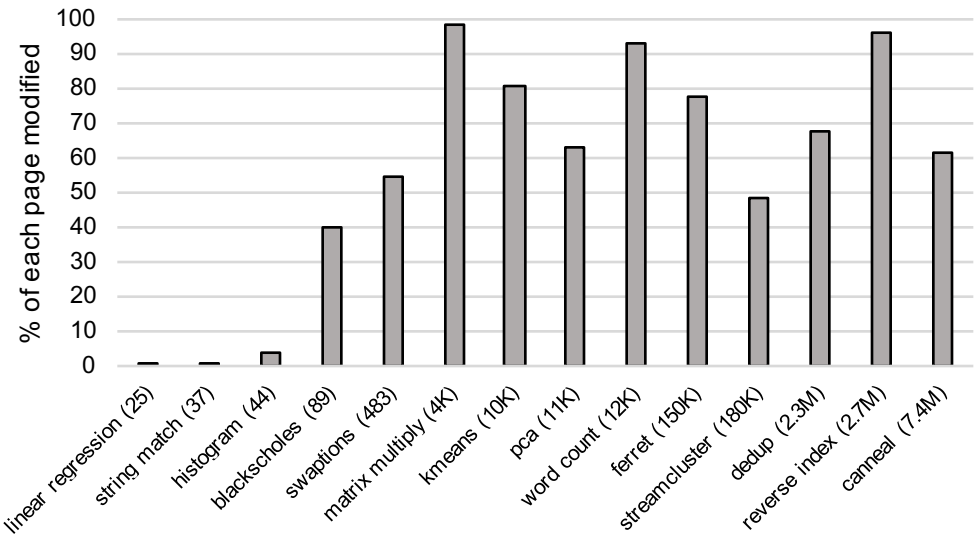

Figure 3.11.: Average percentage of each 4KB page modified by an application. Applications are ordered by total number of modified pages, which is shown in brackets.

9 out of the 14 applications modify more than 55% of each page. This means if these applications modify a page, they generally write more than 2KB of data to the page. This makes it worthwhile to track data at the granularity of a page. Second, of the 5 remaining applications that write only a few bytes per page, 4 of them (linear regression, string match, histogram, and blackscholes) modify fewer than 90 pages during the execution of the program. These applications have few overall writes and, as shown in Figure 3.9 and Figure 3.10, they are less than 20% slower on NVM with NVthreads than running with pthreads on DRAM.

Scalability. We also evaluated whether NVthreads programs scale similar to their pthreads counterparts as core count increases from 1 to 12. As shown in Figure 3.12, our results reveal that pthreads provides low to moderate speedup for the 14 applications, never reaching more than 6.5× speedup over a single core. When using NVthreads 10 applications show similar scalability as with pthreads. Of the remaining four applications, ferret and dedup with NVthreads scale about 70% less than

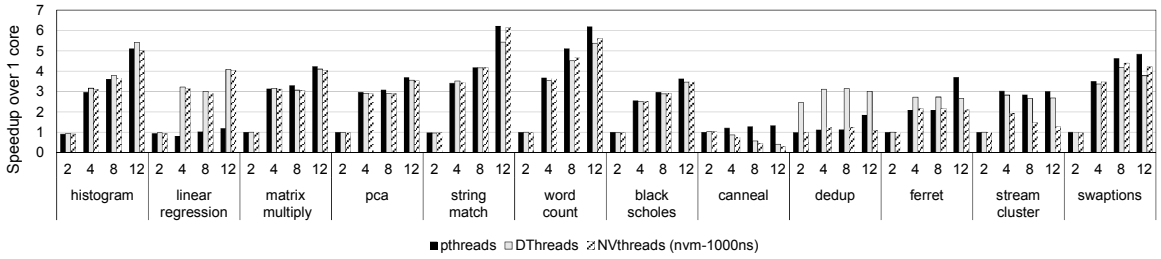

Figure 3.12.: Speedup compared to single core. Higher is better.

pthreads. In `stream cluster`, the pthreads version with 12 cores is 3× faster than one core, but the NVthreads version is only 1.3× faster than single core. Finally, for canneal, the pthreads version at 12 cores is merely 1.3× faster than one core, while the NVthreads version does not scale at all. The low scalability of these 4 NVthreads applications is because of the high synchronization costs and write counts (Table 3.2).

3.6.3 Benefits of Using NVM Versus SSDs

We measure the speedup of programs on NVM over the same programs using ext4 on a solid state drive (SSD) to store the logs. We vary the 4KB page write delay to NVM from 200ns (DRAM-like) to 50μs (Flash-like).

Latency-sensitive applications. Figure 3.13 shows five applications whose performance improves substantially with NVM: streamcluster, dedup, reverse index, canneal, and ferret. There are two interesting observations. First, these applications can be 2× to 22× faster on NVM compared to using SSDs. For example, streamcluster is 22× faster on low-latency NVM than SSD. Second, performance of these applications drop only if NVM page write latency is much more than 1μs. These applications track and write many dirty pages, and will benefit from NVM hardware as long as NVM devices are reasonably faster than SSDs.

Storage-agnostic applications. The remaining nine applications show little difference as we vary NVM latency. Therefore, in Figure 3.13 we plot the performance of

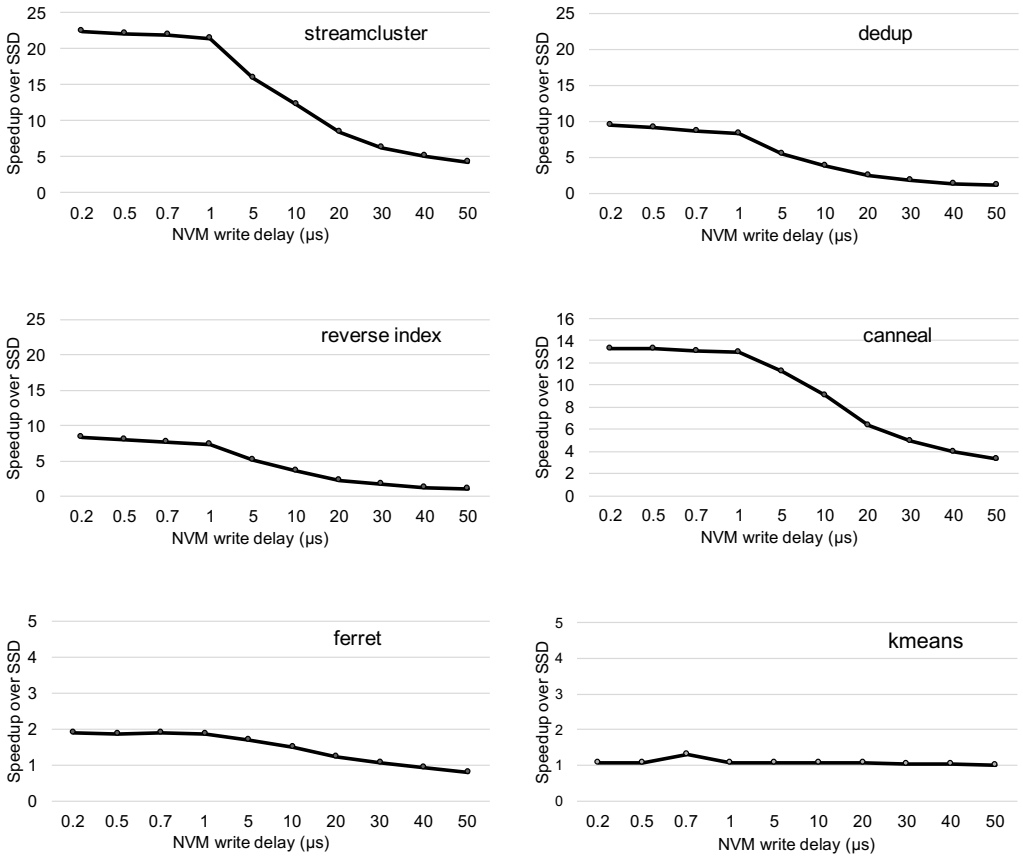

Figure 3.13.: Effect of NVM page write delay on application performance. Speedups are over SSD. Higher is better.

only kmeans. These applications have around 15% performance benefit when using NVM as compared to using an SSD. Note that these applications were anyway incurring a mere 28% overhead compared to their original pthreads versions. They depict a spectrum of applications where NVthreads incurs very little logging overhead to add durability, and we expect programmers to run them with NVthreads even on today's storage systems.

3.6.4 Benefits of Recovery

Figure 3.14 shows the benefits of adding durability to programs. We ran K-means and introduced a crash during its execution. The input data (1M, 20M, and 30M 3-D points) have to be clustered into 1,000 groups. On our datasets, K-means converges after approximately 155 iterations. We had to modify only 4 lines in the program, which nvmalloc and nvrecover the labels array.

In Figure 3.14, the x-axis shows the iteration when the crash occurred. We plot the speedup compared to K-means with pthreads, i.e., the program has to restart from the beginning in the event of a crash. Under these circumstances the maximum possible speedup via recovery on any program is 2×, which happens when the program crashes just before completion, thereby requiring everything to be redone. Figure 3.14 shows that for large inputs, if the crash occurs after 75 iterations, NVthreads' version of K-means converges almost 1.4× to 1.9× faster than starting from the beginning. As an example, for the 30M dataset, the NVthreads version converges 30 minutes earlier than the pthreads version. If the crash occurs too early in the computation, very little work is wasted, and recovery does not provide much benefit. For the small 1M dataset, although the time to converge is only a couple of minutes, NVthreads recovery is still useful if the crash occurs around iteration 150.

3.6.5 Mnemosyne and Atlas

We compare NVthreads with Mnemosyne [63] and Atlas [61] that use word or store level data tracking. We limit our evaluation to microbenchmarks and a real-life graph application because it was challenging to get Mnemosyne to work with other benchmarks.

Microbenchmark. Mnemosyne crashes when we allocate more than 4MB of durable data. Therefore, we use a microbenchmark that allocates 1000 memory pages (4MB of data), and then 4 threads modify a random region of each page. In all cases, threads modify different pages so that the STM in Mnemosyne does not incur any

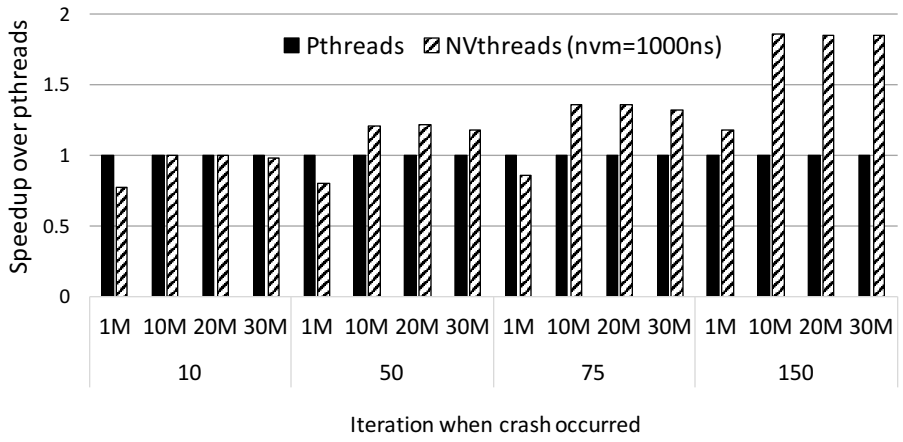

Figure 3.14.: K-means recovery. Higher is better.

concurrency control related aborts. Mnemosyne incurs a constant overhead of 2 seconds when forking threads and using pmalloc. We have excluded this overhead for Mnemosyne. Figure 3.15 shows the slowdown of each system compared to pthreads. Both Mnemosyne and Atlas are 70× slower than pthreads in most cases and more than 200× slower when threads modify 100% of each page. NVthreads incurs an overhead of 25× when only 5% of the data is modified, which decreases to 5× when each thread modifies 100% of each page. Since NVthreads tracks data at page granularity its overheads get amortized as a bigger fraction of each page is modified. As a result, NVthreads is 3×-30× faster than these systems. The bar *Atlas (no-clflush)*, shows that half of Atlas' overheads are due to cache flushes, which point to the high overheads of micromanaging NVM writes. Finally, without any log truncation, NVthreads uses 165MB to store metadata and logs, compared to 550MB-3.2GB by Mnemosyne and 70MB-1.4GB for Atlas.

PageRank. The previous microbenchmark magnifies the overheads of durability since the threads only perform persistent writes. Therefore, we also compare these systems on a real application, PageRank, which is an iterative algorithm to determine the importance of nodes in a Web graph [86]. First, we use the real-world *Slashdot*

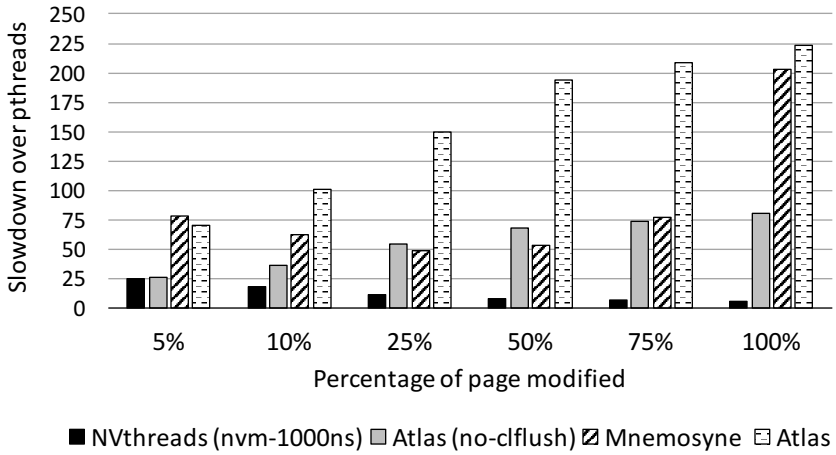

Figure 3.15.: Benefits of NVthreads. Lower is better.

graph dataset which has 82K vertices and 950K edges [87]. We picked this graph because it is the largest input on which we could run Mnemosyne. Even on this small graph, NVthreads is more than 2× faster than Atlas and 10× faster than Mnemosyne at 12 cores, completing an iteration in 170ms compared to 500ms for Atlas and 5 seconds for Mnemosyne. After 10 iterations, NVthreads needs only 273MB space for logs compared to 580MB for Mnemosyne and 600MB for Atlas.

Unlike Mnemosyne, we were able to run NVthreads and Atlas on much larger graphs such as the 1.2 GB *Livejournal* data [88]. NVthreads completes each PageRank iteration in 5s with twelve threads and is almost 6× faster than Atlas which takes 29s. The pthreads version takes less than a second per iteration. NVthreads uses 300MB of log space per-iteration versus Atlas' 650MB.

3.6.6 Key-value Store

Tokyo Cabinet is an high performance, open source library for database management [89]. It stores records as key value pairs. We configured Tokyo Cabinet to

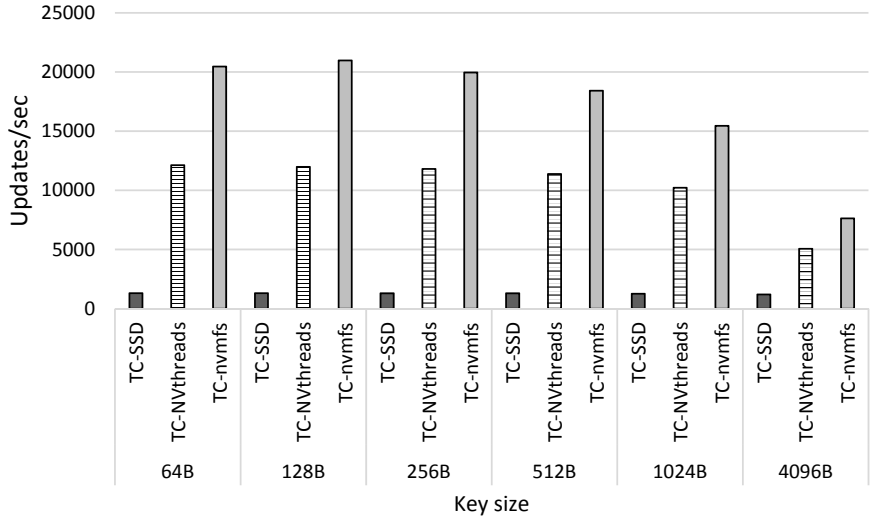

Figure 3.16.: Throughput of Tokyo Cabinet. Higher is better.

organize its records using a B+ tree. Tokyo Cabinet uses a memory mapped file to back data and periodically flushes data using msync to guarantee durability.

Our goal is to make the B+ tree in Tokyo Cabinet durable by linking it with NVthreads. Since Tokyo Cabinet and NVthreads both use page-based writes, we expect unmodified Tokyo Cabinet to have better performance since it uses custom code to implement transations for key-value stores. This experiment shows that NVthreads can be integrated without making any changes to the B+ tree. NVthreads incurs higher overheads but avoids the code complexity of a custom transaction sytem.

In Figure 3.16 we compare Tokyo Cabinet running on (1) an SSD (*TC-SSD*), (2) on our NVM emulator with injected delays (*TC-nvmfs*), and (3) when NVthreads is used to make the B+ tree durable on the NVM emulator (*TC-NVthreads*). In our workload we vary the key and value sizes from 64B to 4096B. We use 8 threads and perform 100,000 write operations. Figure 3.16 shows that when Tokyo Cabinet is used with an SSD (TC-SSD), its throughput varies from 1,300 updates/sec with 64B keys to 1,200 updates/sec with 4096B keys. When we use NVthreads to make the B+ tree in Tokyo Cabinet durable, its throughput on the NVM emulator ranges from

12,000 updates/sec for 64B keys to 5,000 updates/sec for 4096B keys, which is 4×--9× better than using unmodified Tokyo Cabinet on SSDs. Tokyo Cabinet running on our NVM emulator achieves about 20,000 updates/sec with 64B keys and 7,500 updates/sec with 4096B keys. These numbers are about 30%-50% better than what one gets when using NVthreads. Even though the performance when using NVthreads is lower, it provides us an easy way to make the B+ tree durable by merely linking with NVthreads and without developing a custom key-value transaction system.

3.7 Conclusion

NVthreads uses fast non-volatile memory to make C/C++ programs fault tolerant. It leverages synchronization operations to determine consistency semantics, and coarse-grained page-level tracking to manage durable data. Due to these techniques, NVthreads has good performance. Compared to state-of-the-art persistent systems, NVthreads significantly reduces the performance gap between unmodified applications and their crash tolerant versions.

While NVthreads advocates ease of use and the approach of coarse grained tracking, it has certain limitations. NVthreads piggybacks on synchronization primitives to demarcate failure atomic sections. Other explicit or hybrid approaches for detecting failure atomic sections will make the programming model more general. We expect NVthreads to co-exist with systems that embrace fine-grained tracking and logging. NVthreads may perform better on workloads with large amounts of writes in a page, but for certain workloads fine-grained tracking systems may be the appropriate implementation. There are multiple ways to improve our current prototype as well, by adding a memory manager to clean up memory leaks in persistent regions, and by using a combination of regular and huge pages to reduce new overheads seen in in-memory applications [90]. Overall, given the familiar interface of a multithreading library, we believe NVthreads is a design point which makes it simpler for programmers to transition to the non-volatile memory era.

4 MEMORY SUBSYSTEM FOR SCALABILITY

This chapter presents a novel memory subsystem for memory-centric applications to freely access memory beyond the traditional process boundary with support for isolation and crash recovery. Our evaluation shows that the memory subsystem can significantly outperform the traditional process-centric memory architecture when accessing memory in multiple address spaces.

4.1 Overview

In response to the continuous demand for the ability to process ever larger datasets, as well as based on discoveries in memristor [1] and phase-change memory [91] technologies, researchers have been vigorously studying *memory-driven computing* architectures [92, 93] that shall allow data-intensive applications to access enormous amounts of *non-volatile* main memory. It is expected that by 2020 memory-centric computing systems will process vast amounts of data generated by billions of interconnected devices worldwide [94]. These enterprise NVM systems will run applications requiring an unprecedented amount of memory, exceeding what traditional DRAM-based memory architectures can support.

Memory architectures. However, there does not exist a memory architecture that allows applications to efficiently yet safely and securely leverage large (amounts of) memory. As a result, programming with large memory is prone to remain extremely challenging for data-intensive applications. To access large datasets on today's process-centric architectures, programmers have to architect their big data applications using traditional low-level programming interfaces [95, 96] in a tedious and error-prone manner: this includes memory management system calls (e.g., mmap,

munmap) for extending memory space logically, file system interface calls (e.g., open, close) for swapping datasets larger than memory capacity, and multi-processing interface calls (e.g., clone, fork) for managing multiple address spaces and achieving memory isolation. We argue that data-intensive applications require a new memory architecture without involving these traditional interfaces in order to enable memory-centric computing. More precisely, we posit that a memory architecture should provide two forms of decoupling:

Controlled space (de)coupling: In order to scale to large memory, the inherent coupling between a process and its address space has to be abandoned. Doing so allows processes to be coupled with different memory spaces. Multi-processing and parallelism require, inversely, the ability to couple a memory space with more than one process, but security mandates fine-grained access control over shared memory.

Robust time (de)coupling: As both processes and data can outlive each other, it is important that couplings between processes and memory spaces can change over time. With long-lived data, in particular when using large persistent memory to replace external storage systems, it becomes of critical for performance to avoid catastrophic data losses due to host failures.

Enter the AMS. In this paper we thus propose *PetaMem*, a scalable fast memory architecture that enables applications to access memory beyond the traditional process address space boundaries. PetaMem provides a novel abstraction called *autonomous memory space* (AMS) which achieves both controlled space (de)coupling and robust time (de)coupling *in an efficient manner*. More precisely:

Large memory space (LMS) beyond terabytes abstracted conveniently allows for separating processing units from data while avoiding expensive data movement between main memory and back-end storage systems.

Crash recovery (CR) is a key feature *enabled* by future NVM technologies that, in combination with support for retaining consistency by precisely defining commit points, can be leveraged by applications to safely recover data in the event of a crash.

Memory isolation (MI) ensures that data stored in large memory that can potentially be accessed by many processes simultaneously or over time is effectively protected. Compartmentalizing memory with different privileges without expensive context switches enables big data applications to sandbox potentially faulty or insecure software components efficiently and securely.

Efficiency (EF) is achieved in that applications can quickly switch between different address spaces without involving all-level page table manipulation overheads, expensive file swapping operations, or costly process context switches.

Unified programming interface (UPI) centered around the AMS abstraction allows programmers to handle data movement in large memory using simple and pure memory instructions, as opposed to mixing file system interfaces with memory instructions in complex software.

Platform independence (PI) through a software-based memory architecture realizable on any operating system (OS) using page tables for logical address translation allows next-generation data-intensive applications to be deployed on commodity hardware (HW) in a timely manner.

We implemented a prototype of PetaMem on top of the Linux kernel with all the above features. Our evaluation shows that PetaMem can outperform traditional approaches by 500× when accessing large memory and provides up to 5,000× speedup when recovering from failures.

Contributions. This chapter makes four contributions:

- *Design of PetaMem memory architecture* that enables applications to efficiently switch between AMSes to access large memory without any HW modifications.
- *Specification of the PetaMem API* for programmers to utilize the AMSes for fast memory switching, resource isolation, and crash recovery.
- *Implementation of PetaMem memory architecture* that consists of a kernel-level memory management unit and a user space library that implements the PetaMem API.
- *Evaluation of our PetaMem prototype* showing that it outperforms traditional multi-processing and low-level system calls when extending logical memory spaces, and provides effective recovery support after failures.

4.2 Background and Challenges

Memory management is one of the core duties of an OS. In this section we review the challenges for accessing large memory and summarize related work that inspired us to rethink memory management in large NVM systems.

4.2.1 Virtual Memory Size Limitation

Current 64-bit x86 systems follow the AMD64 extension to provide a 48-bit virtual, and up to 52-bit physical, address space. The *48-bit canonical address design* divides the 64-bit virtual address space into higher and lower halves, thereby leaving an unaddressable giant hole in the virtual address space. Since the design implies that a process cannot address more than 48 bits of virtual memory, many vendors currently do not manufacture CPUs that can address more than 48 bits (256TB) of main memory. To echo the increasing demand for addressing large virtual main memory, Intel proposed an architecture that employs a 5-level page table technique named *57-bit canonical address design* [97] allowing 57-bit virtual, and up to 52-bit physical, memory address space. Unfortunately, Intel's proposed solution requires a completely new page table entry (PTE) format in order to address 57 bits (128PB) of virtual

memory, which inevitably breaks the existing AMD64 architecture widely used by modern systems. On today's 64-bit AMD64 systems, the PTE format uses 40 bits to address a physical page, with an addition of 12 bits from the linear address offset to locate data within the page. As a result, even though Intel's proposed 5-level page table uses 57 bits to address virtual memory, the ability to address physical memory is still limited to at most 52 bits (4PB). While today's virtual memory size may seem ample, big data applications already process data exceeding the virtual memory size limitation [98] (LMS), leading data-intensive computing to use traditional techniques for accessing large memory, discussed next.

4.2.2 Traditional Techniques and Issues

Researchers architect data-intensive applications to side-step the problem of insufficient virtual memory by using low-level memory management system calls (e.g., mmap, munmap), file system interface calls (e.g., open, close), or multi-processing interface calls (e.g., clone, fork) [95, 96].

To extend the logical address space of a process, mmap has been a popular choice because of its simplicity (pure memory instructions) and performance (compared to back-end storage swapping) [99]. Nevertheless, constantly changing the memory mappings of a process can be expensive as the kernel needs to establish and tear down all level page tables and data pages for the mapped regions, hampering scalability when the number or the size of memory mappings increases (EF). System V shared memory region has the same scalability problem although it provides persistence. Algorithms that randomly access data across mappings will scale poorly as only few pages are accessed before the mapping is refreshed. File systems are another popular option for accessing data beyond memory capacity [96]. However, in addition to the programming burden caused by heterogeneous data formats (UPI), frequent swapping operations can impact the performance of applications, voiding the low latency advantage provided by large memory (LMS). Finally, multi-processing is used by

programmers to access large memory by partitioning large datasets into chunks and processing the partitioned data into separate address spaces [95, 96, 100], despite the well-known fact that the traditional process boundary makes sharing between address spaces less flexible and more expensive than sharing data directly in memory (EF). Another reason for using multi-processing is the lack of memory boundaries in multi-threaded applications (MI). Many secure applications have been split into multiple processes to achieve strong isolation [6, 7]. As of today, no flexible sharing mechanism exists that provides process-like isolation guarantees and thread-based memory sharing performance for applications to efficiently and securely address multiple address spaces.

4.2.3 Non-volatile Memory

Emerging non-volatile memory (NVM) technologies that promise very large memories are changing the existing design assumptions in system architectures [59]. NVM technologies such as memristor [1] and phase-change memory [91] provide persistent addressable memory at latency as low as DRAM devices [57], i.e., this type of memory is byte addressable, mapped to a process' address space, and persists across process termination and reboots. With careful programming model support, NVM can allow applications to avoid the expensive de/serialization overheads (EF) and recover program state after failures thus achieving crash fault tolerance (CR).

4.2.4 Related Work

Switching address spaces. Modern OSs widely employ the concept of virtual memory to utilize RAM for all processes. Mach [101] manages the memory mappings for a process through machine-independent vm_objects. The physical pages in a memory mapping are accessed by the kernel through the corresponding memory object vm_object. With careful engineering effort, such a design allows fast memory mapping switching by changing the machine dependent vm_objects of a process with-

out modifying the underlying page table entries. SpaceJMP [102] adopts the memory management logic of Mach and promotes address spaces to first-class citizens — *virtual address spaces* (VASs) — allowing threads to switch between virtual address spaces. However, VASs do not support privilege isolation and crash recovery, both being the requirements of a practical memory-driven architecture for data-intensive computing. The design of SpaceJMP can be difficult to be realized in other kernels without the notion of machine-independent memory objects that contains physical pages or similar [103] (PI). On the contrary, PetaMem does not rely on any kernel data structure assumptions but page tables, which are widely used by modern kernels, in order to efficiently support address spaces. In addition, PetaMem allows applications to define flexible memory isolation policies (MI) and to recover from failures when accessing multiple address spaces (CR).

System-level isolation. Switching between address spaces in the traditional process-centric architecture is a costly operation. The heavyweight process abstraction has led researchers to search for lightweight memory isolation techniques. Recent software-based isolation systems such as Nooks [104], Wedge [8], Dune [16], Arbiter [10], SMVs [3], LwCs [105], and seL4 [106] have all demonstrated the feasibility of compartmentalizing memory using finer granularities than a process address space (MI). Other HW-based systems such as CODOMS [40] and CHERI [41] introduced byte-level protection mechanism to isolate memory using specialized HW with capability support (PI). Inspired by the above secure systems, PetaMem's isolation engine makes PetaMem the first memory architecture that allows fast address space switching (LMS) *and* memory isolation (MI) at the same time.

Crash recovery with NVM. Many systems have been proposed to access NVM as main memory with careful programming model support, allowing applications to avoid the overheads of marshaling data between fast memory and slow storage devices and recover program state from crash failures (EF). Atlas [61] infers failure-atomic code sections to automatically checkpoint execution state with compiler support.

Mnemosyne [63] and NV-Heap [62] enable applications to execute durable transactions but they do not support lock-based programs. NVthreads [56] adopts copy-on-write and multi-processing to track data in NVM at page level granularity for performance. Intel's PMDK [93] provides a set of low-level persistent memory tools for programmers to interact with NVM. All above systems can record the execution state of an NVM program and allow for crash recovery (CR), but none were designed to address large memories (LMS). On the other hand, PetaMem allows a program to access memory across different address spaces efficiently and enables a crashed NVM program to resume its execution (CR). PetaMem assumes a system with a large main memory pool without having to involve file system interfaces when accessing NVM (UPI). Accessing NVM through file system APIs is an orthogonal problem that is explored by others (e.g., PMFS [68], BPFS [64], SCMFS [107]).

Single address space OSs. Single address space OSs (SASOSs) [108] such as Opal [109], Singularity [110], and unikernels [111] took an extreme approach to eliminate expensive full context switches by sharing a global virtual address space among *all* processes running on a system. However, the ability to address large virtual memory in SASOSs is still restricted by the canonical address rule (i.e., limited virtual address bits) when running on commodity HW (LMS). In addition, the lack of address space boundary in SASOSs requires the OS kernel and all applications to be written in a memory-safe language to ensure system security (MI). While researchers have proposed an OS kernel written in a memory-safe language [112], in the near future, it is unlikely that existing kernels and all legacy software would be completely rewritten in memory-safe languages for full memory safety [113]. In contrast, PetaMem provides an efficient way for applications to switch between address spaces (LMS) and isolate memory (MI) without completely rewriting the entire software stack, or requiring specialized HW (PI).

4.3 PetaMem Design

PetaMem is a software-backed memory architecture that enables applications to access large memory through a novel address space abstraction. The abstraction allows a process to switch between memory areas in a fast and secure way. This section describes our architectural-level design and the corresponding programming interface.

4.3.1 Autonomous Memory Spaces

To enable a process to address more memory than what today's ISAs (instruction set architectures) can support, we introduce a new memory abstraction called *autonomous memory space (AMS)* that completely decouples the traditional process address space from the process abstraction in the system (LMS), yet in a controlled fashion (MI). An AMS is a special memory mapping with a contiguous range of memory addresses. Data stored in an AMS can be persisted in the system beyond the lifetime of a process and allows for efficient data recovery. As shown in Figure 4.1, PetaMem can dynamically couple an AMS with a process address space and decouple an AMS from a process address space after use. PetaMem guarantees the consistent state of other memory sections including the text, data, and stack in the process memory. PetaMem can keep the data stored in an AMS alive when a process decouples the AMS from its address space. The AMS abstraction enables applications not only to address large memory by switching between different AMSes, but also to checkpoint or to version their execution states. A program can simply re-couple its address space with an AMS from a checkpoint in the previous execution without having to redo all the work from the beginning, thereby improving total execution time in the event of a crash (CR). A program can also go back in time by coupling an AMS to its address space, which can be useful for creating snapshots. In addition, the notion of AMS allows a process to extend its address space without using the expensive fork syscall,

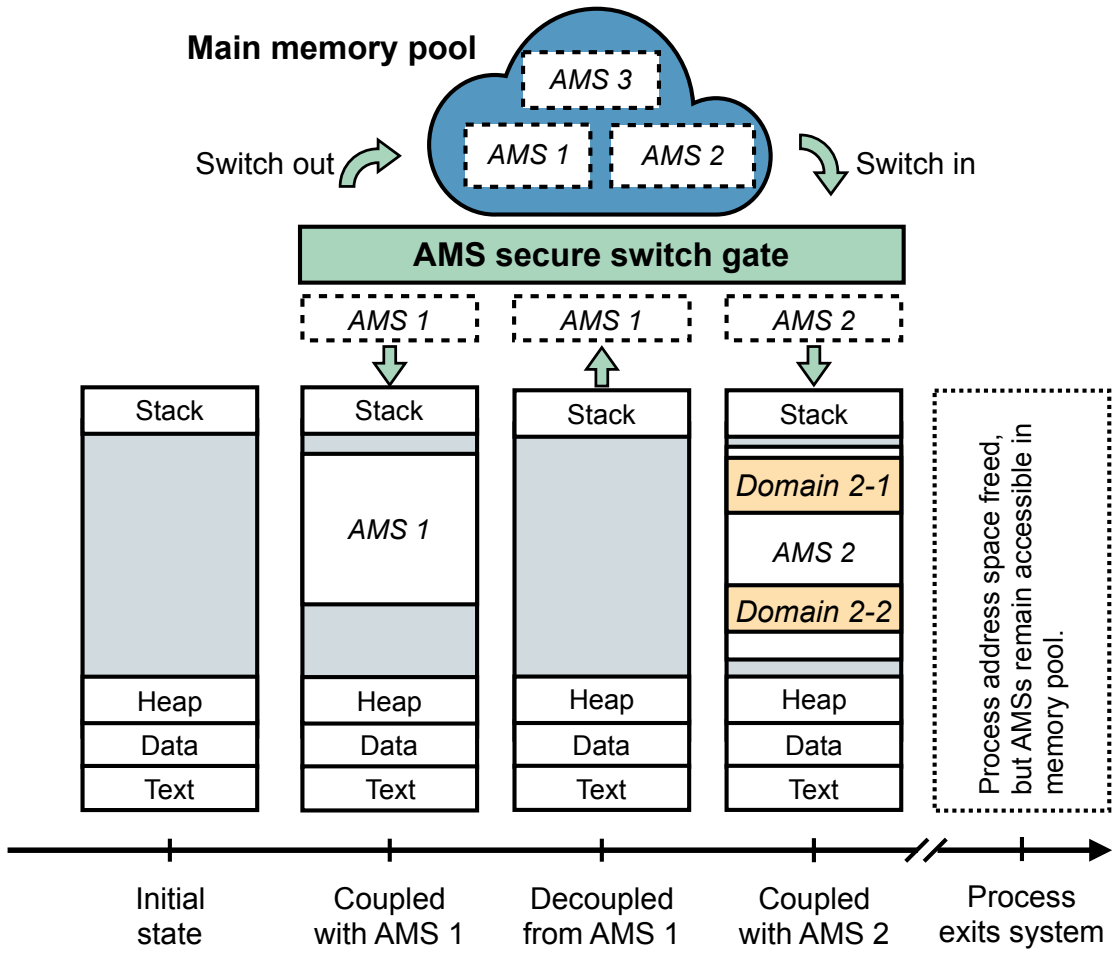

Figure 4.1.: The timeline of a PetaMem process. The PetaMem process memory abstraction allows a process to switch between different autonomous memory spaces (AMSes) that can exist in the system beyond the process lifetime. Note that AMS 2 contains two private domains with special privileges.

which creates an additional heavyweight process with a completely separate address space in the traditional process abstraction.

4.3.2 Inter-process Isolation

We define a *PetaMem process* (PM process) to be a special process that can couple an AMS with its process address space. Each PM process owns a persistent unique identifier in the system. Before a PM process can couple an AMS with its address space, the *AMS secure switch gate* checks whether the requesting PM process has the privilege to access data in the AMS. The creator of an AMS can namely grant other PM processes the privilege to couple the AMS with other address spaces. Each AMS maintains a list of privileged PM processes that are allowed to be coupled with the AMS. With the AMS secure switch gate, PetaMem can prevent unprivileged PM processes from coupling a privileged AMS with their address spaces (MI).

4.3.3 Intra-process Isolation

PetaMem defines a *private domain* to be a contiguous range of memory addresses *within an AMS*. To access memory in a private domain, threads must be granted privileges:

$$\{Read \mid Write \mid Execute \mid Allocate \rightarrow (Domain, AMS)\}$$

The programmer can dynamically manage private domains to selectively isolate an AMS so that only privileged threads can access private domains.

We define a *PM thread container* to be a special sandbox with a collection of domains in an AMS. Threads running in a PM thread container must be explicitly granted permission to access data in private domains. A PM thread container enables programmers to construct software component boundaries within an AMS. For example, two PM thread containers can have different privileges to access the same private domain without having to use different processes to isolate access privileges (e.g., concurrent readers and exclusive writers). Traditional threads (i.e., share all) and processes (i.e., share none, unless with expensive synchronization) do not allow such flexible and secure sharing.

We define a *PetaMem thread* (PMthread) to be a special thread that strictly follows the privileges defined by an assigned PM thread container to access private domains in an AMS. Before a PMthread can access private domains in an AMS, the programmer must grant privilege to the PM thread container that the PMthread is assigned to. PMthreads share process-wide code and data with other PMthreads and regular threads (e.g., pthreads). PetaMem also allows PMthreads to share data in a private domain, which forms a partially shared AMS with a flexible yet secure sharing mechanism (MI). Each PMthread has its own private stack inaccessible to other PMthreads. Such stack isolation is especially desirable for security critical applications that are vulnerable to control-flow hijacking attacks [113].

4.3.4 PetaMem API

We implemented the PetaMem API to allow programmers to interact with AMSes (UPI). Figure 4.3 summarizes the core API. A PM process can create, couple with, decouple from, and delete AMSes. When a process initializes itself as a PM process, PetaMem reserves a contiguous range of virtual memory in the process address space for AMSes. Only the creator of an AMS can modify the privileges of the AMS or delete the AMS from the system. Programmers can configure private domains to achieve selective memory isolation (i.e., controlled sharing) for PMthreads.

Figure 4.2 gives an example of the PetaMem API usage in a program: The process initializes the PetaMem environment and creates two additional AMSes. After switching into AMS 1, the process allocates an integer variable num and assigns it a value 1. Then the process switches into AMS 2 and writes a value 2 to the "same" variable. Since the process is coupled with AMS 2, outputting the variable num shows 2. When the process switches back to AMS 1, the value stored in num becomes 1. Finally the process cleans up the system resources by calling petamem_main_finalize. Note that the runtime and kernel can distinguish between num in AMS 1 and AMS 2. Such ability to decouple memory from a process provides programmers with a powerful

```
1  unsigned long TB_SIZE=1UL<<40; // 1TB
2  void main(){
3      ...
4      // Initialize and couple with AMS 0
5      ams_id[0] = petamem_main_init();
6      // Create additional two AMSs
7      ams_id[1] = petamem_create();
8      ams_id[2] = petamem_create();
9      // Switch from AMS 0 to AMS 1
10     petamem_switch_ams(ams_id[0], ams_id[1]);
11     // Allocate memory in AMS 1
12     num = (int*)petamem_malloc("table", TB_SIZE);
13     // Set variable num to 1 in AMS 1
14     num[0] = 1;
15     // Switch from main AMS 1 to AMS 2
16     petamem_switch_ams(ams_id[1], ams_id[2]);
17     // Allocate memory in AMS 2
18     num = (int*)petamem_malloc("table", TB_SIZE);
19     // Set variable num to 2 in AMS 2
20     num[0] = 2;
21     // In AMS 2 now, output: num[0] = 2
22     output(num[0]);
23     // Switch back to AMS 1
24     petamem_switch_ams(ams_id[2], ams_id[1]);
25     // In AMS 1 now, output: num[0] = 1
26     output(num[0]);
27     // Switch back to AMS 0
28     petamem_switch_ams(ams_id[1], ams_id[0]);
29     // In AMS 0 now, num is uninitialized
30     output(num);
31     ...
32     // Resource cleanup
33     petamem_main_finalize();
34 }
```

Figure 4.2.: Example of PetaMem API usage. The PetaMem API associates a variable name and a AMS upon allocation in a system-wide allocation log to distinguish memory pages.

PetaMem API	Description
int petamem_main_init(**void**)	Initialize PetaMem environment and enter the main AMS.
int petamem_main_finalize(**void**)	Finalize PetaMem environment and clean up the main AMS.
int petamem_create(**void**)	Create an AMS and return a system-wide unique ID assigned by the kernel.
int petamem_kill(AMS_ID)	Kill the AMS and clean up related metadata.
int petamem_get_id(**void**)	Return the current AMS_ID that the caller currently couples with.
int petamem_switch_ams(src, dst)	Switch from one AMS to another. The caller can use memory in destination AMS after the function returns.
void* petamem_malloc(name, AMS_ID, size)	Allocate a memory chunk in AMS_ID and record the allocation in log.
int petamem_free(name, *ptr)	Free the memory chunk and delete the allocation from log.

Figure 4.3.: PetaMem AMS management API.

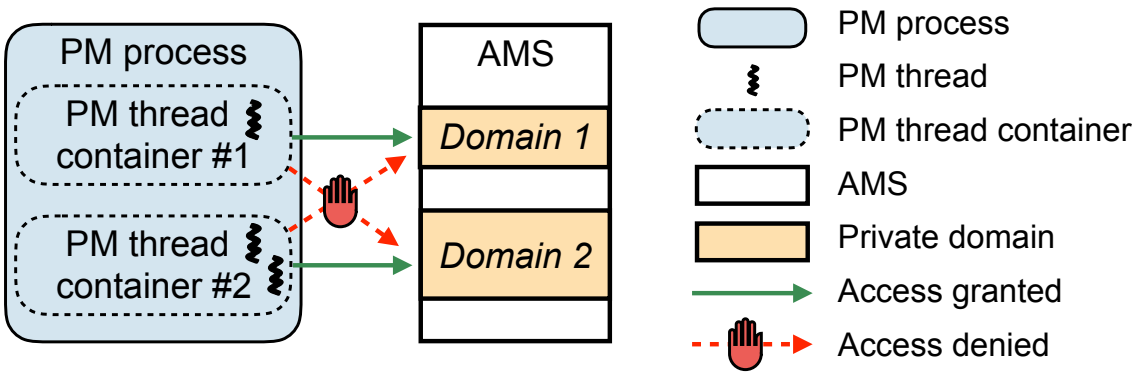

Figure 4.4.: Example of PMthreads isolation in an AMS.

mechanism to pass workloads stored in an AMS between processes (e.g., incremental computation, pipelined operation) as well as a fast way to resume execution (e.g., crash recovery).

To isolate memory within an AMS, PetaMem does not require programmers to create a completely separate address space [7]. Figure 4.4 gives an example of how programmers can structure an AMS with different privileges for PMthreads by using the domain_* functions listed in Figure 4.5. Programmers can store program secrets in a private domain and allow only privileged threads to access memory within the domain. In addition to enforcing a security boundary, such compartmentalization

PetaMem API	Description
int petamem_grant (AMS_ID, *exe_path)	Grant the executable with path *exe_path the permission to couple AMS_ID with its address space.
int container_create(**void**)	Create a PMthread container and return the container ID assigned by the kernel.
int container_kill(CONTAINER_ID)	Delete a container and release its resources.
int PMthread_create (CONTAINER_ID, *fn, *args)	Create a PMthread confined to a container and start execution from function fn with argument *args.
int domain_create(**void**)	Create a private domain in the current AMS and returned the kernel assigned ID.
int domain_kill(DOMAIN_ID)	Kill a private domain from the current AMS.
void* domain_malloc(name, DOMAIN_ID, size)	Allocate a private memory chunk in DOMAIN_ID and record the allocation in log.
int domain_free(name, *ptr)	Free the private memory chunk and delete the allocation from log.
int domain_privs_mod (DOMAIN_ID, CONTAINER_ID, privs)	Grant a container to access memory in a domain with privileges specified by privs.

Figure 4.5.: PetaMem isolation API.

also allows programmers to identify faulty threads that issue unintended memory references across user-defined memory boundaries.

4.4 Recovery Engine

In this section we describe PetaMem's crash recovery engine by which applications resume execution after crashes.

4.4.1 Persistent Memory Views

PetaMem enables applications to perceive memory segments as persistent regions that can survive between process restarts, and even between machine power cycles when using NVM as main memory (CR). Data stored in an AMS does not have to be associated with the life cycle of any processes in the system. Such notion of persis-

tence is equivalent to storing data on file systems for durability, but with the benefit of eliminating the expensive swapping operations (EF). It is achieved by recording a variable as a named data area with an AMS identifier upon an allocation, thus presenting an AMS as a persistent memory view. With petamem_malloc, PetaMem allocates persistent memory in a global domain of the current AMS. When isolation is needed, applications can use domain_malloc to allocate memory in a private domain of the current AMS.

4.4.2 Fault Model

PetaMem allows programmers to commit a persistent memory view by flushing data in an AMS from volatile caches to NVM explicitly. Flushing caches ensures that the data stored in volatile caches reaches memory without losing the data that has not yet arrived at main memory in the event of a crash [56, 61]. Once the data in caches reaches main memory, PetaMem marks the region as recoverable and NVM is able to persist the data even without power. The numbers we report in Section 4.6.4 include the overheads of flushing caches.

The crash recovery engine of PetaMem relies on applications to commit persistent memory views at consistent program points for correct crash recovery. Committing a persistent memory view at an inconsistent program point, i.e., inside a critical section, could result in an inconsistent program state as PetaMem does not maintain a undo or redo log to roll back execution. Programmers should checkpoint persistent memory views outside of any synchronization code sections and maintain application specific metadata (e.g., counters for tracking execution progress) in order to recover from failures. To enable recovery from failures in the middle of a critical section, programmers need to create and commit a consistent version of the shared data before entering a critical section. In the event of a failure, PetaMem guarantees that data stored in a persistent memory view will be accessible through variable name and AMS_ID after the completion of the recovery procedure. Programmers can then use

```
1  void main(){
2    ...
3    // Recovery code
4    if (petamem_crashed()){
5      petamem_recover(&ams_id_array);
6      petamem_recover_var("table",ams_id, size,&table);
7    }
8    // Normal execution
9    else  {
10     table = (int*)petamem_malloc("table",size);
11   }
12   update_loop(table,size);
13   ...
14 }
```

Figure 4.6.: Pseudo-code for crash recovery using PetaMem.

application specific metadata (e.g., variable names, progress counters, or consistent state identifiers) to restore the data in a persistent memory view back to a consistent state. Note that restoring execution state *automatically* for NVM applications is an orthogonal topic that is covered by techniques that inter failure atomic sections using locks [56, 61]. Such user-space techniques can be run on top of PetaMem to simplify code refactoring efforts for programmers.

4.4.3 Recovery Code

PetaMem provides a simple interface summarized in Figure 4.7 for applications to recover from crash failures (UPI). Programmers can use petamem_crashed to examine the crash status of the calling process. In the event of a crash, programs initiate the recovery routine that consists of two parts. First, programmers invoke the application agnostic functions petamem_recover and petamem_recover_var (Figure 4.6) to notify

PetaMem API	Description
bool petamem_crashed(void)	Check whether the process has crashed in the previous run.
int petamem_recover(*ams_id_array)	Recover all AMSes from previous run and couple with the last AMS before a crash.
int petamem_recover_var (name, AMS_ID, size, **ptr)	Recover a variable name allocated in AMS_ID and set *ptr to the recovered memory. Caller must be coupled with AMS_ID before the variable can be recovered.
int petamem_persist(**void**)	Flush caches to persist data pages for the current AMS.

Figure 4.7.: PetaMem recovery API.

the PetaMem recovery engine to recover the PetaMem metadata and the requested variable in a target AMS, which exists in the system despite process failures. The PetaMem crash recovery engine maintains a system-wide crash record that contains all allocated variables in all persistent memory views. Programmers need to specify the variable name and AMS_ID to locate the variable in the specified AMS_ID for recovery. After petamem_recover_var returns, programmers can access the recovered variable in memory. Second, the crashed application may need to execute application specific code to resume operation. The requirement for applications to construct an application specific recovery logic for crash tolerance is similar to other systems [61, 63].

Figure 4.6 gives an example of a crash fault-tolerant application with PetaMem. In normal execution, the application allocates table in an AMS (line 11) and should execute the update loop (line 13) until reaching the end of the execution. If a crash happens in the middle of the update loop, restarting the application will trigger the recovery procedure (line 4 to 8). The crash recovery engine will locate the AMS in the system and recover table to enable the application to resume execution for a faster update loop. While the simple example requires only one line to recover an array for the update loop, the complexity of an application decides the amount of code required for recovery.

Component	LoC	Files	Level
PetaMem API	1281	8	user, untrusted
Recovery engine	480	2	user, untrusted
Communication channel	553	2	kernel
PetaMem Kernel	3035	65	kernel

Table 4.1.: PetaMem component sizes.

4.5 Implementation

In this section we describe the implementation details of our PetaMem prototype. We implemented PetaMem for Linux kernel version 4.4 on the x86-64 architecture. Table 4.1 summarizes the components of our PetaMem prototype.

Figure 4.8 shows the high-level architecture of PetaMem. We implemented the PetaMem design with two main components: a user space programming interface with a crash recovery engine and a kernel space memory management unit. The user space programming interface provides programmers with an API to manage the memory space of an application (UPI). The crash recovery engine tracks the execution status of PM processes in the system and provides useful information to help a crashed PM process resume execution from its last checkpoint (CR). For exchanging messages between the PetaMem API in user space and the kernel, we added the PetaMem channel that sanitizes inputs before handing over messages to the kernel.

The extended kernel implements the core functionalities of PetaMem. The PetaMem metadata manager oversees all PetaMem-related activity. The PetaMem pager switches in and out memory pages of an AMS (LMS). To manage inter-process isolation, we added the AMS secure switch gate to manage system-wide privilege information and ensure that only the PM processes with the correct privilege can access memory in an AMS. To isolate PMthreads, we added an intra-process isolation engine to manage the isolation setup for every AMS in the system. We modified the page fault handler in the kernel to enforce privilege separation and block any underprivileged memory

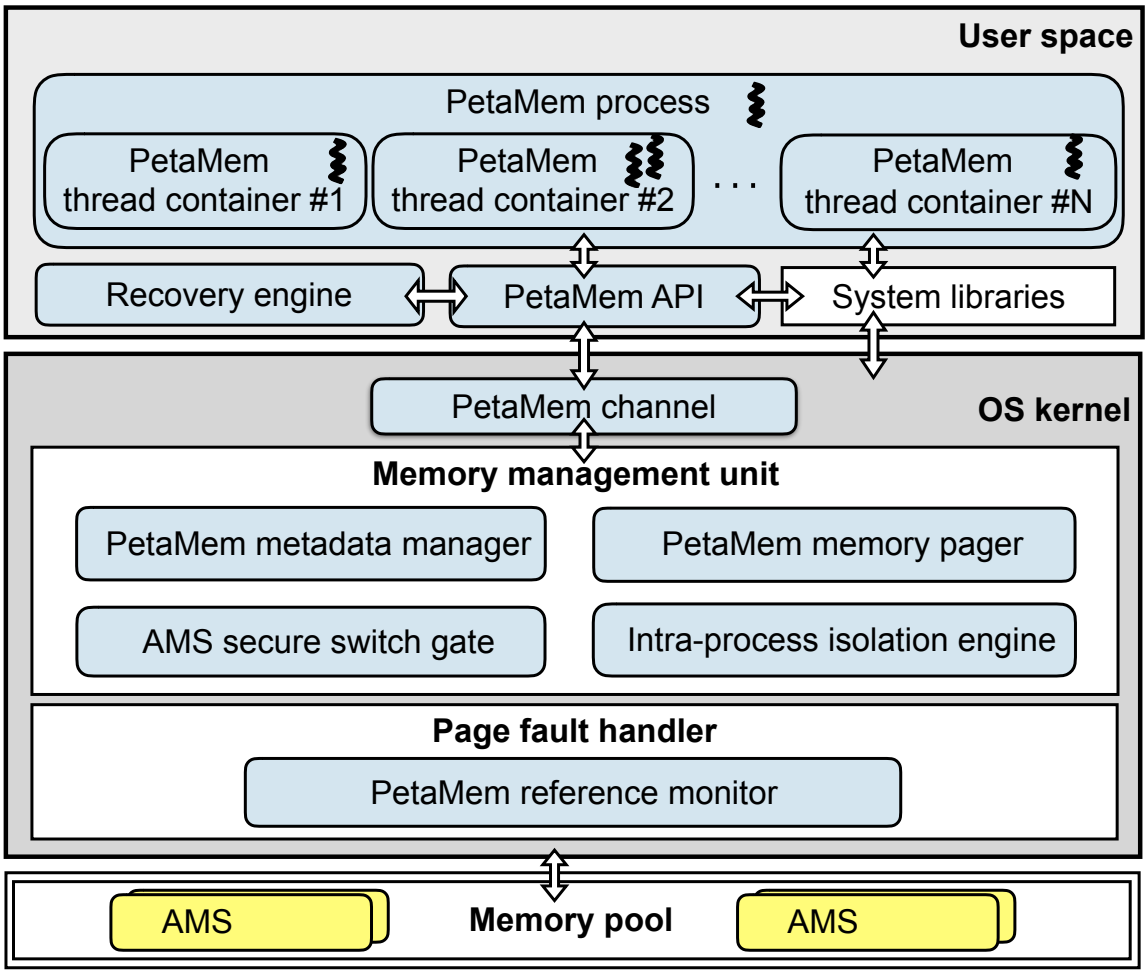

Figure 4.8.: PetaMem Architecture.

references issued by PMthreads (MI). PetaMem can be deployed to OS kernels that use page tables for virtual address translation without any HW modifications (PI).

4.5.1 PetaMem Channel

We developed a PetaMem kernel module that allows the user-space PetaMem API to exchange messages with the kernel through a well-defined Netlink socket interface in Linux. All the messages from the user-space PetaMem API are sanitized by the

PetaMem channel. The kernel interprets the sanitized messages and calls the related kernel functions to complete the requests of the PetaMem API.

4.5.2 PetaMem Metadata Management

The PetaMem metadata management unit in the kernel efficiently manages the state of all AMSes in the system. For each AMS in the system, we added an ams_struct to describe its private domains, PM containers, PMthreads, and privilege information. Within an ams_struct, we added two major kernel objects to manage the private domains of an AMS. (1) domain_struct: private domain metadata for tracking private domains in an AMS. (2) container_struct: privilege information for accessing private domains in an AMS. To efficiently decide whether a virtual address is protected by a private domain, we added vm_range objects to store the virtual address range for all private domains. All vm_range objects are maintained by a red-black tree with a temporal cache, which allows the kernel to retrieve the privilege information when a page fault happens in $O(logN)$ time, where N is the number of private domains. We discuss the privilege check logic in Section 4.5.6

4.5.3 Enabling Multiple AMSes

When a process creates an AMS, PetaMem allocates a new memory mapping vm_area_struct in the kernel to represent the region with a special VM_PETAMEM flag. This special flag prevents the AMS mapping from being merged with neighbor mappings or split into separate mappings. Recall that most CPUs follow the 48-bit canonical address design, which forbids a process to address more than 48 bits of memory. Therefore, we design PetaMem to commit 64TB (46 bits) memory in a single user-kernel round trip. PetaMem chooses to use 46 bits for a typical AMS size to allow for other memory regions such as text, global data, heaps, stacks, and page guards, to co-exist in a process address space. Using multiple AMSes enables a process to unlock the potential for accessing more memory than 48 bits (LMS).

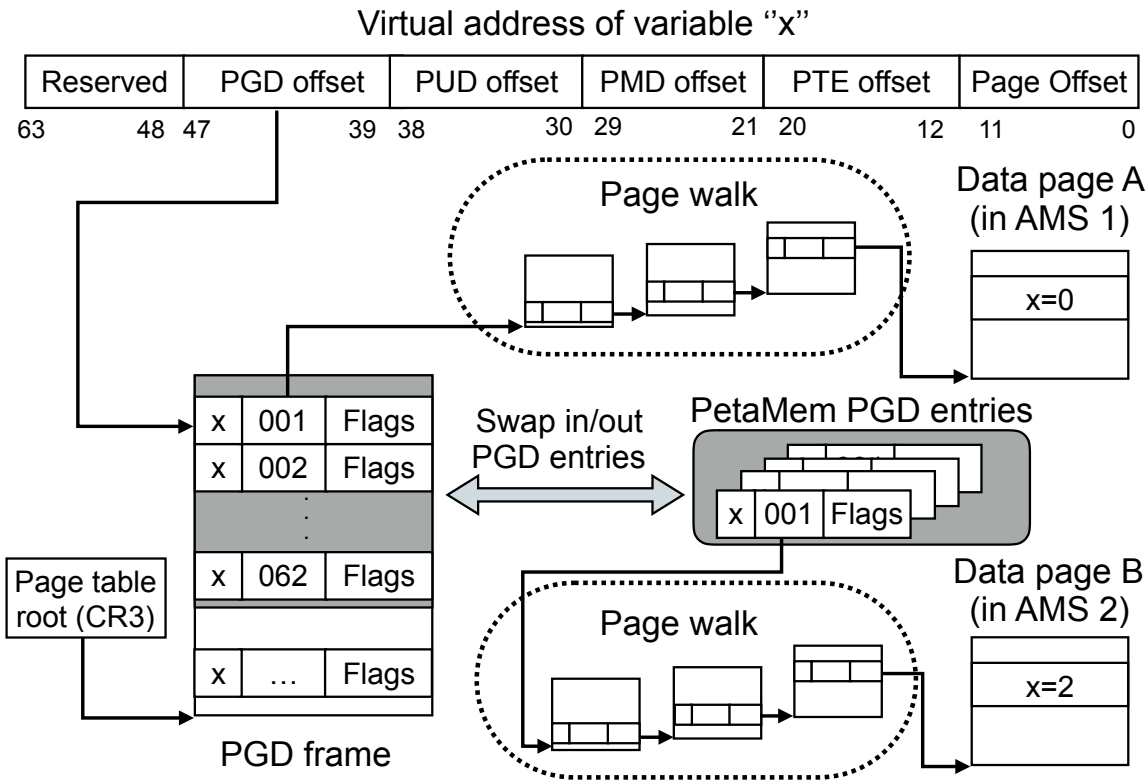

Figure 4.9.: AMS memory management.

PetaMem uses demand paging and pages in data upon page faults. Therefore, the creation time of an AMS does not increase with the size of the AMS. This design effectively removes the scalability issue when accessing large memory.

We developed a novel mechanism to switch the address space of a process from one AMS to another efficiently. PetaMem manipulates the page tables covering an AMS mapping. A straightforward approach is to manipulate every level of page tables when switching address spaces. However, this approach would greatly impact the performance and scalability of a program due to the expensive and frequent page walks. To overcome this well-known problem when modifying page tables, PetaMem swaps *only* top-level page global directory entries (pgd_t) that cover an AMS mapping. We illustrate the design in Figure 4.9: The process reads a variable x that refers two different values in two separate AMSes. When coupled with AMS 1, reading the

variable x returns 0 because PetaMem retrieves data page A associated with AMS 1. After switching into AMS 2, PetaMem retrieves data page B with 2 stored in the variable x by performing a page walk from a pgd_t that belongs to AMS 2.

On a 64-bit architecture, a pgd_t points to 1TB of memory. Therefore, switching 62TB of memory only requires PetaMem to load 124 pgd_ts. The cost of this operation is negligible compared to the naïve approach. PetaMem employs the PetaMem metadata manager to systematically manage the mapping between a set of pgd_ts and an AMS so that an application can correctly roam between AMSes. To prevent unintended access, PetaMem aligns the start and end address of an AMS to page boundaries and inserts a random amount of guard pages on both ends that randomizes the start and end address of the AMS.

4.5.4 PM Process and PMthreads Memory Management

When a PM process couples an AMS with its address space, our kernel assigns the AMS's ams_struct the PM process's memory descriptor mm_struct so that other MMU components in the kernel can easily access the ams_struct of an AMS. Our kernel records the pdg_t entries of an AMS in the AMS's ams_struct. Unlike a traditional process, the memory space described by an ams_struct can be completely decoupled from a PM process's memory. Therefore, a PM process can couple with different AMSes freely, allowing the memory space in an AMS to stay alive in the system beyond the lifetime of any PM process. To separate privileges for PMthreads, each container in an AMS uses one page table root (pgd frame) to describe the isolated memory space. During context switch, our kernel ensures that an isolated PMthread must use its assigned container's page table to access memory to prevent privilege escalation.

4.5.5 Private Memory Allocation

Instead of granting all PMthreads the same permission to access memory in an AMS, PetaMem allows applications to structure AMS into several different private domains. The PetaMem memory allocator internally uses the region-based memory allocator dlmalloc [114] to manage memory in private domains. By specifying a base address and region size, a region-based memory allocator can allocate memory within the specified range. The PetaMem API registers a memory region within an AMS mapping in the kernel for each private domain. Each AMS has a number of new kernel objects vm_range to record the range and permission of private domains. With vm_range, PetaMem does not need to tag individual memory allocations. PetaMem can lookup the access permission to an address by accessing the corresponding vm_range object. The PetaMem API guarantees that all memory allocations in private domains must happen in the registered memory region. We describe how PetaMem detects memory references across domains next.

4.5.6 Enforcing Memory Isolation

Our prototype supports both inter-process isolation for PM processes and intra-process isolation for PMthreads. Since it is possible for a PM process to access an AMS that is created by another PM process in the system by natively switching memory mappings, the AMS secure switch gate needs to enforce that a PM process without privileges cannot couple with a privileged AMS. PetaMem maintains system-wide metadata for each AMSes to describe the privilege mapping between PM processes and AMSes. Each AMS has a petamem_struct in the kernel that records the creator and a list of PM processes allowed to couple with it. Before the PetaMem memory pager swaps in a requested AMS for a PM process, that process must pass the privilege checks performed by the AMS secure switch gate in the kernel. Once a PM process couples an AMS with its address space, PetaMem provides another level of privilege separation within the AMS. Figure 4.10 shows how the PetaMem pager intercepts

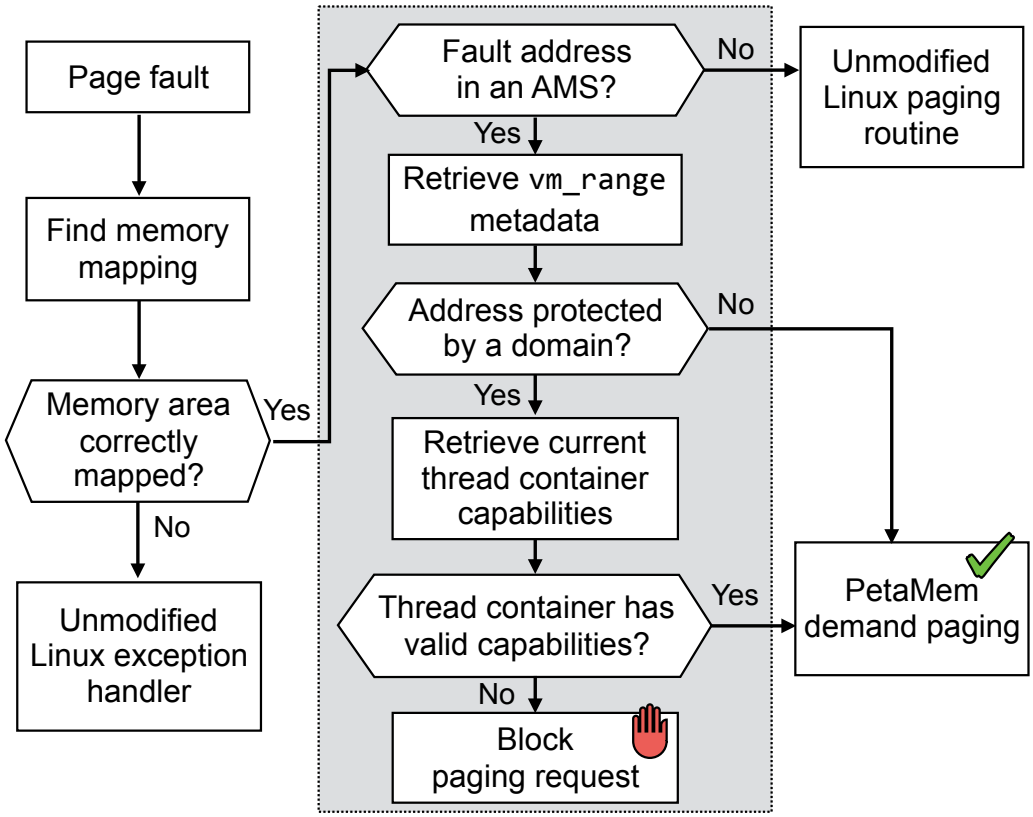

Figure 4.10.: PetaMem pager privilege checks for intra-process isolation. The additional privilege checks are marked in the shaded area.

invalid memory references by performing additional privilege checks for PMthreads in the page fault handler. The red-black tree for vm_range (cf. Section 4.5.2) allows the reference monitor of PetaMem to quickly intercept invalid memory references at page faults.

4.5.7 Recovering from Failures

When a PM process calls petamem_main_init (cf. Figure 4.3) to initialize the PetaMem environment, the crash recovery engine creates a crash entry in the global crash table for the PM process using the complete path of the program binary. When

exiting the system, the PM process calls petamem_main_finalize to request the recovery engine to remove the PM process's entry from the crash table. If the PM process crashes in between these two API calls, the entry will remain in the crash table, leading the recovery engine to decide that the process has crashed in the previous execution. When recovering a crashed PM process, the crash recovery engine coordinates with the kernel space PetaMem metadata manager to locate all metadata for the crashed AMSes. With the metadata, the PetaMem memory pager can identify the memory pages in the crashed AMSes and switch in the pdg_ts when the PM process re-couples with those AMSes.

4.6 Evaluation

4.6.1 Synopsis and Setup

We evaluate PetaMem to answer the following questions: (1) How efficient (EF) is PetaMem when accessing different address spaces (LMS) compared to traditional multi-processing or memory management syscalls (UPI)? (2) Can PetaMem intercept unprivileged memory references to ensure memory isolation (MI)? (3) How significant is fast recovery (CR)?

We ran experiments on a machine with an Intel Xeon E5-4655 (64 cores@2.5GHz) and 512GB RAM to study PetaMem's ability to access large memory. Our prototype successfully passed all stress tests in LTP's [48] runltp script that systematically tests major kernel subsystems.

4.6.2 Performance: Sequential Access

We use the STREAM benchmark [115] from the HPCC benchmark suite to study the performance of PetaMem when accessing large datasets with strong locality. STREAM measures the computation rate of vector style applications that access datasets much larger than the available cache on a system. STREAM allocates three

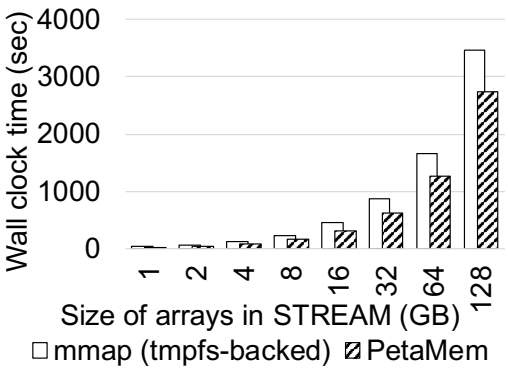

Figure 4.11.: STREAM performance. Lower is better.

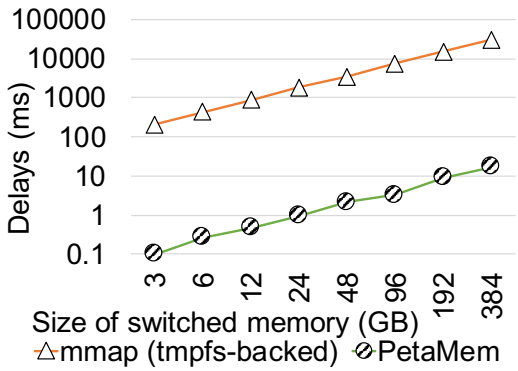

Figure 4.12.: Memory switch delays. Lower is better.

data arrays and performs copy, add, and multiply operations in a loop. We evaluated two versions of STREAM:

STREAM$_{\textbf{mmap}}$, the single-threaded vanilla version of the benchmark except where we moved the data arrays from the global region to heap for large memory allocations. **STREAM$_{\textbf{PetaMem}}$** leverages PetaMem to store datasets in AMSes and efficiently switches between the AMSes.

To study large memory, we choose an array size from 1GB to 128GB for each array. We partitioned each array into multiple 1GB sub-arrays. Both versions of STREAM load the sub-arrays one after another and perform updates until all sub-arrays are updated. The data access pattern has strong locality that favors mmap as the memory switching frequency is minimal (i.e., at most 128 times). However, although mmap notably performs well when few to no changes in memory mappings are involved, Figure 4.11 shows that STREAM$_{\text{PetaMem}}$ is still 1.3× faster than STREAM$_{\text{mmap}}$ because STREAM$_{\text{mmap}}$ is three orders of magnitude slower than STREAM$_{\text{PetaMem}}$ when switching memory mappings (Figure 4.12). Such limitation could greatly impact the performance of applications that have random access patterns, which we discuss next.

4.6.3 Performance: Random Access

We use the GUPS benchmark [100] that is specifically designed for profiling the memory architecture of a system to demonstrate the performance, scalability, isolation ability, and recoverability of PetaMem. GUPS allocates a large logical table in memory and randomly updates the elements in the table. The large table is divided into a number of windows. Within each window, GUPS performs a set of updates at random locations. Each update is a read-modify-write operation on a long unsigned integer. After GUPS finishes a set of updates in one window, it randomly chooses the next window to mutate the elements and repeat the procedure. Originally GUPS requires the maximum table size to be no larger than half the system memory. The memory distribution of the logical table can be implemented in various ways. We evaluate the performance of GUPS using three different memory models:

$\mathbf{GUPS_{MPI}}$ is the vanilla version of the benchmark that uses multiple processes to parallelize the update operations using the OpenMPI framework. All processes are connected in an N-dimensional hypercube. Each process holds a distinct window in its process address space and performs updates to its local table. Each process communicates with its neighbors to coordinate the execution progress before processing the next update set.

$\mathbf{GUPS_{mmap}}$ uses the traditional mmap/munmap syscalls to logically extend its process address space. For each window, $\text{GUPS}_{\text{mmap}}$ calls mmap to read a portion of the global table, then performs updates. To mutate elements in the next window, $\text{GUPS}_{\text{mmap}}$ needs to call munmap for the current window and calls mmap again to load another window into its process address space.

$\mathbf{GUPS_{PetaMem}}$ leverages PetaMem using a single process to efficiently switch between different windows without modifying the memory mappings and their underlying PTEs. Each window is represented as an AMS in the system, enabling $\text{GUPS}_{\text{PetaMem}}$ to dynamically couple with and decouple from a window for fast updates.

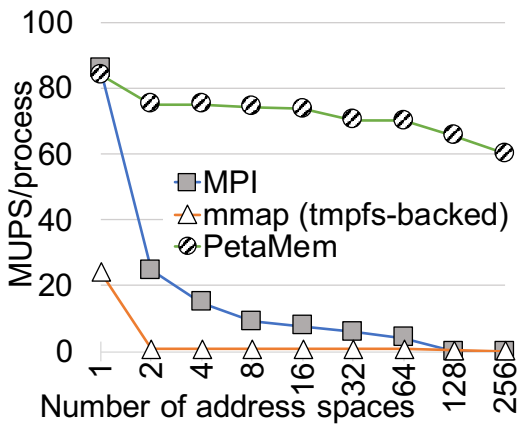

Figure 4.13.: GUPS throughput.
Higher is better.

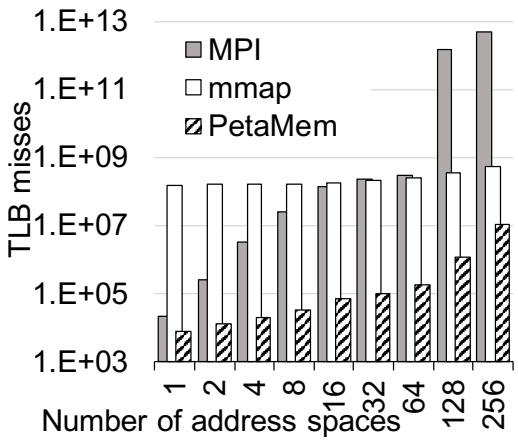

Figure 4.14.: GUPS TLB misses.
Lower is better.

We used GUPS_{MPI} from the HPCC benchmark suite and implemented $\text{GUPS}_{\text{mmap}}$ and $\text{GUPS}_{\text{PetaMem}}$. Figure 4.13 shows the GUPS performance, averaged over 10 runs, for the three memory models. The size of the global table ranges from 1GB to 256GB (half the system memory). We set the window size to 1GB to evaluate the cost of using multiple address spaces (up to 256) and report millions of updates per process on the y-axis. Following the official GUPS benchmark, the runtime measurement for all three memory models only considers the execution time of the update loop without initialization and cleanup.

When using one address space, GUPS_{MPI} and $\text{GUPS}_{\text{PetaMem}}$ perform equally well since no address space switches were needed. However, GUPS_{MPI} becomes extremely expensive when using more than two processes (EF). The performance bottleneck denotes the classic scalability limitation of multi-processing caused by heavy resource usage and expensive synchronization cost. Therefore, such multi-processing is less favorable for memory-hungry applications that require multiple address spaces. Our experiment shows that mmap/munmap is inefficient for logically extending process memory. In the update loop, $\text{GUPS}_{\text{mmap}}$ has to call mmap to create a mapping for every window. Such frequent memory manipulation in the kernel space makes

$\text{GUPS}_{\text{mmap}}$ scale poorly (or not at all) compared to GUPS_{MPI} and $\text{GUPS}_{\text{PetaMem}}$ in all cases. Increasing the number of address spaces only exacerbates the bottleneck of $\text{GUPS}_{\text{mmap}}$ (EF).

Discussion. Since TLB misses can lead to measurable performance degradation, we study the TLB misses of GUPS_{MPI}, $\text{GUPS}_{\text{mmap}}$, and $\text{GUPS}_{\text{PetaMem}}$ to unveil the impact caused by translation misses and present the results in Figure 4.14. When using a single window, $\text{GUPS}_{\text{mmap}}$ incurs most TLB misses because $\text{GUPS}_{\text{mmap}}$ needs to mmap the window to its address space in the update routine, even when there's only a single window. GUPS_{MPI} exhibits the second most TLB misses due to the additional memory access to set up the multi-processing environment. Both GUPS_{MPI} and $\text{GUPS}_{\text{PetaMem}}$ initialize the global table before entering the update loop. $\text{GUPS}_{\text{PetaMem}}$ has the smallest number of TLB misses because PetaMem does not invoke any MPI calls to use multi-processing.

As the number of address spaces increases, the number of TLB misses grows for all three memory models of GUPS. However, once the number of address spaces exceeds the number of cores, GUPS_{MPI} starts to thrash the memory hierarchy, causing GUPS_{MPI} to exhibit most TLB misses among all three memory models when using more than 64 address spaces. For $\text{GUPS}_{\text{mmap}}$, the heavy use of mmap and munmap causes the kernel to constantly change the process's mapping between physical and virtual memory. As a result, the number of TLB misses of $\text{GUPS}_{\text{mmap}}$ increases with the number of address spaces. When more windows are used, the number of TLB misses of $\text{GUPS}_{\text{PetaMem}}$ slightly increases due to higher degree of random memory access. Since PetaMem flushes only the TLB entries in AMSes when switching memory spaces, $\text{GUPS}_{\text{PetaMem}}$ does not pollute the caches like $\text{GUPS}_{\text{mmap}}$ when manipulating mappings, nor does it incur global TLB flushing like GUPS_{MPI}. Therefore, $\text{GUPS}_{\text{PetaMem}}$ outperforms the other two memory models.

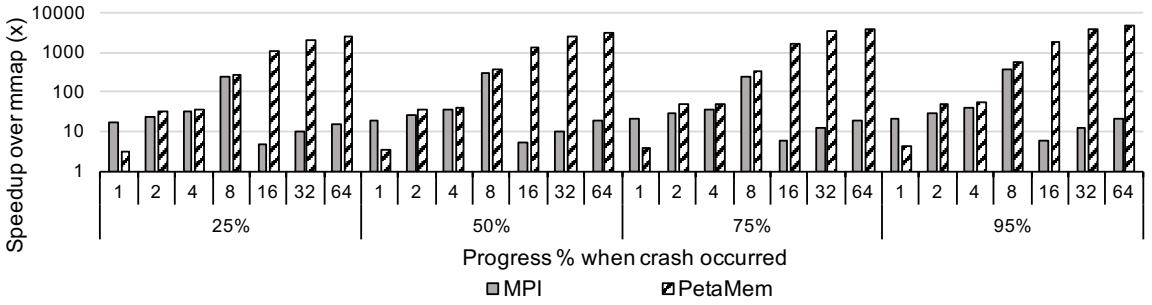

Figure 4.15.: GUPS recovery speedup. Results are averaged over 10 runs. Higher is better.

4.6.4 Application Recovery Speedup

To evaluate the benefits of providing recovery support to programs, we repeated the experiment in Section 4.6.3 and introduced crash points using a machine with an Intel i7-4790 (8 cores@2.8GHz) and 16GB RAM, showing that PetaMem does not rely on large memory to run correctly (PI). We crashed each of GUPS_{MPI}, $\text{GUPS}_{\text{mmap}}$, and $\text{GUPS}_{\text{PetaMem}}$ when they reached 25%, 50%, 75%, and 95% of the total execution progress by calling abort to simulate the abnormal termination of a process. All three versions of GUPS then had to restart and complete the execution. We made $\text{GUPS}_{\text{mmap}}$ able to recover from crash failures by periodically checkpointing the execution state using FS calls. Note that $\text{GUPS}_{\text{mmap}}$ uses tmpfs for memory-like access speed so the kernel does not incur storage device overheads. GUPS_{MPI} has to restart from the beginning and redo all the computation again as the OpenMPI framework does not yet provide an easy way to completely recover the global execution state across all processes. The recovery engine of PetaMem tracks the execution state of $\text{GUPS}_{\text{PetaMem}}$ and allows the system to locate the metadata and page tables from the previous run, enabling $\text{GUPS}_{\text{PetaMem}}$ to couple with the last window from the previous run and continue execution from the last checkpoint (CR).

We plot the speedup over $\text{GUPS}_{\text{mmap}}$ for crash recovery in Figure 4.15. The x-axis shows the percentage of completed progress when a crash happens in major

ticks and the number of address spaces in minor ticks. We normalize the execution times to $\text{GUPS}_{\text{mmap}}$, which is the slowest memory model in all three versions of the GUPS benchmark. When using a single window, GUPS_{MPI} outperforms $\text{GUPS}_{\text{PetaMem}}$ because the recovery engine introduces additional CPU cycles to set up and maintain the execution state. With two or four windows, $\text{GUPS}_{\text{PetaMem}}$ is slightly faster than GUPS_{MPI} because the additional cycles spent by the recovery engine pay off and help $\text{GUPS}_{\text{PetaMem}}$ continue execution from the last checkpoint. Once the number of windows exceeds the number of cores on the system, GUPS_{MPI} slows down to almost the speed of $\text{GUPS}_{\text{mmap}}$, which reflects the finding we showed in Figure 4.13. $\text{GUPS}_{\text{PetaMem}}$ outperforms GUPS_{MPI} significantly when using more than 8 windows.

With the help from the crash recovery engine in PetaMem, we only need less than 20 lines of code to make $\text{GUPS}_{\text{PetaMem}}$ tolerate crash failures (UPI). We allocate the progress counters and the global table in an AMS using petamem_malloc, which automatically tracks the address, name, and size of the variable in a memory allocation log. Before a normal execution, $\text{GUPS}_{\text{PetaMem}}$ checks if the program has crashed in the previous execution by calling petamem_crashed, and invokes petamem_recover and petamem_recover_var to retrieve all PetaMem metadata and bindings to persistent variables (cf. Figure 4.6), enabling $\text{GUPS}_{\text{PetaMem}}$ to correctly couple with the last window and resume execution.

$\text{GUPS}_{\text{PetaMem}}$ uses petamem_checkpoint to explicitly checkpoint the execution progress and notifies the kernel to flush the CPU caches to main memory after finishing every update set. After the API call returns, the PetaMem metadata and the page tables are persisted in main memory. Since the kernel does not clean up PetaMem resources until explicitly instructed to, $\text{GUPS}_{\text{PetaMem}}$ can resume execution by reusing the metadata and memory pages from the last checkpoint. In the event of power failure, using NVM as main memory will allow $\text{GUPS}_{\text{PetaMem}}$ to resume execution without restarting from scratch. The experiment shows the potential for fast failure recovery.

4.6.5 Enforcement of Isolation

In PetaMem, the AMS secure switch gate vets every AMS coupling request issued by a PM process and every page fault triggers the privilege checks as depicted in Figure 2.5. Since we ran all the experiments with privilege enforcement, the performance numbers reported herein already include the overheads of additional privilege checks.

Using the PetaMem API introduced in Figure 4.3, we implemented a security policy allowing only $\text{GUPS}_{\text{PetaMem}}$ to couple with the AMSes that store the partitioned tables of $\text{GUPS}_{\text{PetaMem}}$. No other processes should be able to couple with those AMSes. To test the AMS secure switch gate in PetaMem, we launched a separate PM process as a cross-process attacker that attempted to couple with those AMSes. With the $\text{GUPS}_{\text{PetaMem}}$ not specifying any privileges for other PM processes to access its AMSes (i.e., distrusting other PM processes by default), the AMS secure switch gate successfully blocked all unprivileged coupling requests issued by the attacker PM process (MI).

To verify the enforcement of user-defined memory boundaries for PMthreads with the intra-process isolation engine in PetaMem, we conducted an experiment similar to $\text{GUPS}_{\text{PetaMem}}$ by creating a privileged domain in a special AMS. We randomly chose an AMS as the special AMS and allocated a privileged domain within the special AMS, then put a partition of the global table into the privileged domain. In our experiment, we implemented a security policy that disallows the PMthread of $\text{GUPS}_{\text{PetaMem}}$ to access the privileged domain by isolating the PMthread in a PM container. Note that the PMthread can couple with the special AMS without triggering exceptions. However, when the PMthread of $\text{GUPS}_{\text{PetaMem}}$ attempted to access the partition of the table stored in the privileged domain, the reference monitor in PetaMem successfully blocked the paging request issued by the unprivileged PMthread (MI).

4.7 Conclusion

We have presented the design, implementation, and evaluation of PetaMem, a fast and scalable memory architecture. PetaMem promotes the simple abstraction of an *autonomous memory space* (AMS), which strives for controlled (de)coupling in space and robust (de)coupling in time of processes and memory spaces. This enables applications to access memory beyond process (LMS) boundaries with strong isolation (MI) and crash recovery support (CR). Our evaluation shows that PetaMem allows applications to access multiple address spaces with simple memory instructions (UPI) efficiently. Compared to the traditional process-centric architecture, its crash failure recovery engine in PetaMem enables applications to resume execution after crash failures with drastic speedup (EF) over traditional memory management and file system interface calls. The isolation engine in PetaMem helps applications isolate software components in large memory pools with user-defined boundaries (PI). The privilege enforcement is implemented using existing HW protection mechanisms, providing strong guarantees. In summary, until HW solutions catch up with the rapid growth in data size and memory capacity, PetaMem constitutes a practical pure software-based solution for memory-driven computing. In the future we plan to investigate support for other failure/threat models.

5 CONCLUSION

This dissertation has advanced the design and implementation of existing memory subsystems to enhance the security, consistency, and scalability of applications through a software approach. We presented three practical memory subsystems; SMVs, a comprehensive framework for concurrent threads to achieve selective memory isolation, NVthreads, a user-space library for programmers to easily leverage non-volatile memory, and PetaMem, a novel architecture for data-intensive applications to freely access large amount of memory with isolation and crash recovery support.

We showed that SMVs can improve the security of multithreaded applications by protecting software components from poor memory isolation in today's process memory model. With SMVs, programmers can now structure the intrinsically shared process memory into a dynamic set of memory protection domains using the well-defined API. The kernel-level privilege enforcement in SMVs provides fast and accurate privilege checks without impacting the performance of software. We evaluated our solution through multithreaded benchmarks with complex memory interaction and demonstrated its practicability by porting popular web servers and Firefox web browser for security.

To ensure the consist states for multithreaded applications running on non-volatile memory systems, our solution, NVthreads, leverages synchronization operations to determine consistency semantics and coarse-grained page-level tracking to manage persistent data. Our extensive evaluation showed that NVthreads significantly reduces the performance gap between unmodified applications and their crash tolerant counterparts. With NVthreads, programmers can now easily transition to the non-volatile memory era.

For scalability, our memory subsystem PetaMem provides controlled space and time (de)coupling to enable data-intensive applications to access large amount of

memory with simple memory operations. PetaMem exposes an API which enables programmers to freely access memory in different address spaces efficiently. PetaMem puts together the security benefits of SMVs and the recoverability support from NVthreads for memory-centric computing, making large memory pool more reliable and practical. The evaluation showed that PetaMem significantly outperforms the traditional process-centric architecture.

REFERENCES

REFERENCES

[1] Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. The missing memristor found. *Nature*, 453(7191):80–83, May 2008.

[2] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM.

[3] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, CCS '16, New York, NY, USA, 2016. ACM.

[4] Jerome H. Saltzer. Protection and Control of Information Sharing in Multics. In *Proceedings of the Fourth ACM Symposium on Operating System Principles*, SOSP '73, pages 119–, New York, NY, USA, 1973. ACM.

[5] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept 1975.

[6] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.

[7] David Brumley and Dawn Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[8] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[9] Raoul Strackx, Pieter Agten, Niels Avonds, and Frank Piessens. Salus: Kernel Support for Secure Process Compartments. *EAI Endorsed Transactions on Security and Safety*, 15(3), 1 2015.

[10] Jun Wang, Xi Xiong, and Peng Liu. Between Mutual Trust and Mutual Distrust: Practical Fine-grained Privilege Separation in Multithreaded Applications. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 361–373, Santa Clara, CA, July 2015. USENIX Association.

[11] AppArmor. `https://wiki.ubuntu.com/AppArmor`.

[12] SELinux. https://wiki.centos.org/HowTos/SELinux.

[13] SECure COMPuting with filters. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.

[14] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Everything You Want to Know About Pointer-Based Checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 190–208, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[15] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society.

[16] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association.

[17] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical Capabilities for UNIX. In *USENIX Security 2010*, pages 29–46, 2010.

[18] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 191–206, New York, NY, USA, 2015. ACM.

[19] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making Information Flow Explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[20] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information Flow Control for Standard OS Abstractions. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 321–334, New York, NY, USA, 2007. ACM.

[21] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical Fine-grained Decentralized Information Flow Control. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 63–74, New York, NY, USA, 2009. ACM.

[22] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[23] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[24] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, May 2009.

[25] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. Language-independent Sandboxing of Just-in-time Compilation and Self-modifying Code. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 355–366, New York, NY, USA, 2011. ACM.

[26] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[27] Ciarán Bryce and Chrislain Razafimahefa. An Approach to Safe Object Sharing. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 367–381, New York, NY, USA, 2000. ACM.

[28] Kiyokuni Kawachiya, Kazunori Ogata, Daniel Silva, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Cloneable JVM: A New Approach to Start Isolated Java Applications Faster. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 1–11, New York, NY, USA, 2007. ACM.

[29] Andrew C. Myers. JFlow: Practical Mostly-static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.

[30] Adrian Mettler, David Wagner, and Tyler Close. Joe-E: A Security-Oriented Subset of Java. In *Network and Distributed Systems Symposium*, NDSS 2010. Internet Society, 2010.

[31] Winnie Cheng, Dan R. K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for Usable Information Flow Control in Aeolus. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 12–12, Berkeley, CA, USA, 2012. USENIX Association.

[32] Kevin J. Hoffman, Harrison Metzger, and Patrick Eugster. Ribbons: A Partially Shared Memory Programming Model. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 289–306, New York, NY, USA, 2011. ACM.

[33] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 315–328, New York, NY, USA, 2008. ACM.

[34] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158, Washington, DC, USA, 2010. IEEE Computer Society.

[35] Raoul Strackx and Frank Piessens. Fides: Selectively Hardening Software Application Components Against Kernel-level or Process-level Malware. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 2–13, New York, NY, USA, 2012. ACM.

[36] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative Technology for CPU based Attestation and Sealing. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, New York, NY, USA, 2013. ACM.

[37] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 11:1–11:1, New York, NY, USA, 2013. ACM.

[38] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM.

[39] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 225–240, Berkeley, CA, USA, 2008. USENIX Association.

[40] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. CODOMs: Protecting Software with Code-centric Memory Domains. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 469–480, Piscataway, NJ, USA, 2014. IEEE Press.

[41] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 457–468, Piscataway, USA, 2014. IEEE Press.

[42] Anil Kurmus and Robby Zippel. A Tale of Two Kernels: Towards Ending Kernel Hardening Wars with Split Kernel. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1366–1377, New York, NY, USA, 2014. ACM.

[43] Cherokee Web Server. http://cherokee-project.com/.

[44] Same-origin Policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[45] GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/.

[46] Valgrind. http://valgrind.org/.

[47] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 361–372, Piscataway, NJ, USA, 2014. IEEE Press.

[48] Linux Test Project. http://sourceforge.net/projects/ltp/.

[49] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[50] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 40–40, Berkeley, CA, USA, 2012. USENIX Association.

[51] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic Generation of Data-Oriented Exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 177–192, Washington, D.C., August 2015. USENIX Association.

[52] Interesting stats based on Alexa Top 1,000,000 Sites. http://httparchive.org/interesting.php.

[53] Gregor Wagner, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Compartmental Memory Management in a Modern Web Browser. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 119–128, New York, NY, USA, 2011.

[54] Multiprocess Firefox. https://developer.mozilla.org/en-US/Firefox/Multiprocess_Firefox.

[55] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 117–128, New York, NY, USA, 2000. ACM.

[56] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical Persistence for Multi-threaded Applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 468–482, New York, NY, USA, 2017. ACM.

[57] Intel and Micron produce breakthrough memory technology. http://newsroom.intel.com/docs/DOC-6713, 2015.

[58] SanDisk and HP launch partnership to create memory-driven computing solutions. http://www8.hp.com/us/en/hp-news/press-release.html?id=2099577, 2015.

[59] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[60] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. In *IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015*, pages 1–10, 2015.

[61] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. ACM.

[62] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *SIGPLAN Not.*, 46(3):105–118, March 2011.

[63] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[64] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[65] Christian Bienia and Kai Li. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[66] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[67] B. Dieny, R. Sousa, G. Prenat, and U. Ebels. Spin-dependent phenomena and their implementation in spintronic devices. In *International Symposium on VLSI Technology, Systems and Applications*, VLSI '08, pages 70–71. IEEE, 2008.

[68] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[69] Adam Welc, Antony L. Hosking, and Suresh Jagannathan. Transparently reconciling transactions with locking for java synchronization. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 148–173, Berlin, Heidelberg, 2006. Springer-Verlag.

[70] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 427–442, New York, NY, USA, 2016. ACM.

[71] Ellis Giles, Kshitij Doshi, and Peter J. Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *MSST*, pages 1–14. IEEE Computer Society, 2015.

[72] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.

[73] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight recoverable virtual memory. *ACM Trans. Comput. Syst.*, 12(1):33–57, February 1994.

[74] David E. Lowell and Peter M. Chen. Free transactions with rio vista. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 92–101, New York, NY, USA, 1997. ACM.

[75] Seth J. White and David J. DeWitt. Quickstore: A high performance mapped object store. *The VLDB Journal*, 4(4):629–673, October 1995.

[76] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: Good, fast, cheap persistence for c++. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, OOPSLA '92, pages 145–147, New York, NY, USA, 1992. ACM.

[77] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, New York, NY, USA, 2011. ACM.

[78] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[79] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 401–410, New York, NY, USA, 2012. ACM.

[80] Oren Laadan and Jason Nieh. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 25:1–25:14, Berkeley, CA, USA, 2007. USENIX Association.

[81] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[82] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 18–18, Berkeley, CA, USA, 1995. USENIX Association.

[83] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From aries to mars: Transaction support for next-generation, solid-state drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 197–212, New York, NY, USA, 2013. ACM.

[84] Russell Sears and Eric Brewer. Stasis: Flexible transactional storage. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 29–44, Berkeley, CA, USA, 2006. USENIX Association.

[85] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[86] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. In *WWW7*, 1998.

[87] Stanford network analysis package. `http://snap.stanford.edu/snap`.

[88] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: Membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 44–54, New York, NY, USA, 2006. ACM.

[89] Tokyo Cabinet: a modern implementation of DBM. `http://fallabs.com/tokyocabinet/`.

[90] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 237–248, New York, NY, USA, 2013. ACM.

[91] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting Phase Change Memory As a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM.

[92] Hewlett Packard Labs. Memory Driven Computing. `https://www.labs.hpe.com/next-next/mdc`, 2017.

[93] Intel Corporation. Persistent Memory Programming. `http://pmem.io/nvml/`, 2017.

[94] Kimberly Keeton. Memory-Driven Computing. https://www.usenix.org/conference/fast17/technical-sessions/presentation/keeton, 2017.

[95] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[96] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[97] Intel Corporation. 5-Level Paging and 5-Level EPT. https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf, 2017.

[98] Cloudera Engineering Blog. The Newest Hadoop Framework for CDH Users and Developers. http://blog.cloudera.com/blog/2013/06/cloudera-search-the-newest-hadoop-framework-for-cdh-users-and-deve 2013.

[99] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Radixvm: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 211–224, New York, NY, USA, 2013. ACM.

[100] S. J. Plimpton, R. Brightwell, C. Vaughan, K. Underwood, and M. Davis. A Simple Synchronous Distributed-Memory Algorithm for the HPCC RandomAccess Benchmark. In *2006 IEEE International Conference on Cluster Computing*, pages 1–7, Sept 2006.

[101] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, pages 31–39, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[102] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. SpaceJMP: Programming with Multiple Virtual Address Spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 353–368, New York, NY, USA, 2016. ACM.

[103] LWN.net: News from the source. Introduce first class virtual address spaces. https://lwn.net/Articles/717069/, 2017.

[104] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. Nooks: An Architecture for Reliable Device Drivers. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 102–107, New York, NY, USA, 2002. ACM.

[105] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 49–64, GA, 2016. USENIX Association.

[106] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[107] X. Wu and A. L. N. Reddy. Scmfs: A file system for storage class memory. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, Nov 2011.

[108] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. Architecture Support for Single Address Space Operating Systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 175–186, New York, NY, USA, 1992. ACM.

[109] J. Chase, H. Levy, M. Baker-Harvey, and E. Lazowska. Opal: a single address space system for 64-bit architecture address space. In *[1992] Proceedings Third Workshop on Workstation Operating Systems*, pages 80–85, Apr 1992.

[110] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007.

[111] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. ACM.

[112] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. The Case for Writing a Kernel in Rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, pages 1:1–1:7, New York, NY, USA, 2017. ACM.

[113] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society.

[114] Doug Lea. dlmalloc: A Memory Allocator. `http://g.oswego.edu/dl/html/malloc.html`, 2000.

[115] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.