DSM64: A DISTRIBUTED SHARED MEMORY SYSTEM IN USER-SPACE

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Stephen Holsapple

May 2012

COMMITTEE MEMBERSHIP

TITLE:	DSM64: A Distributed Shared Memory System In User-space
AUTHOR:	Stephen Holsapple
DATE SUBMITTED:	May 2012
COMMITTEE CHAIR:	John Bellardo, Ph.D.
COMMITTEE MEMBER:	Chris Lupo, Ph.D.
COMMITTEE MEMBER:	Phillip Nico, Ph.D.

Abstract

DSM64: A Distributed Shared Memory System In User-space

Stephen Holsapple

This paper presents DSM64: a lazy release consistent software distributed shared memory (SDSM) system built entirely in user-space. The DSM64 system is capable of executing threaded applications implemented with `pthreads` on a cluster of networked machines without *any* modifications to the target application. The DSM64 system features a centralized memory manager [1] built atop Hoard [2, 3]: a fast, scalable, and memory-efficient allocator for shared-memory multiprocessors.

In my presentation, I present a SDSM system written in C++ for Linux operating systems. I discuss a straight-forward approach to implement SDSM systems in a Linux environment using system-provided tools and concepts available entirely in user-space. I show that the SDSM system presented in this paper is capable of resolving page faults over a local area network in as little as 2 milliseconds.

In my analysis, I present the following. I compare the performance characteristics of a matrix multiplication benchmark using various memory coherency models. I demonstrate that matrix multiplication benchmark using a LRC model performs orders of magnitude quicker than the same application using a stricter coherency model. I show the effect of coherency model on memory access patterns and memory contention. I compare the effects of different locking strategies on execution speed and memory access patterns. Lastly, I provide a comparison of the DSM64 system to a non-networked version using a system-provided allocator.

Contents

List of Figures

Chapter 1

Introduction & Related Work

This paper presents DSM64: a lazy release consistent software distributed shared memory system built entirely in user-space. The DSM64 system is a general-purpose SDSM system that is capable of running *any* `pthread` application that is *properly labeled* with synchronization primitives. Applications that meet these criteria can be run on clusters of machines without *any* modifications to the original application binary. DSM64 acheives this by providing a replacement threading interface that is compatible with `pthread`'s own iterface.

The DSM64 system is a general-purpose SDSM system build atop a modified version of Hoard [2]: a fast, scalable, and memory-efficient allocator for shared-memory multiprocessors. The DSM64 system makes minimal changes to the Hoard library by adding a refined *heap layer* which maintains a global address space with standard virtual memory management techniques. In doing so, the DSM64 system is a drop-in replacement for the system-provided memory allocator that allows `pthread` applications to automatically realize the advantages of a distributed system at no additional cost.

As shown in this paper, SDSM systems are implemented in various ways, sustaining a rich environment for different types of designs. As suggested in [6], SDSM solutions are often tailored to the application, making general-purpose SDSM systems a very difficult problem. As identified in survey work done by Nitzberg in [4], there are several fundamental design attributes that designers typically focus on. These attributes are listed and summarized below. The remainder of this section is dedicated to a discussion of these design features as they relate to other systems. In the following sections, special focus is given to systems that inspired the work done in this paper.

Structure & Granularity These two design items are closely related. The structure in a DSM system refers to the layout of the shared data in memory [4]. Most SDSM systems do not structure memory (these systems leave memory as a linear collection of bytes), but some structure the data as objects, language objects or as associative memory. The granularity of a SDSM system refers to the size of a sharable memory unit: a byte, word, page or data structure [4].

Coherence Semantics Perhaps the most defining characteristic of a SDSM system is the memory coherence model that it exhibits, and extensive research has been done just in just this area [7]. Memory coherence semantics refer to how concurrent memory updates are propagated throughout the distributed system. Many different types of coherence protocols exist in the wild [4]. In the field of SDSM systems, the words *coherence* and *consistency* are oftentimes used interchangeably.

Synchronization Depending on the coherence protocol used in a SDSM system, synchronization may or may not be important. For typical SDSM systems

that implement a weaker coherence model, synchronization becomes an important problem: does the SDSM system support synchronization primitives like locks, barriers or semaphores? If the system does require synchronization, how does it enforce these rules in the application program? Does the application require source code annotations? Does the system use a central synchronization manager? These are all questions that refer to a SDSM system's synchronization methods.

Memory Contention Memory contention refers to concurrent access to the same acquirable unit of shared memory. Memory contention typically involves locking a sharable memory unit and degrades the performance of an application. This attribute – referred to as memory or variable contention – is a function of a SDSM system's structure, granularity and coherence protocols. The implementation of all of these design attributes determines how a given SDSM system minimizes memory contention. However, variable contention also heavily depends on the implementation and memory access patterns of the target-application.

Lastly, several seminal distributed shared memory systems are analyzed in regards to these characteristics and their respective contributions to the work done in this project. A section describing the different types of existing shared memory systems will help identify where DSM64 stands in comparison to others. Additionally, because the work discussed in this document will be implemented entirely in user-space, a discussion of virtual memory management (VMM) techniques and existing solutions are explored to provide an appropriate background in memory management in SDSM systems.

1.1 Coherence Semantics

The most defining characteristic of a SDSM is the memory coherence model that it exhibits [7]. These protocols range from strict consistency to weak consistency and oftentimes hybrid coherency that seek to exploit the favorable attributes of a set of coherency protocols – as is the case with a very popular coherency protocol called release consistency.

This section provides a discussion of previously proposed memory consistency models as well as an efficient form of memory consistency used as a base model in DSM64 called release consistency. As in [5], the previously proposed memory consistency models are provided for completeness and uniformity in terminology, and allow for a deeper understanding of the requirements for RC. Readers familiar with historical memory coherence models may wish reference Figure 1.1 and skip to §1.1.5.

The remainder of this section depends on the following definitions of a memory access, originally defined by Debois in [8, 9] and elaborated on in [5]. In the following, P_x refers to a processor x. There are two types of memory accesses: $LOAD$ and $STORE$ operations. These operations are synonymous with memory read and write operations, respectively, and are used interchangably in this paper. A $LOAD$ by P_i is considered performed with respect to P_k at a point in time when the issuing of a $STORE$ to the same address by P_k cannot affect the value returned by the $LOAD$. A $STORE$ by P_i is considered performed with respect to P_k at a point in time when an issued $LOAD$ to the same address by P_k returns the value defined by this $STORE$ (or a subsequent $STORE$ to the same location). An access is performed when it is performed with respect to all processors. Additionally, a distinction between performed and globally per-

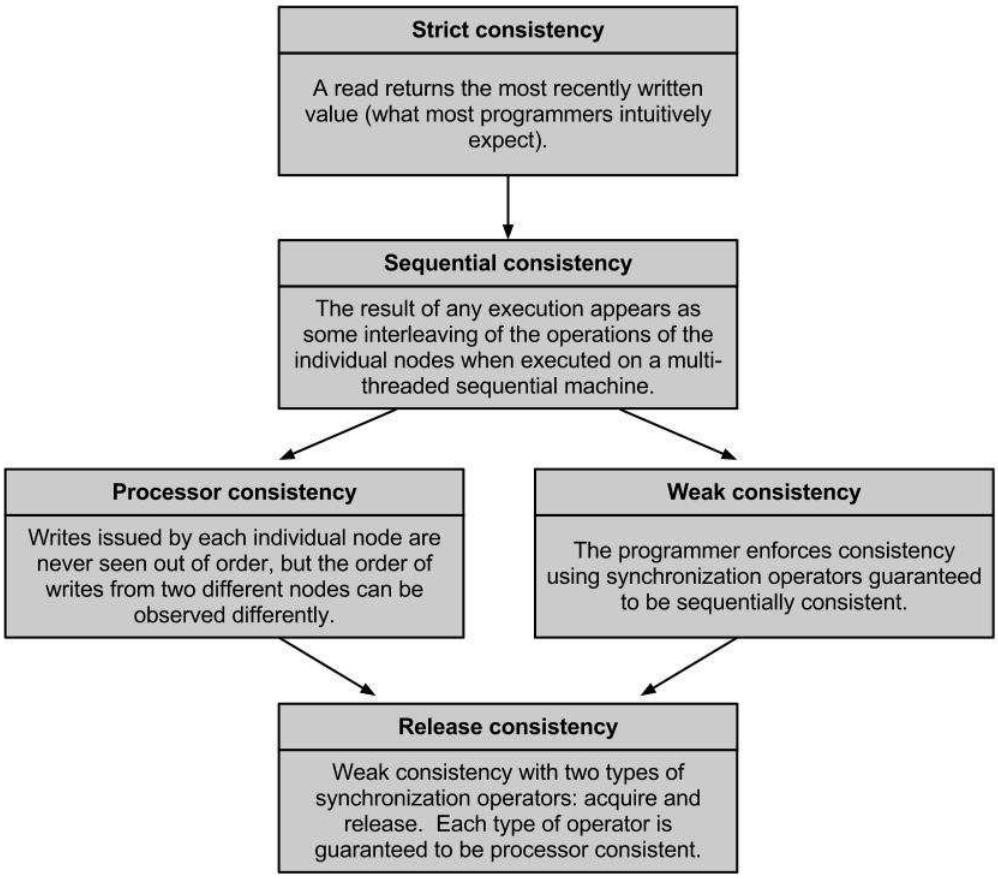

Figure 1.1: Intuitive definitions of memory coherence [4]. The arrows point from stricter to weaker consistencies.

formed *LOAD* accesses is necessary for architectures with non-atomic *STORE* operations. A *STORE* is atomic if the value stored becomes readable to all processors at the same time [5]. For SDSM systems, it is typically not the case that *STORE* operations are atomic unless special hardware is used.

1.1.1 Strict Consistency

In order to write programs that correctly execute on a platform, a programmer must understand the model the system uses to keep memory coherent. The most intuitive model programmers expect from a platform is strict consistency [10].

As identified by Li and Hudak in [1], a memory is coherent if the value returned by a read operation is always the value written by the most recent write operation to the same address. This definition of coherent memory is exactly what strict consistency guarantees: a memory read operation returns the most recently written memory value [4]. This memory consistency model exhibits the most restrictive rules and serves as a base rule in many of Lamport's formalisms in [11]. This coherency model is not used in SDSM systems whose focus is speed or efficiency because the restrictive rules necessitate serial reading and writing of data. The amount of overhead required to keep a memory coherent is too great in SDSM systems because messages must be passed over the network.

1.1.2 Sequential Consistency

On platforms with multiple execution paths programmers expect memory to be sequentially consistent (SC). In a SC model, separate processors are expected to be strictly coherent, but the order of memory access between those processes is defined by the programmer through some sort of synchronization primitive. A system is SC if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual process or appear in this sequence in the order specified by its program [11].

Dubois studied how one may order events to guarantee SC in [8] by formally demonstrating that ordering $LOAD$ and $STORE$ in a fashion that prevents simultaneous access guarantees correct execution of an application program. This makes sense to programmers familiar with multithreaded programming: synchronization primitives *must* be used to keep critical sections safe. The following

conditions were inspired by Dubois and formalized by Gharachorloo in [5].

Condition 1.1 (Sufficient Conditions for Sequential Consistency)**.**

(A) before a LOAD is allowed to perform with respect to any other processor, all previous LOAD accesses must be globally *performed and all previous STORE accesses must be performed, and,*

(B) before a STORE is allowed to perform with respect to any other processor, all previous LOAD accesses must be globally *performed and all previous STORE accesses must be performed.*

1.1.3 Processor Consistency

Goodman in [10] introduced the concept of processor consistency (PC) to relax the ordering of SC. PC requires that memory writes occur strictly consistent only on the executing processor. To another processor the memory events need not occur in a strictly consistent manner. Though this consistency model may incorrectly execute if the programmer assumes SC, Goodman showed that most applications give the same results because explicit synchronization primitives like mutex locks, barriers and semaphores are used [10]. Using formal definitions provided by Goodman in [10], Gharachorloo synthesized the following conditions for PC in [5]:

Condition 1.2 (Conditions for Processor Consistency)**.**

(A) before a LOAD is allowed to perform with respect to any other processor, all previous LOAD accesses must be performed, and,

(B) before a STORE is allowed to perform with respect to any other processor, all previous accesses (LOADs and STOREs) must be performed.

PC is the model that most application programmers expect on multicore machines. On a single core, memory reads and writes are expected to be sequential. However, the memory accesses between more than one core is less clear and race conditions may exist. By using synchronization primitives application programmers can order the events between processors. Processors may execute in any order as long as critial sections are executed sequentially.

1.1.4 Weak Consistency

When discussing weaker consistency (WC) model the discussion is usually limited to two things: correctness and performance. Correctness is important for showing a weaker consistency model is equivalent to a stricter consistency model in regards to a program's execution. The main purpose of weaker coherency models is performance [5]. Weaker consistency models can be derived by grouping competing and synchronized memory requests to synchronization points in the program [5]. In practice, this means delaying the ordering of memory events to some synchronization point – typically, a release or acquire event.

This type of coherency protocol provides a sufficient programming environment for programs that require synchronization – programs that require sequentually executing critical sections [7] – while maintaining a simple programming interface for the application programmer. If a program relies on critical sections to correctly execute, the programmer is responsible for ensuring that the memory written to by the critical sections is not read until all processors are done writing using synchronization primitives provided by the underlying system. This assumption is how WC models ensure correctness.

Additionally, it has also been shown in work such as [8] and [12] that these

weaker consistency models can be implemented invisibility to the application programmer. By implementing a compiler that invisibly enforces consistency rules a program written to adhere to a PC model (e.g., the application was written using synchronization primitives to ensure correctness of execution) can be further translated to work on weaker memory consistency models. Gharachorloo provided the following conditions for WC in [5]:

> **Condition 1.3** (Conditions for Weak Consistency)**.**
>
> *(A) before an ordinary LOAD or STORE access is allowed to perform with respect to any other processor, all previous synchronization accesses must be performed, and,*
>
> *(B) before a synchronization access is allowed to perform with respect to any other processor, all previous ordinary LOAD and STORE accesses must be performed, and,*
>
> *(C) synchronization accesses are sequentially consistent with respect to one another.*

Despite the weaker conditions that WC fulfills, WC is considered too strict for SDSM systems because equivalent models can be constructed that require fewer messages. For instance, some synchronization operations are used only to import or export information about the state of memory itself, as with the acquisition or release of a lock [7]. Weaker consistency models were synthesized to address the restrictive components of weaker consistency models in [5].

1.1.5 Release Consistency

Release consistency (RC) is a weak consistency model that further relaxes the restrictions of the WC discussed in §1.1.4 by overlapping release and acquire synchronization events. This additional change differentiates synchronization and

regular accesses to memory, which allow release consistent models to exploit access patterns to provide better performance. This model effectively creates a pipeline of release and acquire events.

Weaker consistency models like RC exploit information about the different types of memory access to improve performance. Figure 1.2 is a taxonomy of memory accesses a RC model uses. The remainder of this section is a discussion of the different types of labels and their significant contributions to RC.

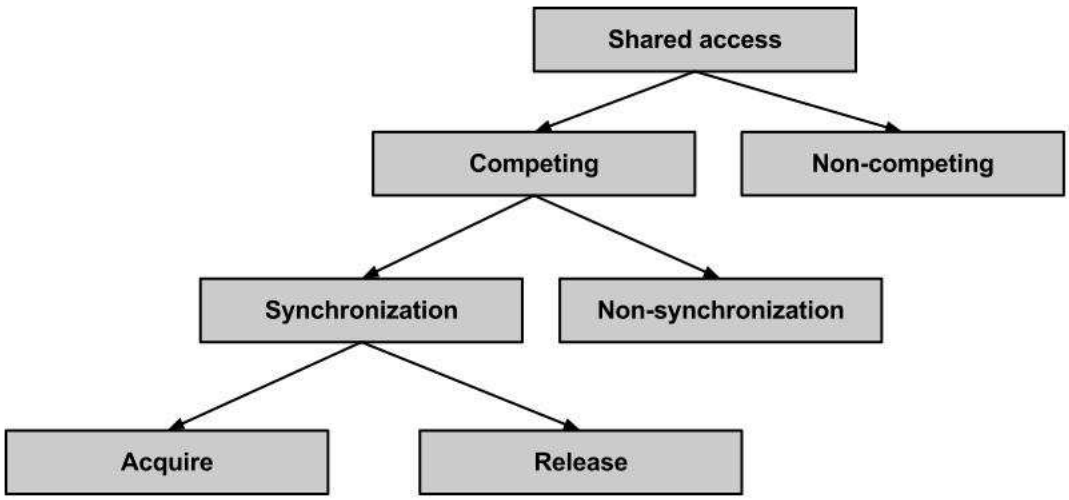

Figure 1.2: A categorization and label of shared memory writable accesses from [5]. The labels at the same level are disjoint, and a label at a leaf implies all its parent labels.

The first notion of competing accesses is described in [5] as the two processors accessing the same location in memory, with one of those accesses being a *STORE* operation. If these two operations are not ordered, a race condition exists and will effect the correctness of the application. This conflicting access is referred to as a competing memory access and must be corrected with some sort of synchronization mechanism.

Processors of a parallel program typically communicate through shared memory. For example, a producer may add data to a shared memory location for a

consumer to process. Modification to shared data structures is usually protected with lock and unlock operations to prevent competing memory accesses. All such accesses used to enforce ordering among processors are called synchronization accesses. Gharachorloo identifies two characteristics of synchronization accesses in [5]: (i) they are competing accesses, with one process writing and the other reading it; and (ii) they are frequently used to order conflicting accesses (i.e., make them non-competing). These types of accesses are used to mitigate the problems caused by conflicting memory accesses.

Synchronization accesses are further decomposed into acquire and release accesses. An acquire synchronization event is performed to gain access to a shared memory location, and a release synchronization event is performed to grant a request. These two types of accesses are equivalent to lock and unlock operations, respectively. Additionally, an acquire synchronization event is always associated with a $LOAD$, or read, and a release synchronization event is always associated with a $STORE$, or write.

The labels discussed in this subsection are synonymous with the following terms used in Figure 1.2: non-competing accesses are referred to as ordinary accesses, competing accesses are referred to as special accesses, and special accesses that require synchronization are referred to as synchronous accesses. As articulated in [5], these labels are used to mark an application's memory accesses until the application is considered *properly labeled*. A *properly labeled* program is one that can use a RC model and maintain equivalence with SC models. A model that is equivalent to another implies that the model maintains correctness.

Researchers have formalized the conditions for RC in works such as [5] and [13], which serve as a basis for other derived coherency models discussed later in this section:

Condition 1.4 (Conditions for Release Consistency)**.**

(A) before an ordinary LOAD or STORE access is allowed to perform with respect to any other processor, all previous acquire accesses must be performed, and,

(B) before a release access is allowed to perform with respect to any other processor, all previous ordinary LOAD and STORE accesses must be performed, and,

(C) special accesses are processor consistent with respect to one another.

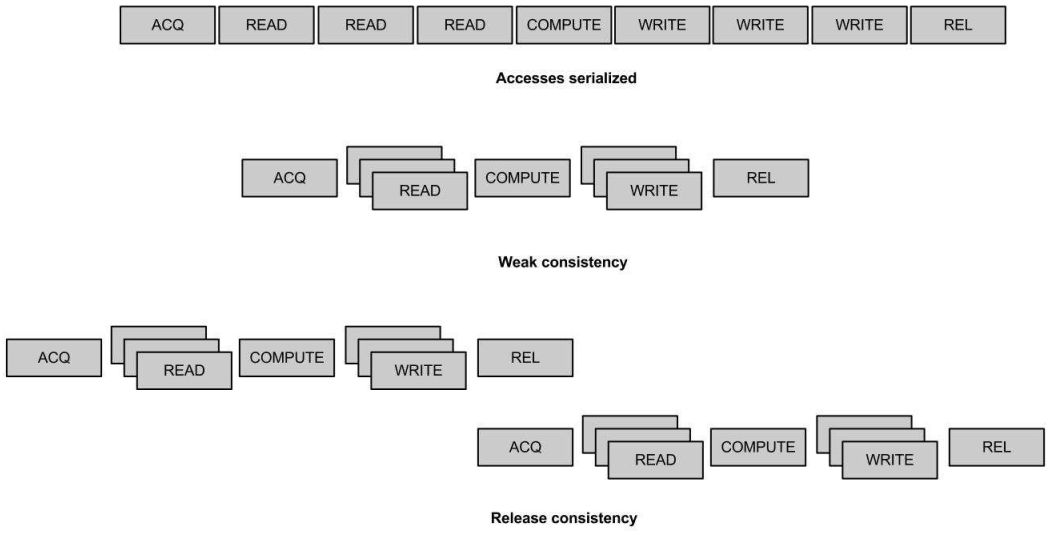

Figure 1.3: This figure demonstrates the capacity for RC protocols to perform quicker than stricter coherency protocols, as a processor need not wait for memory to be released before acquiring it.

Eager & Lazy Release Consistency

The two main variants of RC in literature are eager release consistency (ERC) and lazy release consistency (LRC). These two coherency models are used by the major of SDSM systems, such as Treadmarks in [14, 15], Munin in [13, 16, 17, 6]

and Shasta in [18]. Each of these derivatives obey the conditions for RC derived in [5], but treat communication differently. As the names suggest, ERC propagates memory accesses as they happen, whereas LRC delays propagation of memory accesses until some synchronization event occurs.

Eager Release Consistency ERC protocols delay propagating its modifications to shared data until a release synchronization access and is studied in [13]. At the time of a release, the ERC protocol propagates modifications to all other processors that cache the memory page. A naive implementation of an ERC protocol may propagate these changes at the release or send invalidate requests to processors working on the memory page. However, regardless of the implementation details of the ERC protocol, significant overhead is incurred by communicating changes at each release event. This model may degrade performance because changes are eagerly propagated to nodes that may not require the updated memory unit any longer.

Lazy Release Consistency LRC protocols delay propagating its modifications to shared data until an acquire synchronization access and is studied in [13]. At the time of an acquire, the LRC protocol propagates modifications to the requesting processor by piggybacking the changes to the memory page on a successful acquire message [13]. Optimizations also exist that can allow an acquiring processor to only request changes that it actually needs. This model optimally transfers memory because changes are lazily propagated to nodes only when they read or write the memory unit.

While RC remains one of the most viable solutions for memory coherence in SDSM systems, it has limitations like those measured in [19]. RC systems cannot compete with the speed of coupled processors because of the communication

overhead involved in maintaining consistency in a SDSM system. A harmonious balance between page size, processor speed and consistency protocol can be attained for SDSM systems, but application speedup is still limited. To mitigate communication overhead, a great number of customized RC protocols have been developed like eager invalidate, eager update, lazy invalidate, lazy update, as well as various hybrid protocols that mix and match the aforementioned protocols depending on consistency requirements.

1.2 Synchronization

Lamport declared that to correctly execute an application on a distributed system, the distributed system must understand how events on disjoint processes are ordered [20, 11]. Ordering the events in a SDSM system can be done in many ways, and how a SDSM system acheives synchronization is a defining characteristic of the system.

When one thinks about synchronization in a SDSM system, they're thinking of the how the target application synchronizes its address space for use among multiple processors. Thinking of this concept in terms of a popular threading interface, the `pthread` library, is useful in making sense of this. In a `pthread`-based application, one uses a series of routines to synchronize the `pthread`-based application. These functions may include `pthread_create`, `pthread_join`, `pthread_mutex_lock`, `pthread_mutex_unlock`, among others. Using these routines, one may synchronize access to the processor's memory so that critical sections are protected from concurrent access. The routines provided by the `pthread` library order the events in the system. Ultimately, an application programmer's goal is to yield a *properly labeled* application so that it may correctly execute on

the system.

The `pthread` library orders events in a single application running in a single address space. How one goes about ordering the events in a distributed system is similar. Existing SDSM systems use various methods to achieve this, including busy-wait and spin-lock solutions [21] to check on the status of lock and barrier variables. Systems like Munin in [16, 17] provide their own threading interfaces for the application to be written with. Using the provided interfaces, one can synchronize a distributed application in the same fashion as a `pthread`-based application.

Only *properly labeled* applications can correctly execute on a system that uses RC memory models. This is because of RC's reliance on synchronization events to propagate changes in the system. Consider a matrix multiplication application written using the `pthread` library. One may write a matix multplication application that executes correctly without any synchronization primitives by relying on prescribed thread workloads and ordering threads to write at prescribed offsets. This type of application executes correctly if run on a system where the address space exists on one physical machine, but is not a *properly labeled* program.

1.3 Structure & Granularity

Typically, virtual memory in a sequential computing machine is abstracted as a linear sequence of bytes that can be addressed at some granularity. For instance, the Linux operating system structures memory this way by providing 4,096 byte page memory units addressable at the byte level. Solaris provides a similar interface by providing 8,192 byte page memory units also addressable at the byte level. There is no structure to these memory units, the provided pages

are collection of bytes to be used by the application process.

Not all distributed shared memory systems abstract memory like the Linux or Solaris operating systems do. Some distributed shared memory systems structure memory in special ways to achieve better performance attributes. The remainder of this section covers some of the methods of structuring data in distributed shared memory systems.

1.3.1 Objects

Distributed shared memory systems like Munin discussed in [16, 17] provide type-specific memory objects. These memory objects exhibit attributes that assist memory coherence protocols in efficiently maintaining a memory address space.

The Munin designers identified several different types of memory objects that could be treated differently by memory coherence protocols. Though the following types aren't specific to any SDSM system in particular, one may expect to see memory objects like write-once objects, write-many objects, read-mostly objects, general objects and synchronization objects. These types do exactly what their names describe.

The usage of memory objects imposes restrictions on application programmers that may not allow those programs to express themselves correctly. That is, to use such a memory interface the application programmer must specifically declare memory types, which may necessitate rewriting the application's source code. Alternatively, as is the case in Munin, runtime systems must accurately identify native data structures and label them as a particular type-specific memory unit [16]. In such cases, the use of a memory interface that provides access

to unstructured memory may be a better memory interface.

1.3.2 Unstructured Memory

Perhaps the most versatile memory interface is to provide access to the underlying bytes and is the approach taken in many popular DSM systems like Treadmarks [15], Shasta [18] and many others. This gives application programmers complete control over how they manage the memory their application uses and doesn't impose any arbitrary limitations (e.g., a key-value memory object may impose read and write operations and would forbid an application from manipulating the raw data directly). This type of memory interface also cannot take advantage of type-specific memory objects. Rather, all memory must be treated as a unit that must be synchronized with the rest of the processors in a SDSM system. This is the biggest drawback of such a memory interface, but thorough research has been done on how one can efficiently keep raw memory synchronized with efficient consistency protocols [1].

Perhaps the most redeeming quality that this type of memory interface exhibits is this: allowing the application programmer to design the memory layout the application uses opens the door to many types of optimizations that can foster speedier runtimes and less memory contention. Exploring the problem of memory contention and various methods of resolving the problem are the focus of §1.4. This type of memory interface also opens up the possibility of allowing compiler-time transformations or binary rewriting tools to generate optimal or SDSM-enabled applications, as was done in work on Shasta in [18].

1.4 Memory Contention & False Sharing

In systems with multiple execution units, the problem of memory contention exists and has the potential to severely degrade the performance of an application. Memory contention is the problem of concurrent access to the same data. Memory contention can further be decomposed into two types: true memory contention and false memory contention. True memory contention refers to a scenario in which n number of processors are concurrently accessing the same memory address. False memory contention refers to a scenario in which n number of processors are concurrently accessing the same memory unit (e.g., a page) but not the same memory address. In both types of memory contention, locks on the memory unit are typically acquired and released in a fashion that serializes the accesses and degrades performance of the application.

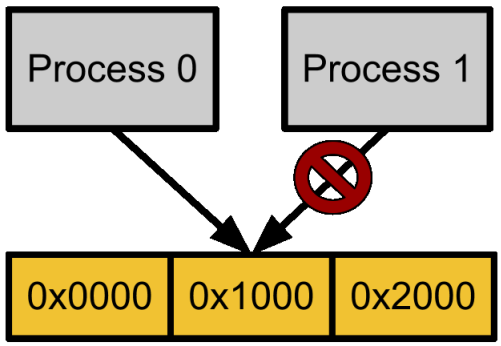

Figure 1.4: Variable contention over page 0x1000. If Process 0 access a page, Process 1 cannot access the same page. Before this can be allowed to happen, Process 0 must relinquish access to page 0x1000.

In addition to memory contention, the problem of false-sharing is a symptom

experienced by systems that do not allow multiple processors to concurrently write to the same memory unit. This can degrade the performance of an application. The problem of false-sharing describes the phenomena observed in systems that duplicate memory regions to each processor n, a cause of excessive communication overhead and an indication that memory isn't shared as much as it is duplicated.

Memory contention and false sharing have an especially harmful effect on SDSM systems in which memory units are located on remote machines and must be faulted in over a network connection. As shown in [22], SDSM systems that suffer from increases in memory latency due to memory contention scale poorly. Furthermore, when n number of remote processors compete for access to a memory unit, a form of thrashing can occur that can render a SDSM system ineffective.

The problem of memory contention and false-sharing oftentimes is addressed in the coherence protocols or structural design of memory units. The remainder of this section examines each of these approaches.

1.4.1 Mitigation Via Coherence Protocol

Multiple-writer protocols seek to alleviate the problem of false-sharing by allowing multiple processors to write to page simultaneously and allow for the system to merge those independent changes at a future synchronization checkpoint [15]. Two major classes of multiple-writer protocols exist: *Tmk* presented in Treadmarks [15] and *HLRC* presented in [23]. Both of these protocols involve twinning memory pages and calculating page diffs. Instead of propagating entire pages at synchronization points, only page diffs are sent.

Multiple-writer protocols were not included in the design of DSM64 because

the experimental data layout used should make such a protocol pointless. The experimental data layout *should* be DSM64's replacement for such a protocol because it will allow multiple processors to access variables that appear to be on the same simutaneously.

1.4.2 Mitigation Via Memory Structure

Several approaches have been taken in related work that structure memory units in a fashion that fortifies the system against the negative effects of memory contention and false sharing. Generally speaking, thorough research has been done on *why* variable contention and false-sharing happen: hot-spot accesses [24, 25], lock contention [26] and data layout [27, 28] are among the major causes. Mitigating these problems typically use approaches involving compile-time transformations, link-time binary rewriting and programming interfaces.

In [27], Jeremiassen et al. investigated the effects of analyzing and transforming application data to lend itself to distributed systems in such a way that false sharing would be minimized. This compile-time approach carefully balanced the benefits of spatial locality with the problem of false-sharing by analyzing what data could be organized onto separate memory units (e.g., pages) to minimize variable contention [27]. Implementing this compile-time approach involved using a modified compiler. However, recent developments in the binary rewriting community are starting to suggest that compile-time solutions can be implemented transparently at link-time. This has several advantages: (i) source code need not be modified and (ii) binary data can be modified to reflect changes at link-time.

The approaches taken in this paper are focused on link-time binary rewriting. Any efforts to manipulate the data layout or binary objects – which traditionally

would be done are done at compile-time – are done at link-time using binary rewriting techniques. Work done in the area of address space randomization in [29, 30, 31, 32] suggests that binary modification and rewriting *is* possible. I used this research as an inspiration to develop a link-time binary rewriter capable of modifing program text instead of just instrumenting it. In doing so, I can implement the compile-time transformations studied in [27] at link-time time with no source code changes.

1.5 Software Distributed Shared Memory

Several of the most influential SDSM systems in literature are Treadmarks [14, 15], Munin [16, 17] and Shasta [18]. Each system is implemented differently, but DSM64 adopts approaches and concepts taken in each of these systems. The remainder of this section discusses other major SDSM systems, and exactly which approaches and concepts are taken from each.

All of these systems participate in some form of virtual memory management (VMM) that either enhances or replaces the VMM facilities provided by the underlying system. Different SDSM systems implement their own mechanisms for modifying the system provided facilities, but there exists a set of fundamental VMM techniques that remain constant between SDSM systems. These techniques are comprehensively studied by Li and Hudak in [1] and provide a theoretically sound bedrock to build many flavors of SDSM systems. The work by Li and Hudak covers various designs for central managers, distributed managers and fault handlers. The work by Li and Hudak inspired design aspects in Treadmarks, Munin and Shasta in these areas.

1.5.1 Treadmarks

Treadmarks is a SDSM system for Unix workstations that runs entirely in user-space. Treadmarks doesn't rely on a particular compiler and depends on user-level memory management techniques to detect accesses and updates to shared data. That is, Treadmarks relies on the target application's use of system provided memory allocators like `malloc`, `mmap`, or `sbrk`. Treadmarks solves excessive network communication with a LRC protocol and mitigates the effect of false-sharing using a diff-based multiple-writer protocol.

The DSM64 system was inspired heavily by the work done by Keleher et al. in Treadmarks [15]. The features most similar between DSM64 and Treadmarks are how the systems handle memory accesses and updates using Unix signal handlers and user-level memory management techniques.

1.5.2 Munin

Munin is a SDSM system for System V workstations that depends on some kernel-specific functionality. Munin relies on a special compiler and language to correctly execute. Munin provides a programming interface that is the same as conventional shared memory parallel programming systems. Munin requires (i) all shared variable declarations to be annotated with their expected access pattern, and (ii) all synchronization must be visible to the runtime system. Item (i) is achieved through memory annotations in source code. Item (ii) is achieved by using a custom threading library that provides routines like `CreateThread` and `DestroyThread`. Later work done on Munin suggested that it used a sophisticated runtime system could guess memory access patterns and automatically annotate memory accesses for the application. Munin employs different memory coherency

models for different types of memories. Different memories can be detected by using source code annotations.

The DSM64 system was not inspired heavily by the work done by Carter et al. in [17] because of the system's reliance on source code annotations and obscure programming interfaces. However, some of the concepts provided in the paper were adopted by DSM64. For example, DSM64 using a popular threading library to make synchronization visible to the runtime system in the same way that Munin uses a custom threading library.

The DSM64 system also adopts Munin's concept of *private* memory. Memory that is *private* is memory that is unique to a node and is not shared or part of the global address space.

1.5.3 Shasta

Shasta is a SDSM system unique among its peers in that it can maintain memory coherence at a fine granularity. Shasta achieves this by transparently rewriting the target application in such a way that all $LOAD$ and $STORE$ instructions pass through a VMM runtime system. This VMM system keeps memory coherent at a high granularity. Shasta also has the ability to dynamically change the size of memory units it keeps coherent. Shasta was capable of running application programs like Oracle 7.3 database system (originally compiled for a single shared-memory multiprocessor) successfully using a RC model without any changes to the original application program [12].

The DSM64 system was inspired heavily by Shasta's strict layout of the target-application's address space. For Shasta, this simplified synchronizing the shared regions of memory between disjoint processors. For DSM64, the same solution is

used.

Though Shasta makes use of a transparent binary rewriter, Shasta only instruments the target-application's executable code. That is, Shasta does not modify the application's executable code. Rather, Shasta instruments it so that all memory access flow though a VMM runtime system. This is a unique strategy for detecting memory accesses and modifications and allows Shasta to provide coherency at a higher granularity than systems that use system-provided signals and memory protections. It is fundamentally different compared to the binary rewriter presented in this paper which seeks to not only instrument but *modify* existing executable code.

1.6 Binary Modification

Binary rewriting is a promising technology that is proving to be useful in various fields – this fact is reinforced by the diversity and quantity of research recently being published on the study and use of binary analysis and modification. Binary rewriting techniques transform compiled executable applications by preserving program semantics, while improving or modifying it in one or more metrics such as runtime performance, energy use, memory use, security and reliability [33, 34]. It is being shown that there are several advantages of binary rewriting over a traditional compiler tool chain. Some of the advantages identified in [33] include whole-program optimizations, transformation coverage, transformation reuse, security enforcement, source code requirements, optimization and platform-aware optimizations.

Modifying binary code is a challenging problem that many researchers have studied in excruciating detail. Binary rewriting can be done statically or dynam-

ically. The work relevant to DSM64 involves statically rewriting binaries (on disk or loaded into memory) and is the topic of further discussion.

The work done in [35] involves rewriting binaries at link-time. This has the advantage of not requiring a special compiler, but is limited in usefulness. Link-time binary rewriting is challenging because binary code must be digested without information that is available to a compiler. Parsing instructions and instruction parameters is a straight-forward task, but deriving information like whether or not an instruction accesses heap allocated memory is more challenging.

Several tools exist that simplify the tasks involved with binary rewriting. The DSM64 system makes use of the DyninstAPI for all of its binary rewriting needs.

1.6.1 DyninstAPI

The DyninstAPI project enables programmers to modify static and dynamic application binaries without requiring the recompilation of the original binary. Based on current research that uses the DyninstAPI, the software has largely been used to instrument applications by adding code to pre-compiled applications [36]. No documented resources suggest the tool is useful for modifying existing program text. The DyninstAPI was inspired by what appeared to be a shortcoming in the traditional software life cycle. The normal life cycle of a developing program is to edit the source code, compile it and execute it [36]. Any modifications to the developing program require re-compilation, re-linking and re-execution. This cycle is considered by some to be too restrictive, as is the case for systems such as performance measurement tools, correctness debuggers, execution drive simulations and computational steering applications [36]. The DyninstAPI seeks to provide a easy-to-use interface to allow application de-

velopers to harness the power of a new software development life cycle through application binary rewriting and instrumentation.

The DyninstAPI itself is an application programming interface that is composed of several other, more specialized, application programming interfaces. The DyninstAPI is largely still being developed. The following is a listing and brief description of each of the component parts of the DyninstAPI that was studied for use in the DSM64 system. Later, I will discuss what parts were relevant to the system presented in this paper, and possible parts that may be relevant to future work on the system.

ParseAPI This component is responsible for parsing binary code (e.g., the machine code representation of a program, library or code snippet) into abstractions such as instructions, basic blocks and functions [37]. The ParseAPI provides the user with a control flow-oriented view of a binary source, where each code object is represented as a collection of functions, basic blocks and edges that represent the control flow graph. The ParseAPI is also extensible to other binary code sources or forms of binary (e.g., memory dumps) [37].

SymtabAPI This component is responsible for providing an abstract view of an application binary by parsing symbol tables, object file headers and debug information [38]. Each application binary is represented with four components: a header block that contains general information about the object (e.g., its name and location), a set of symbols present in the binary, debug information (e.g., type, line number and local variable information) among other additional information that may be provided (e.g., relocation and exception information) [37]. Using the SymtabAPI, developers can add

and remove symbols, fetch pointers to particular regions of the binary (e.g., .data and .text sections), add new types and more.

InstructionAPI This component is responsible for representing raw binary instructions as an abstract form, mainly to describe the semantics of the instruction [39]. The InstructionAPI has three basic capabilities: decoding, abstract representation and disassembly. These capabilities accommodate interpreting sequences of bytes as machine code, representing behavior of instructions and translation of abstract representation of the corresponding assembly language instructions, respectively [39].

StackwalkerAPI This component is responsible for providing a developer a way to walk through the run-time call stack. This gives a developer the ability to read the stack for a given process [40]. The StackwalkerAPI allows for a programmatic means to walk through a stack frame, access process data and look up symbol names.

DynC This component is primarily a convenience for developers whom are injecting code into existing binaries. Because the DyninstAPI allows for developers to inject code, the new code must be formatted as Dyinst-recognizable objects. Using DynC, developers can use C-syntax for injection code [41]. For developers who plan on writing large portions of injectable code, this module makes the process much easier.

ProcControlAPI This component is responsible for providing a developer a way to communicate with processes. The tool is primarily a wrapper over system-provided API that allow the developer to control and modify the attributes specific to a process, like memory address spaces used, when the process may run, among others [42]. A particularly helpful utility this tool

provides controls the memory address space that is used by an application. So, for instance, if one wanted to ensure that several instances of a program on different machines that shared identical address spaces, one could use the ProcControlAPI to do this.

Chapter 2

System Design

The DSM64 system presented in this paper is a software distributed shared memory system. Like Treadmarks in [14, 15], the DSM64 system is implemented entirely in user-space, making complicated kernel-based modifications unnecessary. The DSM64 system features a centralized manager inspired by the designs presented by Li and Hudak in [1]. The DSM64 system maintains an unstructured memory region. In fact, it makes no modifications to the memory structure used by the underlying application unless otherwise noted. This allows the DSM64 system to be used by any application without the need for source code modification, as is the case in some type-specific coherence systems that may require annotations to label memory. The DSM64 system maintains a global address space that follows LRC memory coherency rules. For comparison purposes, the DSM64 system should be able to use other stricter memory coherence models. Which memory coherence model the system uses should be an option chosen at compile-time.

This chapter discusses the assumptions, design considerations, and design decisions of the DSM64 system. The implementation of the design presented in

this chapter is the subject of future chapters.

2.1 Assumptions

Scheduling and deadlines necessitated us to restrict the scope of DSM64 to particular systems running particular libraries. The following is a discussion of each of those assumptions, as well as a brief discussion regarding why the assumption was necessary.

Operating System

DSM64 applications must be compiled, linked and loaded on Linux operating systems (x86 or x86_64). This is due to the system's reliance on the application binary and program loader. Several implications of this assumption include: (i) operators must compile, link and load DSM64 applications with the same compiler tool-chain, (ii) the compiled executable object file format conforms to the ELF specification. These assumptions are necessary because DSM64 makes use of external symbols and relies on characteristics of Linux's loader. Additionally, the binary rewriter presented in this paper only understands the Intel x86 and Intel x86_64 instruction set.

Threading Library Requirements

The DSM64 system is designed to distribute work across multiple physical nodes. The only feasible way to do this is to require that the target application be written using a threading library. The DSM64 system requires that the target application use the `pthread` library, and also assumes that the target application is *properly labeled* using appropriate `pthread` provided

synchronization primitives like `pthread_mutex` lock variables. Currently, the DSM64 system only supports `pthread_mutex`. The DSM64 system does not take advantage of multicore processors and does not run more than one execution unit on a physical node.

Compilation Requirements

The DSM64 system is a shared library that the target-application must be linked with and loaded. This allows DSM64 to replace function definitions for important functions like `pthread_create` and `malloc`.

2.2 Virtual Memory Management Design

Typical virtual memory management (VMM) systems operate in kernel-space, hiding virtually all aspects of memory management to the application programmer. For many types of applications this is permissible, and even preferred. However, for a SDSM system built in user-space, the transparent management of memory is a problem that must be addressed by emulating its behavior in user-space.

The following section is a discussion of the major design concerns and choices made for DSM64. First, I discuss how a VMM must view and organize an application's address space. Second, I discuss various methods for keeping an application's address space synchronized with n many copies. Lastly, I discuss system-provided mechanisms for detecting memory accesses and managing memory permissions.

2.2.1 Address Space Utilization

Virtual memory fools application programs to believe the program has access to a contiguous range of memory. The system-provided VMM does this by mapping process identification and virtual addresses to available physical addresses located in physical memory via lookup tables like the *page table*. This is done transparently. That is, an application never sees or requires the address of a physical page of memory. Rather, these tasks are that of the underlying kernel. For applications that wish to manage memory themselves or for other applications, this is problematic because the underlying placement of memory is invisible to the application. This makes synchronizing memory between multiple processes challenging.

The strategy taken the DSM64 system is to leave the system provided VMM in place and add an additional user-space VMM runtime system to the application program. This has the advantage of allowing one to record the placement of virtual pages, while leaving the task of mapping the virtual page into the underlying system's physical memory. Realistically, a SDSM system one does not care what physical memory is being used.

Using this design, the DSM64 system can share what memory it has available in each of the nodes' virtual memory without the need to know about each nodes' underlying physical memory. It is the task of the user-space VMM to then synchronize available memory and memory permissions to participating nodes.

2.2.2 Address Space Synchronization

I have assumed that the DSM64 system is compiled as a shared library, and that the target application is linked and loaded with this shared library. As a result, the library code exists in the same address space as the target application during the lifetime of the application. This has several non-obvious implications that make building a distributed VMM difficult. This subsection describes the problem of memory allocation in both the target-application and DSM64 library code.

A naive VMM design allows DSM64 to allocate memory from the same memory heap as the target-application. This has the effect of mixing DSM64 heap allocated memory with the target application's heap allocated memory. The problem of deciphering what memory belongs to DSM64 versus what memory belongs to the target-application is difficult. This problem is compounded when one considers that DSM64's and the target-application's heap allocated memory may be mixed on a single memory unit. Additionally, some heap addresses tend to differ between processes which necessitate tedious memory remapping. This was the case with shared libraries loaded by the dynamic linker. This makes synchronization virtually impossible.

Therefore, the first challenge in synchronizing an address space is the challenge of enforcing a clean separation of DSM64 heap allocated memory from the target-application's heap allocated memory. By enforcing a clean separation of DSM64 from the target-application, the problem of synchronization is far less difficult. Using this design, DSM64 may maintain VMM data structures in a separate region of memory that can be synchronized with other DSM64 instances in the cluster. Additionally, this allows us to fix critical DSM64 memory addresses

in memory. Because I have assumed that DSM64 is running the same target application on the same underlying architecture, DSM64 can use fixed addresses and offsets to choose a safe area for its heap.

By maintaining a clean separation of DSM64 memory, DSM64 can instiate VMM data structures in fixed locations of memory. The data structures used to power DSM64's VMM are simple and can be build using *plain old data* (POD) types in C++. These POD objects are self-contained in memory: *all* data is contained in the memory specified by the user. This makes transferring the VMM data structures as simple as type casting a region of memory. Most of the types in the Standard Template Library are POD objects which makes building complex data structures simple, fast and efficient.

2.2.3 Fault Resolution

A distributed node, having been alerted to the shared memory mappings of the network cluster by synchronizing the VMM data structures, needs a mechanism to access the memory that may not be mapped into its own virtual memory. Functionally, this allows a distributed node to read and write memory that another distributed node has mapped into its own virtual memory address space.

Linux provides all of the necessary tools to implement this functionality in two forms: signal handling and access to get/set memory protections. Illegal accesses to memory pages will result in the generation of a `SIGSEGV` signal. By designing a signal handler for `SIGSEGV` one may detect accesses to memory. Additionally, one may even resolve the illegal memory access by remapping memory and allowing the offending instruction to *retry* whatever memory access caused the original fault. Ideally, after resolving the fault and remapping memory, the offending

instruction will execute without illegally accessing memory.

One may manipulate memory protections to override the protections set by the underlying system VMM using the `mprotect(2)` system call. One may combine the use of a signal handler with a system that manages the memory protections to build a VMM in user-space that completely overrides the underlying system VMM. This strategy is very similar to the strategy used by Treadmarks in [15].

2.3 Central Manager Algorithms

2.3.1 Bootstrapping

Bootstrapping a distributed shared memory system is a non-trivial task that requires careful handling of several items closely related to the underlying system. The most important tasks include the loading of the target-application and verification and comparison of address spaces features. Loading an application designed to run on a single machine is a trivial problem where most system-specific intricacies are hidden from the application programmer by their system-provided tool-chain. However, loading an application designed to run on a single machine on a cluster of machines is non-trivial.

Because of the assumptions that the DSM64 system maintains in regards to compilation requirements and operating system similarity, I designed DSM64 in a fashion that allows me to run a *single* application binary on each networked machine. This *single* application binary contains all code to operate as both a master and a worker. This application binary is compiled, linked and loaded with the same compiler tool-chain and and loaded on identical architectures. This

has several beneficial side-effects. Firstly, a single application is distributed between the nodes. Secondly, the object file on disk is identical. Because the same tool-chain and operating system is required, the loaded process image guarantees address space uniformity: loadable segments share the same addresses. The only items that are not guaranteed to be identical is the dynamic linker's placement of shared libraries – this is a security feature that randomizes the places of shared libraries in an attempt to prevent return-to-libc attacks. I use heuristics to identify where process-specific shared libraries exist and take measures to ensure that each process is aware of the placement of other processes' shared libraries. This strategy has the potential to waste memory, but in practice the amount of wasted memory does not exceed 2 MB.

Another non-obvious challenge that the bootstrapping design must address is a facility to halt the target-application until a message or signal is passed to it. Applications designed to run on a single machine need not worry about this design issue because once an application is loaded it is ready to run code – presumably its `main` entry point. This is not the case in a distributed system. The DSM64 bootstrapping system is tasked with determining a master node, waiting for worker nodes to join and initialize themselves, and lastly, signaling a *single* target-application to call its entry point.

2.3.2 Central Manager

There are several networking facilities that the DSM64 system must be able to support. Because of the broadcast nature of many events, the networking facilities use both uni-cast and multi-cast sending machinery. Work shown in [43] suggests that using multi-cast sending performs significantly better than in the past, and

is a natural fit for synchronizing clusters of machines. This problem and the ability to communicate with other nodes is a trivial problem that uses standard socket programming paradigms. Instead, the task of the network managers is to detect *when* certain networking events should happen. For example, *when* should a master node synchronize its data with a new worker or *when* should a master create or join threads on a worker node are events that the cluster coordinator must identify.

The target-application is a `pthread` based program linked with the DSM64 library. Because of this, DSM64 can make use of several system tricks to help identify interesting events. First, the DSM64 library can make use of symbol manipulation to replace or instrument the implementations of functions like `pthread_create` or `pthread_join` with different implementations with stronger symbols. Second, the DSM64 system can make use of allocator hooks to instrument or completely replace the implementation of `malloc` and `free`. Combining these strategies provides a solid foundation to populate an event queue capable of powering a SDSM system.

The DSM64 system requires an event-driven networking infrastructure to communicate the events previously described. I found that using a mixture of multi-cast and uni-cast datagrams made the implementation most straightforward. Furthermore, I found that using a single client-server combination was inappropriate, and opted to maintain multiple servers per-process. Among other reasons, using this design the DSM64 system is capable of performing blocking work without halting the servers responsible for listening and queuing important events in the cluster.

2.3.3 Coherence Semantics

One of the most important features the DSM64 system must support is a weak memory coherency model. Though a DSM system may operate correctly using a stricter coherency model, efficiency and communication overhead become serious concerns. Supporting a weak coherency model maximizes efficiency by minimizing the communication overhead involved in maintaining a global address space.

To implement such a coherency model, the DSM64 system needs to know certain information about the access patterns the target-application exhibits. In systems like Munin [13, 16, 17, 6], this is done through memory annotations. This model was considered too restrictive for DSM64, as it requires source code annotations or an intelligent runtime system to categorize memory accesses. Because of the assumption that the target-application is a `pthread` based program, DSM64 can exploit synchronization primitives like `pthread_mutex`s to enforce a WC model.

A weaker coherency model can be designed in DSM64 by intercepting `pthread_mutex_lock` and `pthread_mutex_unlock`. When a worker node acquires a lock and receives confirmation that the lock was acquired from the central manager, that worker can record what pages are dirtied during its execution. Upon releasing the lock, that worker can transmit what pages it has invalidated to the central manager. The central manager can then lazily propagate these invalidated pages to future acquirers of the same mutex. This approach hangs program correctness on the locking used in the application. That is, program correctness depends on if the progam is *properly labeled* or not.

This approach requires no *additional* source code annotations. I feel that

assuming *properly labeled* applications is a reasonable assumption. There are several interesting caveats that must be taken into consideration when determining what types of applications are *properly labeled*. For example, most `pthread` based applications can make the assumption that the threads are running within the same address space on the same machine. This assumption often leads to a program that allows threads to read or write memory concurrently without the use of locks or barriers, as long as those threads are not reading and writing the *same* memory concurrently. One practical example of this type of application is a `pthread`-enabled version of matrix multiply. Running this type of application on a DSM will not execute correctly because memory modifications will not be propagated to other nodes correctly. These programs are not *properly labeled*. Applications that do not make this assumption and use locks and barriers to synchronize read and write accesses to memory will realize the full power of WC models.

2.4 Binary Mutator

A binary mutator is an advanced tool that makes compilation and execution environments much easier to study and control. Because the DSM64 system runs only on the Linux operating system, many of the features of the underlying ELF object files and program loaders can be studied and exploited. The following is a discussion of the ways a binary mutator may be employed in the DSM64 system.

A binary mutator can be employed to coerce a compiled data layout into the experimental data layout required by the DSM64 system. This can be done by mutating `.data` or `.bss` sections and referring `.text` instructions that referred to the original data offsets. Doing this would allow concurrent and efficient access

to the global and uninitialized data in the compiled application without requiring source code changes and recompilation of the original compiled application.

A binary mutator can also be employed to achieve several other desirable features. One desirable feature is the ability to add code to the compiled application without requiring the original application to be recompiled. This has the advantage of providing a mechanism to add subroutines that aid in the initialization of the network cluster for bootstrapping. This same approach can be taken to instrument and modify the behavior of existing functions. Another desirable feature is the ability to control and verify process address space parameters that may affect the the behavior of the system loader used to run the application.

2.4.1 Experimental Data Layout

The problem of memory contention and false sharing was addressed in §1.4. Memory contention and false sharing is the focal problem that the DSM64 system seeks to mitigate via an experimental data layout. This experimental data layout was designed in such a way to minimize memory contention between the processing cores in the SDSM system.

As a reminder, memory contention is the problem of concurrent access to the same data and is decomposed into two types: *true* memory contention and *false* memory contention. *True* memory contention refers to a scenario in which n number of processors are concurrently accessing the same memory address, whereas *false* memory contention refers to a scenario in which n number of processors are concurrently accessing the same memory unit (e.g., a page) but not the same memory address.

The problem of *true* memory contention is a difficult problem that will degrade

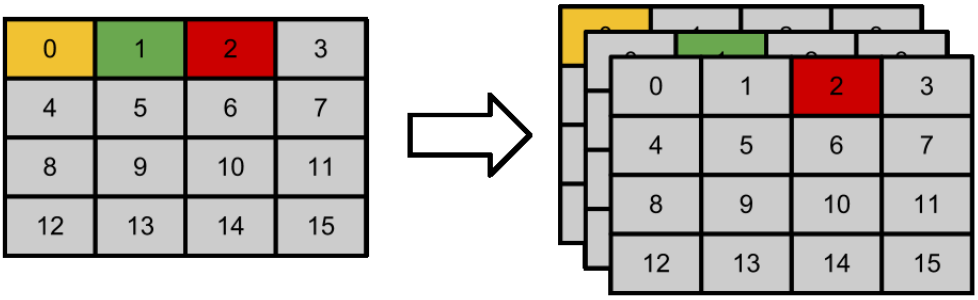

Figure 2.1: An experimental data layout pads variables on a page by a page's worth of bytes. This causes each simple data type to fall on its own page. This layout of memory should minimize variable contention and obviates the need for a multiple-writer protocol.

the performance of a system, as there is no way to allow separate processors with identical access rights to concurrently access the same memory. However, the problem of *false* memory contention is a problem that could be resolved if data can be herded into memory units that processors do not require simultaneously. This strategy also exhibits a beneficial side-effect of decreasing the amount of overhead involved in enforcing protections on memory (usually by a locking mechanism). For example, if processor a wishes to write memory at `0x1004` and processor b wishes to write memory at `0x1008`, both processors will compete for a lock over the memory page which base address `0x1000`. This lock can be avoided entirely if the data exhibits a different layout. Perhaps, the memory processor a wishes to write could be placed at `0x2004` and the memory processor b wishes to write

could be placed at `0x3008`, thereby bypassing any need to compete for the lock on the same memory unit. The layout of data in memory can drastically effect the amount of overhead involved in enforcing protections on memory. This is a highly desirable effect for SDSM systems because of the cost of propagating memory locks or updates over a network connection.

The experimental data layout that DSM64 exhibits organizes data in a way such that every variable falls on its own unique memory unit. This is admittedly a naive decision, but is meant to demonstrate the effectiveness of using the layout of memory to minimize variable contention and nothing more. This decision was made in the face of deadlines.

The task of organizing such a layout in memory is primarily the responsibility of the memory allocator that resides at the heart of the VMM. However, the task also requires the assistance of a binary rewriter to (i) rewrite pre-compiled object files so that variables present in `.data.` and `.bss` sections share the same data layout in memory, and (ii) mutate direct (via absolute offset) and indirect (via pointer or array) references. As I show later, the problem of rewriting indirect references is an extremely challenging problem that ultimately excluded the binary rewriter from final DSM64 deliverable.

Chapter 3

Virtual Memory Management

The virtual memory manager is the soul of the DSM64 system. It understands and coordinates the events that drive the network stacks, and is the primary data structure that coordinates the cluster. At the heart of the VMM resides Hoard [2, 3]: a fast, scalable, and memory-efficient allocator for shared-memory multiprocessors. The VMM is the primary source of events in the system that are communicated to other nodes via the network stacks.

The following subsections discuss Hoard, the VMM used by DSM64, and the process of resolving various types of page faults.

3.1 Overview

In §2.2 I discussed a rudimentary design for a VMM runtime system which addressed problems like address space utilization and synchronization. In §2.3 I discussed the need for several in-process network daemons. One design requirement that is the same from both of these sections is that there must exist a clean

separation of the following: process-specific memory, shared memory and application memory. The following is a list and summarization of the different types of memory that may exist in a target-application linked with the DSM64 library.

Private Memory

Any memory that is unique to a DSM64 node is considered process-specific memory. Memory of this nature is kept separate from other types of memory, and is not subject to synchronization or change from any other node. The memory that the network daemons use is an example of this type of memory. The network daemons' call stacks and heaps must be unique per process because each node is running their own network stack. The best way I found to ensure that other nodes do not interfere with this memory is to maintain a process-specific region of memory.

Shared Memory

Any memory that should be shared between DSM64 nodes is considered shared memory. Memory of this nature should ideally be identical between all nodes in the cluster. An example of this type of memory is the page table and other data structures used to maintain page ownership and location.

Application Memory

Any memory that is allocated or used by the target-application is considered application memory. This is the memory that one uses when using `malloc` within a normal application. This memory is distinct from private memory. Application memory is memory that is *managed* by DSM64 – this memory has metadata that exist in the VMM data structures. Application memory need not exist on all nodes at all times, but it must be accessible to all nodes at all times.

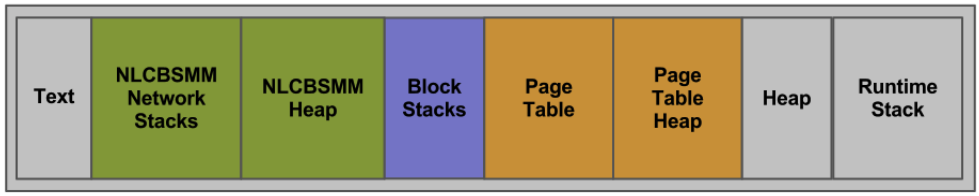

Figure 3.1: The DSM64 process structure segregates memory private memory, shared memory and application memory into regions to make synchronization easier.

Maintaining a clean separation of the different types of memory makes implementing the SDSM system significantly easier for the reasons described in §2.2. I implemented this requirement by allocating and managing fixed regions of memory in the process address space as depicted in figure 3.1. General-purpose use of these fixed regions of memory required custom allocators built to manage memory over these fixed regions. I opted to use *heap layers* to construct these allocators. Hoard and *heap layers* are discussed in greater detail in future sections.

3.2 Managing Virtual Memory

In this section I discuss implementation details of the DSM64 system as it relates to Hoard. I discuss a design paradigm that uses C++ *mixins*, I discuss how Hoard makes use of this design paradigm, and lastly, discuss how Hoard can be minimally modified to build a distributed shared memory system like DSM64.

3.2.1 An Efficient Design Paradigm For Allocators

VanHilst and Notkin introduced an implementation of *collaboration-based* designs that used C++ templatized class inheritance known as *mixins* [44]. Compared to other techniques, their method yields less redundant and reusable com-

ponents. The standard approach to build components that exhibit a collaboration-based design in C++ use virtual method calls at each abstraction boundary [45]. The cost of virtual method dispatching cannot be overlooked in high-performance applications, as is the case for memory allocators in which the cost of dispatching virtual functions is significant compared to the cost of allocating memory. Using a mixin design, one avoids the cost of virtual method dispatch entirely.

The basic structure of a C++ mixin is as follows. This simple example demonstrates how one may use template inheritance to build layers of objects.

```
template <class Super>
class Mixin : public Super {
  ... /* mixin body */
};
```

C++ mixins exhibit the following design pattern. As discussed by VanHilst and Notkin in [44], C++ mixins are the result of the observation that a conceptual unit of functionality is present in neither one object or the other. Rather, the unit of functionality spans multiple layers of objects – each layer providing new refined functionality. Typically, a mixin body calls its superclass's version of the same method, adding a refined layer over the method rather than overriding it entirely.

```
template <class NextLayer>
class ThisLayer : public NextLayer {
  public:
    class Mixin1 : public NextLayer::Mixin1 { ... };
    class Mixin2 : public NextLayer::Mixin2 { ... };
    ...
};
```

Despite the efficient design, C++ mixins are not without some cost. VanHilst and Notkin found it necessary to discuss "pragmatic considerations" in regards

to the use of C++ mixins in [44]. In their analysis, they discuss shortcomings of the C++ compiler regarding templates, type checking and user-error. Interestingly, VanHilst and Notkin also discuss how C++ mixin designs are oftentimes so pragmatic that they may not scale well among large developer communities. The amount of time that went into understanding the complicated design is a testament to VanHilst and Notkin's claim.

3.2.2 Heaplayers

Mixins overcome the restriction of a single class hierarchy, enabling the reuse of the same classes in different class hierarchies. Emery Berger introduced *heap layers* in [45]. Heap layers use an elegant C++ mixin design to compose layers of memory allocators – each layer adding refined functionality to `malloc` or `free`. A heap layer is a mixin that provides a `malloc` and `free` methods and follows memory allocation guidelines. That is, the `malloc` function must return a memory block of a specified size and the `free` function must deallocate the memory block. By design, any mixin class that obeys these guidelines can be used as a heap layer. As discussed in §3.2.1, heap layers can be combined with other heap layers to compose specialized heap layers. A heap layer can communicate with its parent by calling `Super::malloc()` and `Super::free()`.

Hoard is delivered with an existing library of heap layers. The most used layers of this library are listed in Table 3.1. One can observe from the table a taxonomy of heap layers: top heaps, building-block heaps, combining heaps, utility layers, among others. The breakdown of heap layers is by design. For example, all memory allocators must interface with the operating system at some point to obtain more virtual memory. This problem can be solved by implemented a heap

layer responsible for obtaining memory from the operating system. This must be done with `mmap`, `sbrk` or `malloc`. These *top heaps* are the parent heap layers that other refined layers must call via `Super::malloc()` to obtain memory.

Berger provides a short solution for a common way of improving memory performance for applications that allocate many objects from a highly-used class. He did this to demonstrate how specialized layers are composed quickly and with minimal source code changes. His solution uses a per-class pool of memory and uses the `new` and `delete` operators to interface with heap layers.

```
template <class Object, class SuperHeap>
class PerClassHeap : public Object {
  public:
    inline void* operator new (size_t sz) {
      return getHeap().malloc (sz);
    }
    inline void operator delete (void* ptr) {
      getHeap().free (ptr);
    }
  private:
    static SuperHeap& getHeap (void) {
      static SuperHeap theHeap;
      return theHeap;
    }
};
```

Refining a given class, `Foo`, can now be done in as little as two lines of code using the C++ mixin heap layer. Because objects are a fixed size, one can improve memory usage in the allocator by managing freed objects with a `FreelistHeap` – a FIFO linked list that manages objects freed with `free`. This layer can be added by simply adding the heap layer to the class declaration. This example and the composition of more sophisticated layers like the Kingsley and Lea heaps can be studied in Emery Berger's Ph.D. thesis in [45].

```
// Instantiating a refined Foo object in two lines of code
class PooledFoo:
public PerClassHeap<Foo, FreelistHeap<mallocHeap> >{ };
```

One of the key advantages of this C++ mixin approach over other infrastructures is that the compiler can inline method calls. At compile time, this collapses heap layers into a monolithic class with fewer function calls. Additionally, cross-layer optimizations may take place on this new class. These types of optimizations are impossible for compilers to provide for designs that make use of virtual functions at abstraction boundaries.

A Library of Heap Layers	
Top Heaps	
mallocHeap	A thin layer over *malloc*
mmapHeap	A thin layer over the virtual memory manager
sbrkHeap	A thin layer over *sbrk* (contiguous memory)
Building-Block Heaps	
AdaptHeap	Adapts data structures for use as a heap
ChunkHeap	Manages memory in chunks of a given size
CoalesceHeap	Performs coalescing and splitting
FreelistHeap	A freelist (caches freed objects)
Combining Heaps	
HybridHeap	Uses one heap for small objects and another for large objects
SegHeap	A general segregated fits allocator
StrictSegHeap	A strict segregated fits allocator
Utility Layers	
ANSIWrapper	Provides ANSI-malloc compliance
DebugHeap	Checks for a variety of allocation errors
LockedHeap	Code-locks a heap for thread-safety
PerClassHeap	Use a heap as a per-class allocator
General-Purpose Heaps	
KingsleyHeap	Fast but high fragmentation
LeaHeap	Not quite as fast but low fragmentation

Table 3.1: Hoard provided heap layers can be combined to form high-performance custom memory allocators

3.2.3 Hoard

Hoard is memory management system that uses a set of heap layers to provide a fast, efficient and scalable memory allocator. It is a shared library that replaces the system provided allocator (e.g., `malloc` and `free`) with its own implementations that draw memory from a customized composition of heap layers. Hoard was a natural starting point for the implementation of DSM64 because it provided all of the necessary infrastructure for building a SDSM system: system hooks to replace memory allocators and a replaceable allocator composed of heap layers.

The Source Heap

Nested deep in a complicated C++ mixin design is Hoard's source heap – demarcated as `SourceHeap` in the library code. This heap is where Hoard itself gets memory. The object itself is composed of several heap layers, but its type is not important. What is important is the *top heap* that Hoard's `SourceHeap` uses. As discussed in §3.2.2, all of the refinement layers must eventually call `Super::malloc()` to obtain memory from the system. The memory provided by Hoard's *top heap* is then managed by various refinement layers. For Hoard, these layers refine how the *top heap* optimizing memory allocation on a per-thread basis. The `SourceHeap` is Hoard's single replaceable interface to the system.

The DSM64 system implements a special heap layer used for allocating memory. This allocator is responsible for maintaining a global address space. The heap layer I introduced is a *top heap* – it is the single interface between the application program and the operating system. The `SourceHeap` is especially easy to replace because new heap layers can be used to instantiate a completely dif-

ferent heap in as little as two lines of code. Before any node allocates memory, the underlying operating system is made aware of virtual memory mappings on all nodes in the cluster. This way, new memory mappings will never conflict with existing mappings on a different node. Additionally, when new mappings are introduced, internal data structures can be updated and distributed to other nodes in the cluster. The internal data structures used by this heap layer are the subject of the following section. This heap layer is a drop in replacement for Hoard's `SourceHeap`. By adjusting a single declaration in Hoard, I am able to provide an entirely new memory allocator for the system.

In addition to providing new heap layers for the DSM64 system's global address space, I introduced new heap layers for the data structures that maintain the global address space. The C++ Standard Template Library (STL) provides many helpful data structures that can be parameterized with a custom allocator. I added heap layers to act as custom STL allocators. The use of these allocators will become more apparent in the following sections, but are primarily used for enforcing the placement of different types of memory maintained by DSM64's VMM.

System Hooks

To replace the system-provided memory allocator and threading library with one that understands the needs of Hoard and DSM64, several functions or symbols are modified. At a minimum, the Hoard system requires the system-provided allocator to be replaced. The DSM64 system requires the system-provided memory allocator and the system-provided threading library to be replaced.

The system-provider memory allocator was modified by replacing a special

symbol declared in `malloc.h` named `__malloc_initialize_hook`. Using this symbol, Hoard directs the initialization hook to call a custom initialization function – `my_init_hook`. During this custom hook, the Hoard system is able to redirect calls to `malloc` and `free`. I was able to adapt this custom hook to fulfill DSM64's needs by replacing `my_init_hook`.

```
#include <malloc.h>

static void my_init_hook (void);

// New hooks for allocation functions.
static void * my_malloc_hook (size_t, const void *);
static void my_free_hook (void *, const void *);

// Store the old hooks just in case.
static void * (*old_malloc_hook) (size_t, const void *);
static void (*old_free_hook) (void *, const void *);

void (*__malloc_initialize_hook) (void) = my_init_hook;

static void my_init_hook (void) {

  // Register NLCBSMM signal handlers
  nlcbsmm_init();

  // Store the old hooks.
  old_malloc_hook = __malloc_hook;
  old_free_hook = __free_hook;

  // Point the hooks to the replacement functions.
  __malloc_hook = my_malloc_hook;
  __free_hook = my_free_hook;
}
```

Hoard provides implementations for both `my_malloc_hook` and `my_free_hook`. These new implementations use Hoard's `SourceHeap`. In doing so, all memory events flow through Hoard's code. Hoard's `SourceHeap` – having already been replaced by DSM64 – provides an identical memory allocation interface because

it is composed of heap layers. By replacing the system provided allocator with a replacement, DSM64 is able to control all aspects of memory management in the application.

I modified Hoard to intercept addition `pthread` library routines. Doing so enables DSM64 to detect important threading events like creation, destruction and synchronization. These library routines were not designed to be replaceable, thus no initialization hook are provided. Instead, the library routines were replaced by declaring symbols with stronger linkage. This is done by undefining the `__GXX_WEAK__` symbol, which effectively ensures that symbols declared in this object file are stronger than those in following object files – e.g., `libpthread.so`. The following is an example of a replacement routine for `pthread_create`, as well as an example of how a function pointer to the original `pthread_create` may be obtained.

```
#undef __GXX_WEAK__

extern "C" int pthread_create (pthread_t *thread,
      const pthread_attr_t *attr,
      void * (*start_routine) (void *),
      void * arg) {
   char fname[] = "pthread_create";
   // A pointer to the library version of pthread_create (just in case)
   static pthread_create_function real_pthread_create =
      reinterpret_cast<pthread_create_function>
      (reinterpret_cast<intptr_t>(dlsym (RTLD_NEXT, fname)));
   *thread = /* create a network thread */
   return 0;
}
```

Using these two strategies, Hoard and the DSM64 system are able to control required features of the application program. For Hoard, this was to manage the application's memory. For DSM64, this was to not only manage the application's memory, but also the application's use of the `pthread` library.

3.3 Page Table

In traditional systems a VMM system is powered by a data structure called a *page table*. The page table maintains mapping between a process's virtual address space and the underlying physical pages managed by the kernel. The data structures that power the DSM64 system were inspired by this traditional design, but differ slightly only in implementation. The DSM64 system's data structure resembles a traditional page table in that it maintains what virtual memory has been allocated in the system. However, instead of mapping virtual pages to physical pages, the system maps virtual pages to Internet addresses. Each Internet address indicates the current owner of that page.

The primary data structure used in DSM64 has the following `std::map` type. The data structure maintains a virtual address to a tuple containing pointers to `Page` and `Machine` objects, respectively. These objects maintain meta-data about the items they represent. For virtual pages, this meta-data includes the address, page protection bits, a page dirty bit and lastly, a version. For machines, this meta-data includes the Internet address and a status enumeration indicating what the machine is currently doing.

```
typedef
std::pair<Page*,  Machine*>
   PTElement;

typedef
std::map<vmaddr_t,
      PTElementType,
      std::less<vmaddr_t>,
      PTAllocator<std::pair<vmaddr_t, PTElement> > >
          PTable;
```

The DSM64 *page table* is instantiated in a fixed location – see Figure 3.1 –

using placement new. The `PTAllocator` is a special allocator composed of a `FirstFitHeap` wrapped over a `MmapHeap` type. The allocator ensures that all memory allocated by the `std::map` object resides in the fixed region of memory designated for the page table's heap. As a result, the `std::map` object and its associated heap reside in a fixed and known location in memory.

The most convenient feature of this design is the minimal amount of effort required to instantiate the same `std::map` object on n many nodes. For example, after instantiating an object of type `PTable` – presumably on the master node – one must simply copy this memory from one node to another. After memory is copied from the master node to a worker node, the worker node need only cast the new memory region as a `PTable`. At this point, the worker can use the `std::map` object as if it had instantiated the object itself.

Synchronizing the entire page table region is a heavyweight operation that requires locking and significant network communication overhead. At the very least, the operation involves transmitting roughly six pages of data (24576 bytes) of memory. Therefore, synchronizing the entire page table region is typically done very infrequently. Special care is taken when using the page table to ensure that write operations are done by the master.

3.4 Fault Handling

At this point, I have established how DSM64 has instantiated a global address space. Once a node is made aware of memory mappings of other nodes via synchronizing its page table memory region with the master's, the act of looking up who owns a memory mapping is as simple as executing a call to `std::map::find` with the page-aligned address. Using the information stored in the page table, a

node can learn the owner of each page available in the global address space.

DSM64 registers a signal handler for the `SIGSEGV` signal during initialization code that bootstraps the system. This signal is generated upon an illegal memory access either because the offending instruction accesses memory that has not been allocated in the process's address space or because the memory protections on that page forbid access to it. Access, in this case, can refer to *read*, *write*, *execute* or any combination of the above. Upon receiving a `SIGSEGV` signal, one may examine the `siginfo_t` data to discover what address the instruction attempted to access. One may use this address as a key into the page table to fetch the meta-data for the page. If no mapping exists, the signal exits the program – this is a real segmentation violation. If a mapping exists, this means that the address refers to a valid memory page in the cluster.

If a valid memory page exists in the cluster, it can exist on the faulting node or on another node available on the network. In the former case, the node may only have read-only access to the page. In the latter case, the node may be trying to access memory that it did not allocate locally. In either case, the signal handler has all the information necessary to resolve the fault. In the former case, the signal handler need only set memory protections to resolve the fault. This case will set the page's dirty bit, so that changes can be propagated to other nodes at a future synchronization point. In the latter case, the node may initiate a network transaction to download the page from the current owner. This case involves communicating with the central manager to register the ownership change. After the fault is resolved, the offending instruction can be re-executed.

Memory access violations that can be resolved by DSM64 fall into two categories: read faults and write faults. The handling of each fault is different and allows DSM64 to detect page dirtying. The following is a discussion of each type

of fault handler.

3.4.1 Read Fault Handler

A faulting node that requires read access to a page that it does not own is handled by the read fault handler. These types of faults involve contacting the owner of a memory page, moving the memory page to the faulting node and updating the ownership information for the memory page. These faults utilize the DSM64 system's network stacks. The following is the logic used to resolve read faults.

1. Block future `SIGSEGV` signals from being delivered. Signals received during this time will be delivered when the handler unblocks `SIGSEGV`. Because only a single execution thread exists per-node, DSM64 need not handle the case in which another execution thread running on the same node generate a different `SIGSEGV` signal.

2. Use the `siginfo_t.si_addr` field to calculate what page access caused the fault.

3. Use the page aligned address to index into the page table. If a mapping doesn't exist, halt the system.

4. Request ownership from the current owner of the memory page. Block until ownership is acquired.

5. Set the new memory page to have `PROT_READ` permissions using `mprotect(2)`.

6. Return – causing offending instruction to be re-executed.

3.4.2 Write Fault Handler

A faulting node that requires write access to a page that it does not own is handled by the write fault handler. However, if the faulting node requires write access to a page that it already owns, the fault can be handled locally. These types of faults involve locally updating the memory permissions with `mprotect(2)` and setting the memory page's dirty flag to true. In this way, DSM64 can detect when pages are dirtied. This information is important for implementing a LRC model. The following is the logic used to resolve write faults.

1. Block future `SIGSEGV` signals from being delivered. Signals received during this time will be delivered when the handler unblocks `SIGSEGV`. Because only a single execution thread exists per-node, DSM64 need not handle the case in which another execution thread running on the same node generate a different `SIGSEGV` signal.

2. Use the `siginfo_t.si_addr` field to calculate what page access caused the fault.

3. Use the page aligned address to index into the page table. If a mapping doesn't exist, halt the system.

4. Set the page's dirty bit in the page table to true. This indicates the page has been modified.

5. Set the new memory page to have `PROT_WRITE` permissions using `mprotect(2)`.

6. Return – causing offending instruction to be re-executed.

Chapter 4

Central Manager Implementation

In this section, I describe a straight-forward implementation of DSM64 – a distributed shared memory system implemented entirely in user-space. This component was inspired by Li and Hudak's work in [1] which discusses various approaches for implementing shared virtual memory systems. Namely, the concepts and algorithms related to local servers per node and synchronization via a central manager.

4.1 Local Manager

Every DSM64 node, regardless of master or worker status, contains local managers responsible for maintaining memory coherence. Though the managers were implemented as a single object, their functionality can be split into two main functions: a read server and a write server. Using these two servers, a DSM64 node can send or receive memory pages. The servers presented in this section are the married partners of the fault handler servers presented in §3.4.

4.1.1 Read Servers

A read server's primary responsibility is to respond to incoming requests for a page with either the contents of the page or with the current owner of the page. A node can respond to a request without affecting the permissions it itself has on the page. A read server uses the following logic to resolve requests:

1. Lock the `PTable`.
2. If I am the owner of page p, maintain current permissions on p as `PROT_READ` or `PROT_WRITE`.
3. If I am the manager, then ask the owner of page p to send a copy to the requesting node.
4. Unlock the `PTable`.

4.1.2 Write Servers

A write server's primary responsibility is to respond to incoming requests for a page with either the contents of the page or with the current owner of the page. However, in the case of a write, a responding node must relinquish ownership and access rights to the page. A write server uses the following logic to resolve requests:

1. Lock the `PTable`.
2. If I am the owner of page, set the owner of the page in `PTable` to be the receiving node. Change my permissions on p to `PROT_NONE`. Send manager modifications to p.

3. If I am the manager and the current owner modified p, then ask the owner of page p to send a copy to the requesting node.

4. Unlock the `PTable`.

4.2 Central Manager

In addition to performing the tasks of a local manager, discussed in §4.1, the central manager – or master – is responsible for bootstrapping and running the target application on the cluster.

4.2.1 Bootstrapping

When a normal application is loaded, the application's `main` routine is called. In doing so, the application immediately begins to execute code. The DSM64 system requires slightly more control of an application's lifecycle. First, DSM64 requires that an application pause. This provides enough time for the DSM64 system to initialize nodes. Halting all of the target applications also has the convenient side-effect of allowing the DSM64 system to select a central manager, and only start the central manager's target application.

To halt target applications, the target application's source code requires the following code patch. This function creates a named-pipe, and blocks waiting for input. Upon receiving input, the function unblocks, and restores control to the target application's `main` routine.

```
#include <sys/types.h>
#include <sys/stat.h>
void blocking_entry(void) {
   FILE* fp;
```

```
    mkfifo("/tmp/go-pipe", 0755);
    fp = fopen("master-start", "r");
    fprintf(stderr, "> Waiting for unblock signal\n");
    fgetc(fp);
}
```

This solution was chosen in the face of deadlines. Other alternatives exist that would nullify the requirement for patching the target application's code. These alternatives involve rewriting the target application's binary executable or providing a instrumented `main` routine.

Note, this solution blocks the target application's `main` routine, not the DSM64 system's servers. When a target application is blocked and a DSM64 node is un-initialized, the node uses a broadcast address to contact an existing cluster. If no such cluster is found, the un-initialized node can assume a master role. If a cluster is found, the node must synchronize itself with the central manager. This process copies the data structures discussed in §3.3 and verifies that the new node's address space matches the central manager's.

4.2.2 Application Execution

All DSM64 nodes contain the same executable code. After bootstrapping a DSM64 cluster, each node also has identical process images loaded into memory. To execute correctly, one node must act as a central manager. This central manager is responsible for executing a *single* instance of the target application. In practice, this is done by feeding a signal into the the master node's named-pipe. Upon doing so, the target application on the master node begins to execute the `main` routine.

Assuming the target application is written with the `pthread` library, certain

functions can be instrumented or replaced to detect interesting events. The following is a discussion of each of the `pthread` functions DSM64 replaces along with a description of the reason why the event was considered interesting.

pthread_create

By intercepting and replacing `pthread_create` DSM64 is able to prevent new threads from being created locally – presumably on the master node – and instead send a message to a remote node to create the thread there. This was possible because the remote node shared an identical address space. The loadable segment that matters in this case is the `.text` section which holds all of the process's executable code. Because this segment is identical, a function pointer can be passed in the message. The receiving node can use this function pointer to create a local thread using `clone(2)`. A thread identifier is returned to the node who issued the create message. Any memory pages that this thread illegally accesses during executing will be faulted to the local node running the thread.

pthread_join

By intercepting and replacing `pthread_join` DSM64 is able to prevent a parent thread from continuing to execute until all children threads have been waited for. This event causes the master thread to send a message to the node executing this thread. The node can be discovered by querying local data structures that maintain a mapping of thread identifier to Internet address. The child node – the one executing the thread – can then call the real `waitpid(2)` call to wait for the `clone(2)` thread.

pthread_mutex_init

Though not necessary, I found that intercepting and replacing `pthread_mutex_init`

was convenient for building data structures on the central manager. When a mutex is initialized, a message can be sent to the centralized manager notifying it that a new mutex has been created. The address of this mutex can then be stored and used by the central manager – or master – to lookup future requests to that mutex. I assume that *properly labeled* `pthread` applications initialize any mutex variables with `pthread_mutex_init`.

pthread_mutex_lock

By intercepting `pthread_mutex_lock` DSM64 is able to send messages to the centralized manager requesting lock access to a mutex variable. This call blocks until the lock has been granted. This function has special semantics in regard to the coherency protocol. The handling of this function is explained in greater detail in §4.3.

pthread_mutex_unlock

By intercepting `pthread_mutex_unlock` DSM64 is able to send messages to the centralized manager indicating that a thread no longer needs lock access to a mutex variable. This call does not block. This function has special semantics in regard to the coherency protocol. The handling of this function is explained in greater detail in section §4.3.

Harnessing applications written with the `pthread` library allows us to experience several beneficial side-effects. First, the DSM64 system is provided with many source code annotations free of cost. DSM64 assume that a *properly labeled* application with `pthread` routines and synchronization primitives comes at no cost. Second, because the DSM64's replacement routines maintain the same semantics as their original `pthread` based routines, no application modifications are required to correctly execute the application on a DSM64 cluster.

4.3 Memory Coherence

The memory coherence model is one of the most important features of a SDSM system. As discussed in related work and shown in this paper, the memory coherence protocol used directly affects how efficient an SDSM system can execute programs. SDSM systems that make use of a SC or PC memory coherence model *will* work – oftentimes without any annotations or explicit synchronization mechanisms – but will perform poorly because of the communication overhead involved in keeping and address space coherent. This is why SDSM designers sought to weaken coherency requirements by requiring annotations and explicit synchronization mechanisms.

By relaxing memory coherency rules, communication and synchronization can be compiled into synchronization events. Indicating when synchronization events should occur is a task left for the application programmer. The DSM64 system makes use of explicit synchronization mechanisms by replacing existing `pthread` based synchronization primitives. In doing so, DSM64 can exploit a weaker memory coherency protocol known as lazy release consistency (LRC). The remainder of this section is a discussion of the implementation details for DSM64 in regards to the LRC memory coherency protocol.

4.3.1 Lazy Release Consistency

Also covered in Chapter 1, Gharachorloo et al. in [5] discussed the necessary conditions for RC. The formalism is as follows and is included for completeness.

Condition 4.1 (Conditions for Release Consistency)**.**

(A) before an ordinary LOAD or STORE access is allowed to perform with respect to any other processor, all previous acquire *accesses must be performed, and*

(B) before a release *access is allowed to perform with respect to any other processor, all previous ordinary LOAD and STORE accesses must be performed, and*

(C) special accesses *are processor consistent with respect to one another.*

The following is a discussion of the conditions in greater detail using vocabulary relevant to the DSM64 system. Note, the handling of `pthread_mutex_lock` and `pthread_mutex_unlock` have special semantics in regards to implementing a lazy release consistency (LRC) coherency model.

Condition (A)

Condition (A) states that before a processor may perform a *LOAD* or *STORE* access, all other *acquire* access, or `pthread_mutex_lock` events, must be performed. This forces an ordering of events. Viewing this condition from the perspective of a `pthread` application, this states that only one execution thread may enter a critical section at any given time. This rule is enforced by locking a `pthread_mutex` lock variable.

Condition (B)

Condition (B) states that before a processor may perform a *release* access, or `pthread_mutex_unlock` event, all *LOAD* an *STORE* accesses must already be performed. Again, viewing this condition from the perspective of a `pthread` application, this states that before a processor must *LOAD* and *STORE* all memory before exiting a critical section. This rule is enforced by unlocking a `pthread_mutex` lock variable.

Condition (C)

Condition (C) simply states that the ordering of *acquire* and *release* accesses do not matter. This also can be translated into terminology that makes sense for `pthread` applications. This condition simply states that the order of lock and unlock accesses on the same `pthread_mutex` lock variable does not matter – that is, it can be performed in any order.

In this section, I have shown that a *properly labeled* `pthread` application can be cleanly translated into a distributed application which uses a LRC memory coherence model. Additionally, I have seen that in practice most `pthread` applications are written in a way that allow DSM64-enabled applications to correctly perform using LRC while preserving `pthread` semantics.

4.3.2 Implementation

To order events in the distributed system, a central lock manager is used. The central lock manager maintains a data structure that tracks what locks exist in the system, what node currently holds the lock and what nodes are waiting on the lock. This data structure exists only on the central manager to simplify the task of synchronization, as is also the case in systems like Munin and Treadmarks. This data structure uses the following types:

```
typedef
   std::deque<struct sockaddr_in,
   CloneAllocator<struct sockaddr_in> >
      WaitQueue;

typedef
   std::map<vmaddr_t, WaitQueue,
   std::less<vmaddr_t>,
   CloneAllocator<std::pair<vmaddr_t, WaitQueue> > >
```

```
MutexTableType;
```

All synchronization events flow through the central manager. These events include lock initialization, lock acquisition and lock release. The most relevant types of synchronization accesses in a LRC system are acquire and release accesses. The DSM64 treats `pthread_mutex_lock` and `pthread_mutex_unlock` events as acquire and release synchronization accesses, respectively. The following is a brief overview of each type of synchronization access.

Acquire

Locks are awarded in a FIFO nature based on the order `pthread_mutex_lock` messages are received by the central lock manager. After a mutex variable has been initialized, nodes may acquire a lock on the mutex by sending a mutex acquire message to the central manager. This network transaction blocks the sending node. The central manager is then tasked with determining if the requested lock can be granted to the sending node. This is done by checking the `MutexTable` indexed with the address of the mutex variable. One of two events may happen at this point in time. In the case that the central manager discovers that no other node currently holds the lock, the manager may award the lock to the sending node by sending an appropriate message to the node attempting to acquire the lock. In the case that the central manager discovers that another node currently holds the lock, the manager must add the node attempting to acquire the lock to the `WaitQueue` for that particular mutex variable. In both cases, the central manager can resolve the request in a non-blocking manner.

Release

After a node acquires a lock, it must release that lock via a call to `pthread_mutex_unlock`. Upon a `pthread_mutex_unlock` event, the central lock manager is notified of which lock variable has been released. When a manager receives a lock release message, it must take the following actions. In the case that there is no one currently waiting for the lock, the central manager can mark the mutex variable unused and quit. In the case that there is another node currently waiting for the lock, the central manager can send a lock grant to the waiting node. This, like the acquire event, can be handled in a non-blocking manner on the central manager.

Lazy Modification Propagation

One of the most advantageous features of LRC memory coherence models is that changes to memory can be lazily propagated between nodes. This feature comes with two challenges. First, the system must be able to detect what memory page have been modified during a synchronization access – demarcated by calls to `pthread_mutex_lock` and `pthread_mutex_unlock`. Second, the system must have a mechanism to propagate the changes to the correct node.

The DSM64 system solves both of these problems. Detecting what memory pages have been modified during a synchronization interval is done with a fault handler capable of distinguishing between read and write faults. When a page is modified by node p, the system must guarantee that another node q sees the new version of the page upon the next access following a synchronization event. Assuming a program is *properly labeled* with lock variables, this can be done by propagating what pages were modified in the message for `pthread_mutex_unlock`.

Using this strategy, the central lock manager is informed of what pages are currently in a modified state. This implies that the node that issued the lock release message is the current owner, as that node is the node that last wrote to the page.

When a new node performs a `pthread_mutex_lock`, the central lock manager can inform the node of what pages the old owner modified. This must happen before the new node accesses the page. DSM64 can guarantee this by setting dirtied pages' memory protections to `PROT_NONE` on the new node. The new node can then fault in only the pages that the previous owner dirtied. Additionally, this happens only when the new owner accesses that page – lazily propagating the new page to the new node at the time of access. This approach uses significantly less messages, performs correctly and performs significantly quicker than the system that exhibits *strict consistency*.

Chapter 5

Binary Modification

The DSM64 system requires a binary mutator to support its experimental data layout in memory discussed in §2.4.1. In conjunction with the VMM runtime system, a binary mutator can rewrite indirect data references in a way that can address the format used by the VMM runtime system – a format which minimizes variable contention. Though this feature never fully came to fruition, this chapter provides some of the insights, solutions and challenges of binary rewriting. Without the time and energy spent on studying the ELF specification, the process of linking and loading, the structure of a process image, among many other details, the efforts taken in this paper to implement a SDSM system would have been futile. The knowledge from the study of binaries and binary rewriting heavily influenced how DSM64 verified address spaces, organized address spaces and aided in the methods involved in symbol maniuplation and binary modification.

In this chapter, I discuss standards and concepts of binary rewriting. I provide an in-depth discussion of object file formats and the processes of linking and loading applications. I report on the effectiveness of the DyninstAPI, a tool for

binary analysis and instrumentation. Lastly, I discuss what shortcomings and limitations prevent this component's inclusion in DSM64.

5.1 Design & Implementation

With an understanding of Executable and Linkable Format (ELF) and the tools used to analyze and rewrite pre-compiled ELF object files, one can move onto investigating the process of rewriting ELF object files. Using the DyninstAPI, one may rewrite object files statically (modify the contents of the object file on disk) or dynamically (modify the contents of the process loaded into memory). The remainder of the work done in this paper statically modified ELF object files.

5.1.1 Binary Analysis

This section seeks to examine the internals of a compiled ELF executable object file to exercise the knowledge discussed in Appendix A. Additionally, several tools available on Linux platforms are discussed to analyze the internals of ELF executable object files – namely, *readelf*, *objdump* and *hteditor*.

For the remainder of this subsection, let us consider the following simple C application compiled on a 32-bit Debian workstation running a GNU/Linux 3.0 kernel. Later, more advanced examples will be used to illustrate the functionality of the binary rewriting tool introduced in this paper.

```
#include <stdio.h>
int global_ab = 0xAAAABBBB;
int global_cd = 0xCCCCDDDD;
int main(int argc, char* argv[]) {
```

```
  int temp = global_ab + global_cd;
  fprintf(stdout, "> %d <\n", temp);
  return 0;
}
```

Note I compile the simple C application with debugging flags enabled with no optimizations. This is to force the compiler to annotate symbols with debugging information and to disable the compiler from performing any transformations to the code that the application doesn't explicitly specify. To compile this C application, I use the GNU C 4.2.1 compiler and the following command:

```
[sys]$ gcc -O0 -g simple-c-application.c -o simple-c-application
```

After compiling, one is left with an ELF executable object file named `simple-c-application`. To examine the innards of this ELF object file one can use tools like *readelf* – a tool provided by the *binutils* suite.

```
[sys]$ readelf -S simple-c-application
There are 36 section headers, starting at offset 0x13b8:

Section Headers:
[Nr] Name              Type            Addr     Off    Size
[ 1] .interp           PROGBITS        08048154 000154 000013
[11] .init             PROGBITS        080482d8 0002d8 00002e
[12] .plt              PROGBITS        08048310 000310 000040
[13] .text             PROGBITS        08048350 000350 0001ac
[14] .fini             PROGBITS        080484fc 0004fc 00001a
[21] .dynamic          DYNAMIC         08049f28 000f28 0000c8
[22] .got              PROGBITS        08049ff0 000ff0 000004
[23] .got.plt          PROGBITS        08049ff4 000ff4 000018
[24] .data             PROGBITS        0804a00c 00100c 000010
[25] .bss              NOBITS          0804a020 00101c 00000c
[34] .symtab           SYMTAB          00000000 001ef8 0004a0
```

This command dumps the section headers for this ELF object file. For sake of brevity, many of the thirty six sections were removed. From the little information provided, one can learn many things about this application binary. For

instance, one can learn the size and offsets of each section. One can learn that this application has initialized global data (located in the `.data` section), as well as un-initialized global data (located in the `.bss` section).

```
[sys]:~$ readelf -l simple-c-application

Elf file type is EXEC (Executable file)
Entry point 0x8048350
There are 9 program headers, starting at offset 52

Program Headers:
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg
PHDR           0x000034 0x08048034 0x08048034 0x00120 0x00120 R-E
INTERP         0x000154 0x08048154 0x08048154 0x00013 0x00013 R--
[Requesting program interpreter: /lib/ld-linux.so.2]
LOAD           0x000000 0x08048000 0x08048000 0x00620 0x00620 R-E
LOAD           0x000f14 0x08049f14 0x08049f14 0x00108 0x00118 RW-
DYNAMIC        0x000f28 0x08049f28 0x08049f28 0x000c8 0x000c8 RW-
NOTE           0x000168 0x08048168 0x08048168 0x00044 0x00044 R--
GNU_EH_FRAME   0x000528 0x08048528 0x08048528 0x00034 0x00034 R--
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW-
GNU_RELRO      0x000f14 0x08049f14 0x08049f14 0x000ec 0x000ec R--

Section to Segment mapping:
00  ;; the program header (displayed above)
01  .interp
02  .interp .note.ABI-tag .note.gnu.build-id  .gnu.hash .dynsym
    .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init
    .plt .text .fini .rodata .eh_frame_hdr .eh_frame
03  .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
04  .dynamic
05  .note.ABI-tag .note.gnu.build-id
06  .eh_frame_hdr
07
08  .ctors .dtors .jcr .dynamic .got
```

This command dumps the program header table for this ELF object file. As covered in §A.1, one can learn many details about the application binary from this output: section-segment mappings, segment types, segment attributes, among other details. For example, one can learn the section-segment mappings by

iterating through the second displayed table. This table shows us that segment 3 contains both `.data` and `.bss` sections, and that segment 2 contains most of the executable code in the object file (e.g., the `.text`, `.fini`, `.plt` sections, among others). One can also learn the target virtual memory addresses each segment is destined for, as well as each segment's size on disk and size in memory. Lastly, one can learn which segments are actually loaded by the system interpreter at load time by referring to each segment's type.

As much as one can learn from the output of *readelf*, the tool does not disassemble the ELF object file. To see more into the innards of this application binary, tools like *objdump* can be used to disassemble sections and display their contents.

```
[sys]$ objdump -d -j .text simple-c-application

simple-c-application:     file format elf32-i386

08048404 <main>:
 8048404: 55                    push   %ebp
 8048405: 89 e5                 mov    %esp,%ebp
 8048407: 83 e4 f0              and    $0xfffffff0,%esp
 804840a: 83 ec 20              sub    $0x20,%esp
 804840d: 8b 15 14 a0 04 08     mov    0x804a014,%edx   ;; attn
 8048413: a1 18 a0 04 08        mov    0x804a018,%eax   ;; attn
 8048418: 01 d0                 add    %edx,%eax
 804841a: 89 44 24 1c           mov    %eax,0x1c(%esp)
 804841e: ba 20 85 04 08        mov    $0x8048520,%edx
 8048423: a1 20 a0 04 08        mov    0x804a020,%eax
 8048428: 8b 4c 24 1c           mov    0x1c(%esp),%ecx
 804842c: 89 4c 24 08           mov    %ecx,0x8(%esp)
 8048430: 89 54 24 04           mov    %edx,0x4(%esp)
 8048434: 89 04 24              mov    %eax,(%esp)
 8048437: e8 04 ff ff ff        call   8048340 <fprintf@plt>
 804843c: b8 00 00 00 00        mov    $0x0,%eax
 8048441: c9                    leave
 8048442: c3                    ret
```

This snippet of disassembled `.text` section provided by *objdump* can be used to examine the raw instructions that compose the main procedure meant to run on the system's processor. It can be studied to learn the behavior of the application binary, as well as some the the behavior dictated by the ELF format. In regards to `simple-c-application`, one can see at offset `0x804840d` and offset `0x8048413` that two absolute addresses are being loaded into registers and subsequently added (these `.text` offsets were annotated in the above example with `attn` comments for readability).

```
[sys]$ objdump -d -j .data simple-c-application
simple-c-application:     file format elf32-i386

Disassembly of section .data:

0804a00c <__data_start>:
 804a00c: 00 00
0804a010 <__dso_handle>:
 804a010: 00 00 00 00
0804a014 <global_ab>:
 804a014: bb bb aa aa
0804a018 <global_cd>:
 804a018: dd dd cc cc
```

Using *objdump* to disassemble the `.data` section, one can clearly see the application binary is adding `global_ab` and `global_cd`. Using *objdump* one can completely disassemble and study the contents of any section in the ELF object file to learn what data it is using, what instructions it possesses, what functions it uses, among many other things. In conjunction with $readelf$, one can learn any details the operating system may require to load and run the application. Referring back to Figure A.4, one can see that the data displayed in the two tools used in this subsection operate over the ELF data specified in [46].

5.1.2 Binary Modification

Up until this point, I have merely analyzed the contents of a simple C application. This subsection delves into the process and implementation taken in this paper to rewrite the static data in the ELF executable object files used in this experiment. Currently, no support for rewriting indirect data references exists. The implementation of the binary rewriting tool involved extensive uses of the suite of tools used to build the `DyninstAPI` (this suite of tools is discussed in §1.6.1).

Goal & Overview

It is important to remember the goal of the binary rewriting task: to arrange the `.data` (and perhaps the `.bss` section) in a fashion that preserves program semantics (e.g., the mutated program must execute the same as the original program) and permits its execution on a platform that requires a specific `.data` and `.bss` format.

With this goal in mind, the process taken in this mutator program is as follows. Each of the items in the following list is the focus of a separate subsection, where implementation details are discussed.

1. Rewrite the `.data` section.
2. Rewrite the `.text` section to reflect changes to the `.data` section.
3. Rewrite the `.symtab` section (e.g., the symbol table) to reflect changes to both the `.text` and `.data` sections.
4. Emit a new binary program with a mutated `.data` section that exhibits the same behavior as the original binary program.

The following C code was used to compile an ELF object file used for testing the base functionality of the binary rewriter. Base functionality was defined as the emission of a mutated binary that differed in its `.data` section while maintaining the original program's semantics.

```
#include <stdio.h>
int var_1 = 100;
int var_2 = 200;
int var_3 = 300;
int var_4 = 400;
int var_5 = 500;
int var_6 = 600;
int main() {
  fprintf(stdout, "%p: %d\n", &var_1, var_1);
  fprintf(stdout, "%p: %d\n", &var_2, var_2);
  fprintf(stdout, "%p: %d\n", &var_3, var_3);
  fprintf(stdout, "%p: %d\n", &var_4, var_4);
  fprintf(stdout, "%p: %d\n", &var_5, var_5);
  fprintf(stdout, "%p: %d\n", &var_6, var_6);
  return 0;
}
```

After compiling this C application, the `.data` section appears as follows. The most pertinent information to take from this data is the offsets of each of the global variables.

```
08049754 <__data_start>:
 8049754: 00 00 00 00
08049758 <var_1>:
 8049758: 64 00 00 00
0804975c <var_2>:
 804975c: c8 00 00 00
08049760 <var_3>:
 8049760: 2c 01 00 00
08049764 <var_4>:
 8049764: 90 01 00 00
08049768 <var_5>:
 8049768: f4 01 00 00
0804976c <var_6>:
 804976c: 58 02 00 00
```

Data Modification

The first task of the mutator program is to modify the `.data` section of the original program. This is a relatively simple task for the provided example with help from the `SymtabAPI`.

```
#include <Module.h>
#include <Symbol.h>
#include <Variable.h>

using namespace Dyninst;

typedef std::map<SymtabAPI::Offset, SymtabAPI::Offset> OffsetMap;
typedef std::vector<SymtabAPI::Variable*> VarList;

class SimpleDataMutator : public DataMutator {
   public:
      // The main symtab object
      SymtabAPI::Symtab* symTab;
      // A region object (.data)
      SymtabAPI::Region* dataRegion;
      // A region object (.bss)
      SymtabAPI::Region* bssRegion;

      // A mapping of old offset to new offset
      OffsetMap newOffsets;

      // Pointers to raw data
      unsigned char* oldRaw;
      unsigned char* newRaw;
      unsigned char* oldRawBss;
      unsigned char* newRawBss;

      // Raw data sizes
      unsigned int newDataBase;
      unsigned int newDataSize;

   public:
      DataRewriter(Symtab*);
      VarList filterVariables(Region*);
      void organizeNewData(Region*, unsigned char);
      void organizeNewBss(Region*, unsigned char);
```

```
};
```

Without going into excessive code implementation details, the `SymtabAPI` allows for the population of the member data of a `SimpleDataMutator` object through a public interface that provides handles to region and offset abstractions. These abstractions provide a friendly interface for getting and setting the data that composes a ELF section. Among these public interfaces is the ability to attain a pointer handle to raw data regions. Using this interface raw data can be identified, read and placed into a new memory allocated buffer in any arbitrary structure or order. This is precisely the function of `filterVariables()` and `organizeNewData()`. Post refactoring, a pointer handle to the new raw data buffer can be provided to `SymtabAPI` to update the ELF object file. The result of this process results in the following data section:

```
08049754 <__data_start>:
 8049754: 00 00 00 00 aa aa aa aa

0804975c <var_1>:
 804975c: 64 00 00 00 aa aa aa aa

08049764 <var_2>:
 8049764: c8 00 00 00 aa aa aa aa

0804976c <var_3>:
 804976c: 2c 01 00 00 aa aa aa aa

08049774 <var_4>:
 8049774: 90 01 00 00 aa aa aa aa

0804977c <var_5>:
 804977c: f4 01 00 00 aa aa aa aa

08049784 <var_6>:
 8049784: 58 02 00 00 aa aa aa aa
```

The implementation of `organizeNewData()` pads integer values with an additional `0x0004` units of memory (set to the value `0xaa` for sanity checking purposes). This small padding minimally satisfies the requirements of mutating the `.data` section enough so that the variables can be considered relocated. The amount of data padding can be arbitrarily large. This allows one to pad variables with enough extra data so that each variable falls on its own memory unit.

Text Modification

By mutating the `.data` section of the ELF object file, the binary mutator is obligated to shift focus to the `.text` section. This is necessary to preserve the semantics of the original program. After all, relocating (e.g., changing the offset) of global variables will drastically change the performance of an application if the program text that references that data uses the old offsets. Rewriting a program's `.text` section is a fascinating but very challenging problem. The binary mutator presented in this paper provides the following public interface for rewriting `.text` sections:

```
#include <Immediate.h>
#include <InstructionDecoder.h>
#include <Module.h>
#include <Symbol.h>
#include <Variable.h>

using namespace Dyninst;
using namespace InstructionAPI;

typedef std::map<SymtabAPI::Offset, SymtabAPI::Offset> OffsetMap;
typedef std::vector<SymtabAPI::Variable*> VarList;

class SimpleTextRewriter : TextMutator {
  public:
      // The main symtab object
      SymtabAPI::Symtab* symTab;
```

```
      // A region object (.data)
      SymtabAPI::Region* dataRegion;
      // A region object (.bss)
      SymtabAPI::Region* bssRegion;
      // A region object (.text)
      SymtabAPI::Region* textRegion;

      // A mapping of old offset to new offset
      //  (Generated by a DataMutator)
      OffsetMap* relocs;

      // Pointers to raw text
      unsigned char* oldText;
      unsigned char* newText;

    public:
      TextRewriter(Symtab*, AddrMapping*);
      void organizeNewText();
};
```

The `DyninstAPI` suite of tools supports many features but is lacking in those that permit modification of program text. To achieve the goals of this module, the the `ParseAPI` and `InstructionAPI` proved to be indispensable. After obtaining a pointer to the instructions in the `.text` section (attained in the same fashion as the `.data` section) the following code snippet allowed the parsing and disassembly of each instruction:

```
InstructionDecoder decoder(oldText, textSize, Arch_x86);
Instruction::Ptr i = decoder.decode();
while (i != NULL) {
  // Replace absolute offsets using relocs
  // Decode next instruction
  i = decoder.decode();
}
```

For readers familiar with the Intel x86 instruction set, this interface is a blessing: instruction set peculiarities like variable length instructions are handled

transparently. However, the library is still rather limited for the task of statically rewriting the object file's text. Based on the `DyninstAPI` documentation and function signatures it is clear that the library is intended to be used at runtime in order to profile the running application (e.g., analyze control flow, register contents, etc.) instead of mutating the instructions themselves.

Because the binary mutator presented in this paper uses the `InstructionAPI` in a way that it wasn't originally intended for (the tool was used to get and set the raw bytes of each instruction) a new problem was introduced: comprehensive knowledge of the Intel x86 assembly language was necessary to correctly recognize the different mechanisms used to access, refer to and write data. Building a Intel x86 disassembler was outside the scope of this project. Therefore, to allow progress on the construction of the binary mutator heuristics were used to identify and replace absolute addresses in `.text` sections. Among these heuristics are hints as to which types of instructions usually access memory via absolute offset and which instruction parameters typically contain absolute addresses.

Using heuristics, instructions like `0x8b 0xa1 0x58 0x97 0x04 0x08` (the raw representation of `mov EDX, 0x8049758`) could be identified as access to global memory in the `.data` section. It would subsequently be written back to the object file as `0x8b 0x15 0x5c 0x97 0x04 0x08` (the raw representation of `mov EDX, 0x804975c`). This transformation rewrites an instruction that refers to `var_1` in the example application. Curious readers may also choose to cross references these offsets with the updated `.data` sections.

From testing and observation, if an offset was replaced, it was always correct because the instruction could be verified access the `.data` section. However, guaranteeing that all absolute references were replaced was more difficult to guarantee and often was verified manually.

Symbol Table Modification

The implementation of the symbol table rewriter was trivial. The `SymtabAPI` is a very complete and developed API that made updating the ELF object file's symbol table a simple task. Functionally, updating the symbol table allowed the mutated binary to behave correctly cosmetically – that is, the mutated binary would display the correct data types, sizes, names, offsets and more when using tools like *readelf* and *objdump*. Updating the symbol table would be more important if one was rewriting a static or shared library which depended on the symbol table for linking.

Program Header Modification

No public interface exists in the `DyninstAPI` for rewriting the program header table. Therefore, additional methods were employed to achieve this goal. Rewriting the program header table was necessary for one primary reason: when one increases the size of a section there exists the risk of colliding with other sections. The approaches taken by the binary mutator presented in this paper grow the `.data` section to pad each of the variables present. By not updating the program header, this enlarged `.data` section and respective program text references collide with the `.bss` section. A mutated program that was rewritten, but not had its program header updated, would have any variables that mistakenly cascade with the `.bss` section zeroed out by the program interpreter at load time. This process was done manually using *hteditor* – a hex editing tool that understands ELF and other binary formats. Using *hteditor* I was able to increase the sizes of enlarged sections and write the modifications back to the original ELF object file.

5.2 Shortcomings & Limitations

The work presented in this chapter *does* work: using the binary rewriter I was able to enlarge the size of application's `.data` and `.bss` sections, move the addresses of variables and rewrite `.text` to reference the new variable offsets. In doing all of this, I was able to maintain program correctness. Admittedly, the binary rewriter was not packaged with the final DSM64 deliverable because of several hard problems that remained unsolved. These hard problems still exist in even the most advanced binary mutators. This section discusses some of the more problematic issues that I encountered while attempting to support the experimental data layout discussed in §2.4.1.

5.2.1 Indirect Data References

The first and most limiting shortcoming of the binary mutator presented in this paper is the requirement that data must be referenced with an absolute address. It is very unlikely that anything more complex than the simple examples presented in this paper will make all data references via absolute offset. Rather, some references to data will be made indirectly (for C programmers, you can think of this as data being references by a pointer or array). Smithson in [33] experienced the same problem when attempting to relocate data. This was especially troublesome [33] because program data was often mixed into the `.text` section which, when relocated, was very difficult to resolve due to indirect references to the embedded data. Smithson avoided this problem by not relocating any program data. Unfortunately, this is not an acceptable solution for the binary mutator presented in this paper because of the fundamental requirement to relocate data in the `.data` section. Additionally, *all* indirect references may have

to be rewritten to address memory using the experimental data layout.

5.2.2 Composite Types

Programs that use composite types (e.g., structures or classes) are significantly more difficult to rewrite correctly. Beyond discovering the type and size of the composite type (assuming debugging information was provided), the main difficulty in rewriting composite types is dynamically padding the composite type itself. For instance, if a class provides a sequence of lock variables as private member data and the goal is to place each of these locks on its own page of virtual memory (e.g., padding each lock variable with `0x1000` bytes of memory) then it makes no sense to rewrite the original binary in such a way that each instance of the class be padded next to other global data. Rather, it makes sense to pad the member data itself. Modifying the memory layout of composite types introduces new uncertainties that cloud the correctness of the binary rewriter introduced in this paper. This problem is compounded when one considers how the binary mutator should operate with composite types that are composed of other composite types.

5.2.3 Heuristics & Mutation

The use of heuristics in mutating a program's text is a brittle method to rewrite ELF object files. This shortcoming could be resolved by developing a Intel x86 interpreter that understands the different methods and mechanisms of accessing data in a program. By not having to depend on heuristics to identify absolute address, and instead rely on the standards the Intel x86 instruction uses a binary mutator would be able to reliably rewrite *all* references to absolute

offsets in *all* types of references-capable instructions.

Chapter 6

Results

Even though the binary mutator never fully came to fruition in this paper, the system still *works*. The DSM64 system runs correctly using various memory coherence models and yields similar results to the SDSM systems studied in previous chapters. Because the binary rewriter never fully came to fruition, the DSM64 system suffers from false-sharing. Even so, the DSM64 system yields impressive results.

6.1 Basic Operation Costs

The DSM64 system shares many of its basic operation costs with Treadmarks. The minimum round trip time using UDP sending and receiving primitives is 500 microseconds. Treadmarks reported that this figure is composed of sending and receiving via a Unix socket for 80 microseconds per operation (a total of 320 microseconds) and an additional 180 microseconds of wire-time, interrupt processing time and resuming the blocked processor in receive. These costs are incurred for each message sent by DSM64 and are factored into the results presented in this

section.

The minimum time for DSM64 to resolve a read fault - an operation that involves transferring a 4096 memory page over a Unix socket - is 1909 microseconds. This outperforms Treadmarks minimum time to resolve a read fault (referred to as a remote fault in [15]) of 2792 microseconds. This difference is likely due to hardware differences between the two systems. The minimum time for DSM64 to resolve a write fault – an operation that involves adjusting the local memory permissions on a node and lazily invalidating the page – is 45 microseconds.

6.2 Matrix Multiplication

I collected runtime characteristics for DSM64 using an unoptimized `pthread`-based matrix multiplication benchmark that sought to replicate the matrix multiplication benchmark used for Munin in [47]. However, because of implementation differences two changes were made: (i) the algorithm does *not* use barriers and (ii) the algorithm uses threads to calculate rows of the result matrix instead of chunks of the matrix. These differences do change access pattern characeristics in such a way that require DSM64 to communicate more. I opted to use an unoptimized `pthread` application to demonstrate that the target-application need not be designed to run on the SDSM system, as would be the case for systems that use a custom threading interface. The application I used is *properly labeled* using mutex lock variables.

The runtime characteristics presented in this section use the following algorithms for multiplying matrices, each variation exhibiting slightly different locking patterns. The application executes with the following logic.

1. The main thread allocates the input matrices into the global address space using `malloc`. This places input and output matrices on the heap.

2. The main thread initializes the input matrices with pseudo-random data.

3. The main thread spawns n many worker threads, starts each worker thread and waits for the threads.

4. Each worker thread performs work and joins the master thread.

5. The main thread serially verifies the correctness of the result matrix.

Firstly, I present `mmult-lock-row`, a matrix multiplication algorithm which requests a write-lock on the output matrix for each row of output. This version of the algorithm access memory in row-major order, which results in successive elements in the matrix being physically located on the same page. As a result, I can expect processors to contend for pages that contain the result matrix. The following is a slightly simplified version of the algorithm:

```
while (i < MATRIX_DIM) {
   pthread_mutex_lock(&lock);
   for (j = 0; j < n; j++) {
      sum = 0.0;
      for (k = 0; k < n; k++) {
         sum = sum + a[i][k] * b[k][j];
      }
      c[i][j] = sum;
   }
   pthread_mutex_unlock(&lock);
   i += NUM_THR;
}
```

As an experiment, I decided to implement a version of the matrix multiplication algorithm that accessed memory in a column-major order. Work done in

[48] suggested that a column-major allocation strategy may relieve memory contention over output matrices. In the case of the matrix multiplication algorithm, this would decrease the number of messages and faults associated with contention over the result matrix. However, as also suggested in the paper, the advantages from just the allocation strategy are not as clear as one would like: relieving memory contention depends on a mixture of synchronization, allocation strategy and the number of processors. Even so, I decided to run `mmult-lock-col` on DSM64 to understand how this allocation strategy will effect the system. As suggested in [22], custom programmed memory allocation and access algorithms are oftentimes one of the most effective ways to relieve memory contention. The following is a slightly simplified version of the algorithm:

```
while (i < MATRIX_DIM) {
   pthread_mutex_lock(&lock);
   for (j = 0; j < n; j++) {
      sum = 0.0;
      for (k = 0; k < n; k++) {
         sum = sum + a[k][i] * b[j][k];
      }
      c[j][i] = sum;
   }
   pthread_mutex_unlock(&lock);
   i += NUM_THR;
}
```

To correctly execute this distributed application, the `pthread`-based target application must be *properly labeled* using `pthread` synchronization primitives. I assume *properly labeled* applications. However, it is interesting to note that target-applications that do not make use of synchronization primitives can still correctly execute with a stricter coherency protocol – namely, *processor consistency*. Any `pthread`-based application that depends on thread identifiers and write-offsets instead of synchronization primitives will correctly execute only on

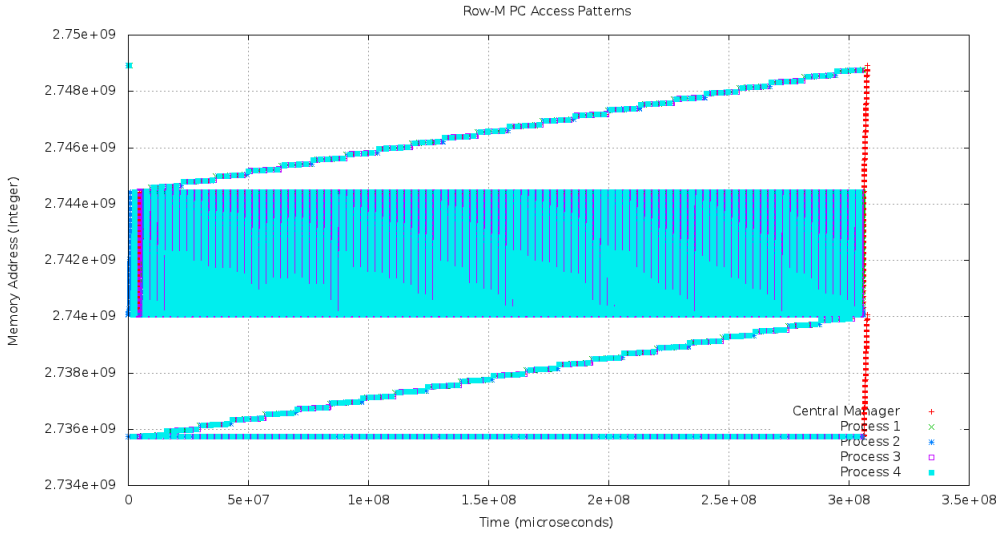

Figure 6.1: Access patterns exhibited by a *PC* coherency model during the running of a matrix multiplication benchmark application accessing memory in row-major order.

DSM64 versions that use PC.

6.2.1 Access Patterns

To demonstrate the different natures and effects of coherency models, I included Figure 6.1 and Figure 6.2. Figure 6.1 depicts the access patterns of a matrix multiply benchmark application using a PC model. This coherency model necessitates that each read or write to memory happen on the latest version of a page. The only feasible way to enforce this is to acquire a lock on each page regardless of access type.

Figure 6.2 depicts the access patterns of the same matrix multiply benchmark application using a *LRC* coherency model. This coherency mode provides a mechanism for each node to maintain read-only pages, and lazily propagates modified pages only to processors that actually require the updated page.

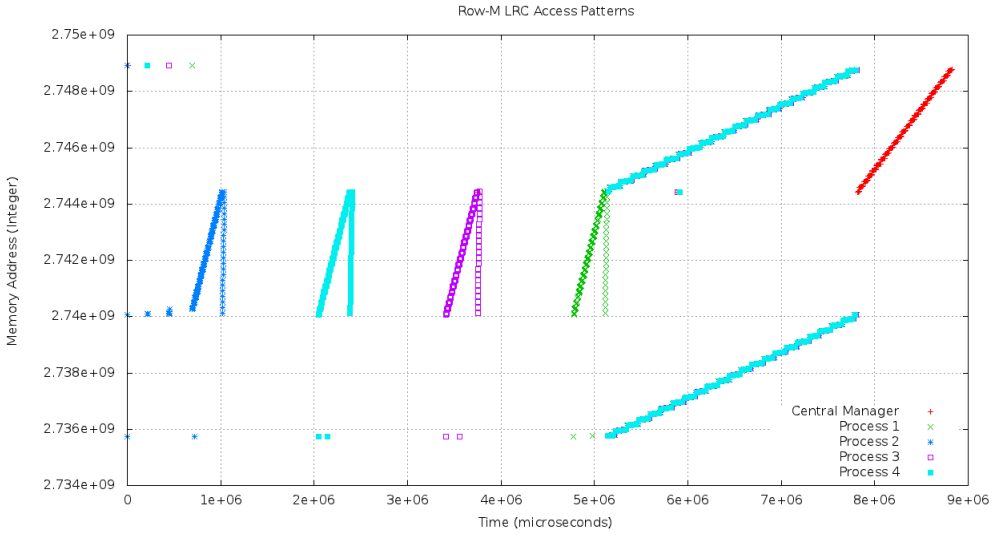

Figure 6.2: Access patterns exhibited by a *LRC* coherency model during the running of a matrix multiplication benchmark application accessing memory in row-major order.

One can see from the two figures the difference in access patterns. Both figures demonstrate how the matrix multiplication application accesses memory over time. Both applications begin by *warming up* their address space. After the address space is populated with data – either the input matrices or the output matrix – the threads begin executing.

Stricter coherency models require many more messages and read faults – as observed in Figure 6.1. The additional messaging observed in Figure 6.1 is due to the coherency protocol's inability to provide read-only pages, and is worsened by the model's requirement that all changes in the system be propagated to all other nodes in the system. The application which exhibits LRC coherency in Figure 6.2 demonstrates the coherency protocol's ability to maintain read-only pages and the model's efficiency in modification propagation. This is depicted by the series of memory accesses in Figure 6.2, and the reduced amount of memory contention over the output matrix.

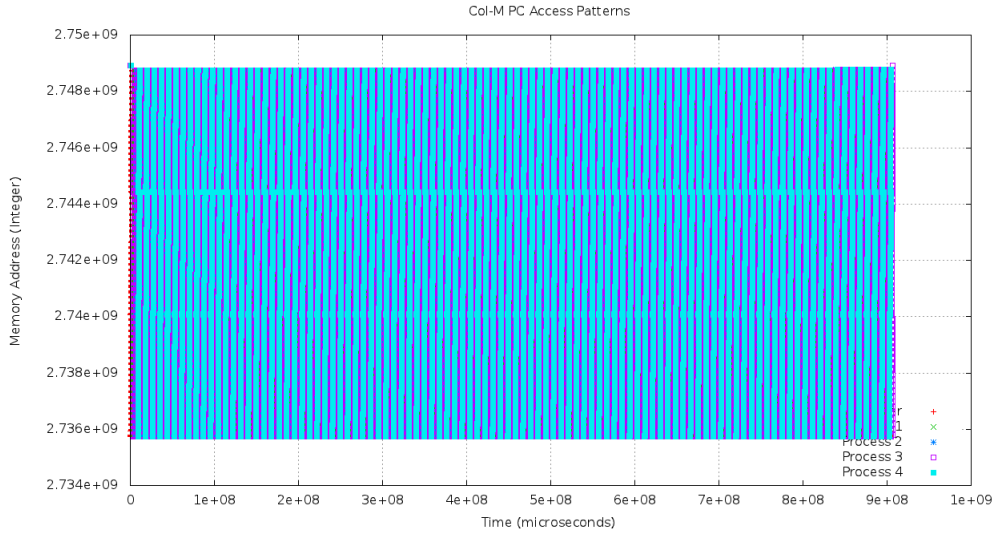

Figure 6.3: Access patterns exhibited by a *PC* coherency model during the running of a matrix multiplication benchmark application accessing memory in column-major order.

I observed that column-major allocation strategy did not improve performance or relieve memory contention over the result matrix in Figure 6.3 and Figure 6.4. In both scenarios, column-major accesses caused significantly more pages to be modified during an iteration of the matrix multplication algorithm. This caused significantly more data to be transferred between nodes. Perhaps, a multiple-writer protocol would yield different results for this type of access pattern.

This is strong evidence to why strict coherency models aren't typically used in DSM systems. Weaker coherency models facilitate the transfer of modified pages only, which has the advantage of requiring fewer messages and faults. The fewer number of messages and faults required to correctly execute minimizes the overall execution time.

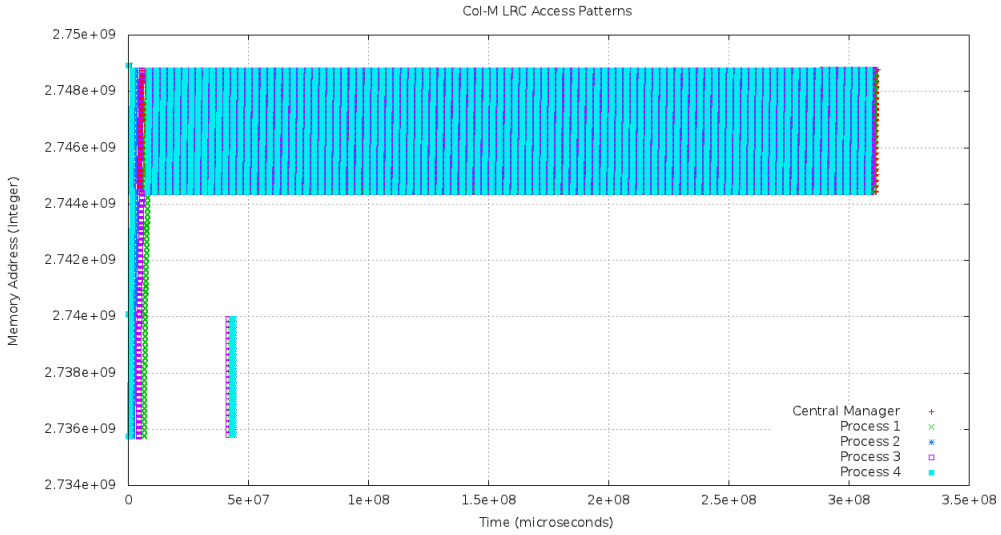

Figure 6.4: Access patterns exhibited by a *LRC* coherency model during the running of a matrix multiplication benchmark application accessing memory in column-major order.

6.2.2 Performance

I executed each of the matrix multiplication applications presented in §6.2 on two 400x400 matrices to mimic similar benchmarks executed on the Munin system in [17]. The authors of Munin report in [17] performance characteristics for matrix multiplication, among other various benchmarks. Unfortunately, the same code could not be obtained to benchmark against DSM64, but the matrix multiplication algorithm used in this paper aims to offer as generic of a solution as possible.

The number of messages required to correctly execute a program is an indicator of how well the system performs. Table 6.1 displays the average number of messages required to correctly execute a matrix multiplication algorithm (row-major or column-major access patterns) paired with a PC or LRC coherency protocol. These figures are similar to those presented in [13] which offer a comparison between Treadmarks using a SC and LRC models. Using 16 nodes, a

#	Row-M PC	Row-M LRC	Col-M PC	Col-M LRC
1	2142	1429	2142	1429
2	288536	2788	849670	283645
4	289170	3508	850304	285073
8	289730	4948	850758	287929
16	289894	7828	851030	587282

Table 6.1: Average number of messages required to correctly execute matrix multiplication applications (row-major and column-major access patterns) using PC or LRC memory coherency. The large number of messages required for applications using PC memory coherency is a result of memory thrashing.

#	Munin	Munin (SOR)	DSM64 (PC)	DSM64 (LRC)	Malloc
1	-	-	4.11	3.67	1.14
2	382.15	380.51	311.82	5.50	1.14
4	196.92	194.27	312.20	8.45	1.14
8	105.73	102.84	313.52	17.55	1.14
16	72.41	67.32	326.24	28.66	1.14

Table 6.2: Runtime performance characteristics in seconds of Munin running regular matrix multiplication, Munin running optimized matrix multiplication and DSM64 running a threaded version of matrix multiplication. All results are computed using an input size of two 400x400 integer matrices.

LRC row-major matrix multiplication requires 7828 messages. Munin was able to correctly execute the same test using only 1567 messages. The difference in the number of messages is due to the fact that Munin's worker threads calculate more than one row at a time. This allows a thread to perform more work (with the same amount of messaging overhead) before releasing the memory that it owns.

One can see that the row-major algorithm using LRC correctly executes with the fewest number of messages, suggesting that it also will perform the quickest. I measured that this was the case. Not only did the system perform the quickest, but also performed with the least amount of data transfer.

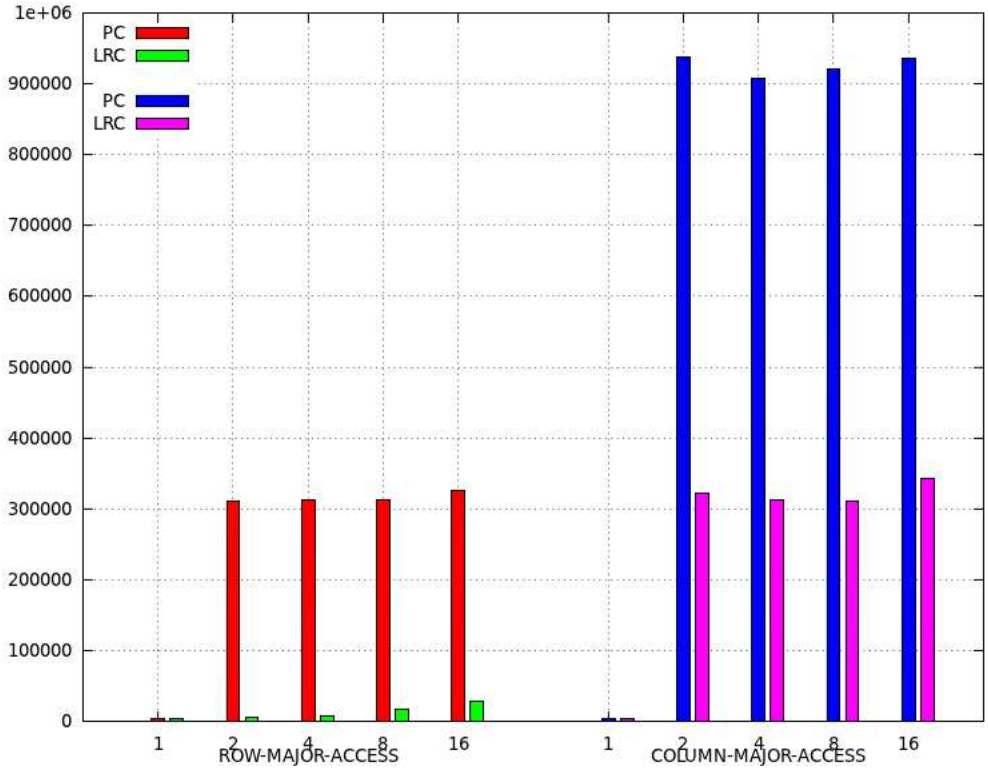

Figure 6.5: Runtime performance characteristics in milliseconds of DSM64 executing with 1, 2, 4, 8 and 16 threads for a matrix multiplication application access memory in row-major order and column-major order. The input size used for this sample was two 400x400 integer matrices.

The DSM64 system suffers from false-sharing over the result matrix. This is the primary reason that the results in Table 6.2 suggest that the more processors one adds to the cluster the longer DSM64 will take to execute the same program. This is to be expected: as n many processors compete for access to the result matrix, a page is faulted between nodes n many times. This is the primary reason for slowdown in the system. The only way to resolve this is to add a multiple-writer protocol (or find a way to resolve the shortcomings of the binary rewriter presented in this paper).

Additionally, I observed during testing that there is a base cost for each of n processors to *warm up* their address space. During this period, each processor

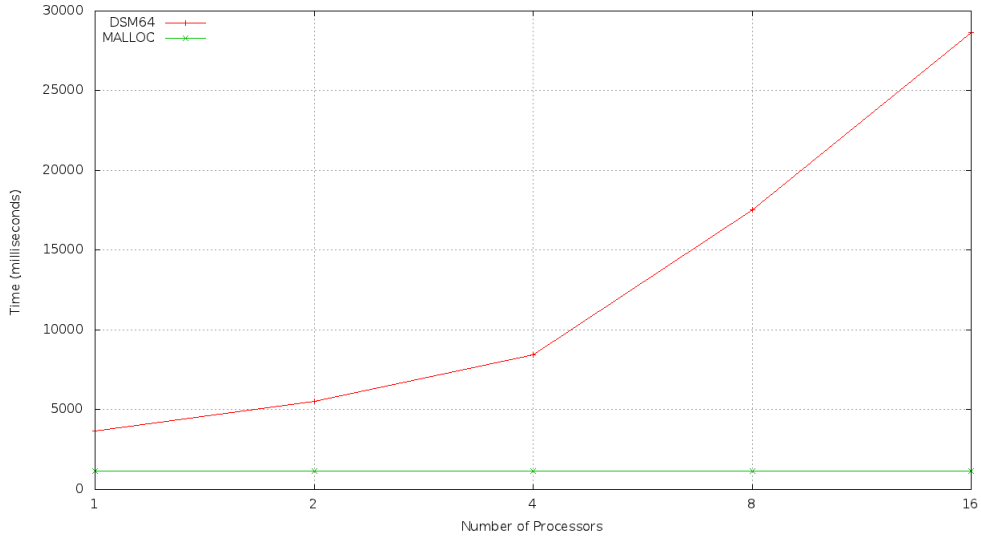

Figure 6.6: Runtime performance characteristics in milliseconds of DSM64 and the system-provided allocator executing with 1, 2, 4, 8 and 16 threads for a matrix multiplication application access memory in row-major order. The input size used for this sample was two 400x400 integer matrices.

faults in read-only copies of the input matrices. Due to the locking mechanisms present in the `pthread` application, this task is done serially and incurs noticeable slowdown as the number of processors n grows. For clusters running with 1 or 2 processors (e.g., $n = 2$) the warmup time was negligible. For clusters running with 4 processors (e.g., $n = 4$) the warmup time was roughly 2 seconds. For clusters running with 8 processors (e.g., $n = 8$) the warmup time was roughly 10 seconds. Lastly, for clusters running with 16 processors (e.g., $n = 16$) the warmup time was roughly 16 seconds. These figures contributed to the runtime of the system. One can observe the access patterns of this *warm up* period in Figure 6.2 in the time interval between 1 second and 5 seconds. This interval is the period of time each node spends faulting in the input matrices. This phenomena also occurred on the Midway system in [49] where identical results were reported.

The DSM64 system does not outperform the system-provided allocator for

several reasons. Firstly, DSM64 suffers from false-sharing. This fact aside, there are several other reasons that this is the case. The unaltered system-provided allocator's results were collected with an application that was run in a single address space on a single machine. This implies that the various threads would never compete for memory and never need to acquire memory via the network.

6.3 Contributions & Shortcomings

The work presented in this paper describes what I thought was an extremely rewarding and interesting problem. I was able to implement a complete lazy release consistent software distributed shared memory system in user-space. Implementing virtual memory managers provided insights into the underlying system workings that an application develop would otherwise be ignorant of. I feel that there are several items that I contributed to the state of the art. Namely, I compare coherency models and their effect on memory access patterns and performance for a matrix multiplication algorithm. Second, I provide supporting evidence of the SDSM performance characteristics provided by Treadmarks and Munin. Lastly, I presented a new SDSM system that extends Hoard: a fast, scalable, and memory-efficient allocator for shared-memory multiprocessors.

6.3.1 Effect of False-sharing

The DSM64 system, unlike its peers, does not experience speed-up with the addition of more computation nodes. This problem is apparent in the run-times for DSM64 in Figure 6.5 and Figure 6.6. In fact, the DSM64 system experiences slow-down with the addition of more computation nodes. This phenomena

deserves more discussion and is the topic of this section.

The first and most important question that must be addressed is why the DSM64 system suffers from false-sharing. The primary reason DSM64 suffers from false-sharing is the lacky of a multiple-writer protocol. DSM64 was originally designed to use a binary rewriter to coerce existing binaries to use an experimental data layout. The experimental data layout enforced a layout of memory in such a way that memory contention and false-sharing would be impossible because variables would reside on their own memory unit (e.g., memory page). Because such a layout allows variables to be addressed individually, no false-sharing can be experienced. If a node needs access to an element of memory to write to the output matrix, it can acquire that memory unit.

Several months were dedicated to developing the binary rewriter presented in this paper. Though the binary rewriter was excluded from the final deliverable, the reasons for doing so were limited. If a method for padding compound data types and rewriting indirect data references, the binary rewriter in this paper *could* be used to implement the experimental data type. In the case that such a process became possible, the VMM used by DSM64 could be extended with a refined *heap layer* that was responsible for organizing heap allocated data into the experimental data format. Work has already been done on this layer but it was excluded from the final deliverable when I discovered that the binary rewriter could not accomplish its goals.

The DSM64 system experiences slow-down because it suffers from false-sharing. The DSM64 system suffers from false-sharing over the result matrix when executing a matrix multiplication program. This phenomena was also experienced by Munin. The effect of false-sharing can be described as *memory thrashing* in that any memory is excessively transferred between any nodes that need access

to the page. Each fault that takes place requires 4096 bytes to be transferred, fault handlers to execute on both nodes, the memory protections updated and the page table adjusted. The thrashing causes the system to exessively communicate and seriously slows down the system.

Implementing the binary rewriter would allow the DSM64 system to experience speed-up instead of slow-down. If many processes could write to the same memory units at the same time, the system would provide comparable results to its peers. However, I feel until certain breakthrows in binary rewriting happen other alternatives need to be investigated. Namely, the use of multiple-writer protocols. A multiple-writer protocol would allow n many processors to simultaneously write to a common page, and merge those changes at a future synchronization point. The SDSM systems evaluated in this paper accomplish this task using diff-based approaches, that allow a processor to locally write a page and propagate only the page diff at synchronization events.

Another approach that would alleviate the effect of false-sharing would involve running more than a single execution thread on a distributed node. Considering that each node typically has a multicore processor, the DSM64 system could leverage this fact to improve performance by running more than a single process on each node. These processes that share the same physical hardware would share their address space and need not incur expensive read faults. Additionally, because each process would share the same memory protections on memory, each process could write to the same memory unit. This approach could dramatically reduce the number messages required to execute an application and should alleviate memory contention. This approach may also help DSM64 scale to larger clusters.

Appendix A

Executable and Linkable Format

Files that conform to the executable and linkable format (ELF) are considered executables, object files, shared libraries or core dumps and first appeared in System V specifications [46]. These terms are used interchangeably in this subsection. Today, the ELF format is used as the object file type in various Unix based operating systems. These file formats typically contain program header tables, section header tables and the data referred to by entries in the aforementioned sections [46]. This data is required to compile, link and load the application by the operating system.

The purpose of this section is to establish a set of vocabulary and background understanding of the system abstractions necessary to rewrite ELF binaries. Readers familiar with ELF format and function may wish to skip to §1.6.1.

A.1 Object Files

Object files are compiled binary representations of programs intended to execute directly on a processor. There are three main types of object files: *relocatable files*, *executable files* and *shared object files*. Relocatable files contain code and data suitable for linking with other object files to create an executable or shared object file. Executable files contain a program suitable for execution. A shared object file contains code and data suitable for linking in more than one context.

A.1.1 File Format

Object files are used for linking (building a program) and loading (running a program) executable applications, and as such can be observed from two different, but parallel, perspectives. These linking and loading perspectives are referred to as *views*.

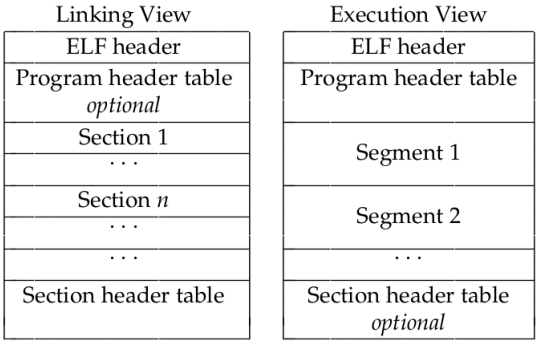

Figure A.1: Two parallel views of an ELF object file. On the left is the linking view of the object file. To the right, the execution view of an executable object file being loaded into memory.

The ELF header consists of a road map describing the file's organization and has a fixed location in the object file. The ELF header typically describes a set of *sections* that hold the bulk of the object file's information: instructions, program

data, symbol tables, relocation information and other things. The *program header table*, if present, tells a system how to create a process image and is required for all executable object files. This table describes the sections and their respective names, offset on disk and size. The data and structure types used to build the ELF header and the and the various section entries are defined in the ELF specification [46].

A.1.2 Sections

An object file's section header table describes all of the object file's sections. The section header contains information like the offset on disk, number of entries and the size of each header – the information necessary to index and read the object file's sections by the operating system at load time.

An object file can contain many sections, and arbitrary sections can be introduced to the object file with no adverse effects as long as the ELF header and *section header table* are updated accordingly. To link and load applications a certain set of headers must be present. A non-exhaustive list of the special sections present in an executable object file is presented in Figure A.2.

A.1.3 Symbol Table

A symbol table is a data structure used by the system interpreter to relate identifiers with definitions or declarations. These identifiers typically refer to data, data types, locations, functions, files and more. Using the information provided in the symbol table, the system interpreter may locate (or relocate) the symbolic information in an object file. Without the information provided in an object file's symbol table, linking an application composed of several object

Name	Description
.bss	This section holds uninitialized data that contributes to the program's memory image. By definition, the system initializes the data with zeros when the program is loaded. The section occupies no file space.
.data	This section holds initialized data that contributes to the program's memory image on disk. This section is *memcpy*'d into a process's address space at load time.
.fini	This section holds executable instructions that instruct the system how to terminal the process. This code is executed when the process exits normally.
.got	This section holds the global offset table. This section is described in more detail in section A.2.
.init	This section holds executable instructions that instruct the system how to set up the process. This code is executed before the *main* entry point is called.
.interp	This section holds the path name of a program interpreter for dynamically linking the program (e.g., GNU's *ld* linker).
.plt	This section holds the procedure linkage table used for dynamic linking. This section is described in more detail in section A.2.
.symtab	The section holds a symbol table. This section is described in more detail in section A.1.3.
.text	This holds the program's "text" (e.g., the executable instructions) to executed by the the system's processor.

Figure A.2: A non-exhaustive list and description of special *section* entries present in executable object files. An exhaustive list is provided in the ELF specification.

files would be impossible because the system interpreter would not be able to identify and address the individual components of each object file. Additionally, applications that depend on shared libraries would be impossible to load by a system interpreter without information that described the locations of "missing" functions by the object file.

Concretely, ELF symbol tables are composed of an array of structures which encapsulates symbol information like its name, value, size and descriptor flags.

The value in a symbol's structure can refer to an absolute value (e.g., global data) or an address (e.g., the location of a function). Each symbol also comes with a binding (or scope) instructing the system interpreter where the symbol may be applied.

A.2 Program Linking & Loading

There are three major components that must be addressed to understand program linking and loading: the *program header*, the process of *program loading* and *dynamic linking*. As discussed in section A.1, object files statically represent a program on disk. To execute an executable object file, the system interpreter must open and read the object file to construct the program in memory – this process `memcpy`s things like program data, program text, the program's stack and so on. This process also resolves undefined references (called *symbolic references*) to identifiers that the program did not define itself, as is often the case with program's that use shared libraries provided by the system.

A.2.1 Program Header

The program header is defined in the executable object file and is the primary data structure used by the program interpreter to identify and locate the segments present within that executable object file. The segments described by the program header typically encapsulate several of the sections described in section A.1. Using this data structure, the system interpreter can create an executable memory image for the process.

The program header itself is composed of an array of structures which describe

a segment's type, offset on disk, the size of the segment on disk, desired virtual address to be loaded at and various other descriptor flags.

The program header also contains various other bits of information that may or may not be relevant for a given object file. For instance, the program header contains a *base address* to instruct the system interpreter the lowest virtual memory address associated with the memory image of the program's object file. This information could be used to relocate segments during dynamic linking. Additionally, a note section is also described by the program header which no defined use if prescribed. Rather, developers and vendors are permitted to use these sections to include any information they would like.

A.2.2 Program Loading

Program loading refers to the process a system interpreter follows to load an ELF executable object file into memory. This involves opening and reading the object file on disk and `memcpy`ing the *segments* that compose the object file into a process's virtual memory address space. Using the ELF header table an object file's segments are loaded into the virtual memory address space at prescribed locations – this process is depicted in figure A.4 and figure A.3.

A process loaded into memory is not a mirror image of the executable object file on disk. Special locations exist for the various segments (which themselves are composed of various sections) which compose the object file. Additionally, a structural format dictates areas designated for padding, areas for the run-time stack to operate in as well as areas for dynamic memory allocation to take place (e.g., the heap). These regions are typically page-aligned in the process's virtual memory address space.

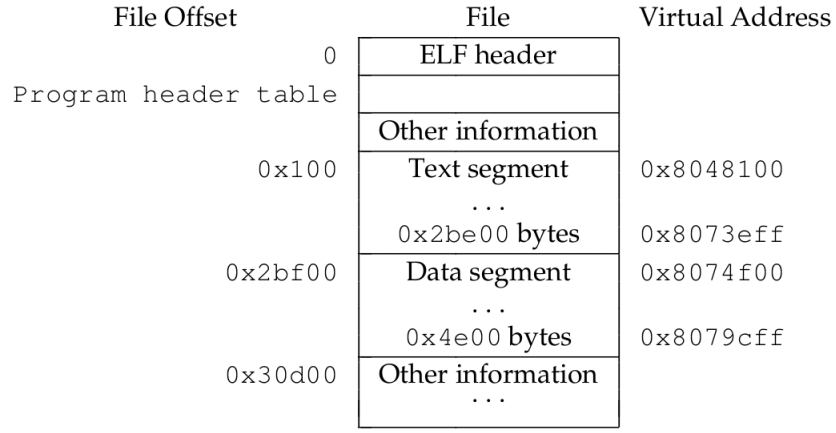

Figure A.3: Two parallel views of an object file defined by a ELF header table described in figure A.4.

Member	Text	Data
p_type	PT_LOAD	PT_LOAD
p_offset	0x100	0x2bf00
p_vaddr	0x8048100	0x8074f00
p_filesz	0x2be00	0x4e00
p_memsz	0x2be00	0x5e24
p_align	0x1000	0x1000

Figure A.4: An example ELF header table used to construct the object file in memory for figure A.3 and figure A.5.

Functionally, separating different segments of an executable object file makes sense for several reasons. First, this separation allows the system to enforce memory permissions as if each segment were separate and complete. For instance, this allows a process to set its `.text` to be read-only (a basic security feature provided by most operating systems). Secondly, this allows for large contiguous areas of virtual memory addresses to be reserved for the run-time stack and heap.

Figure A.5 depicts a loaded ELF executable object file (again, this figure assumes the use of the ELF header table in figure A.4).

As previously mentioned, a loaded process is composed of several parts.

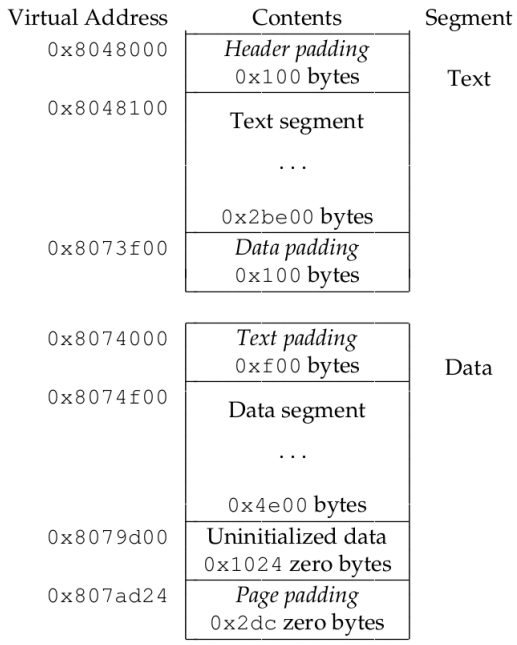

Figure A.5: An ELF executable object file loaded into memory.

Specifically, the following memory regions are set aside for ELF processes. The first `.text` page contains the ELF header, the program header table and other information. The last `.text` page holds a copy of the beginning of data. The first data page has a copy of the end of `.text`. The last data page may contain file information not relevant to the running process. Additionally, an area is reserved for uninitialized data (recall from section A.1 the special section `.bss`). This information is used for sanity checking.

A.2.3 Dynamic Linking

Even when an executable object file is completely loaded into memory, it is likely still not ready to be "run". This is because application programs oftentimes depend on linked dependencies that must be added to the application program dynamically at load time. It is the task of the dynamic linker to resolve these

undefined references (called *symbolic references*) by loading shared libraries into the process's virtual address space. This is when the DSM64 library is loaded.

Object files that require dynamic linking notify the system interpreter of this at load time by including a program header of type PT_INTERP – this header notifies the system interpreter which dynamic linker it should use to correctly resolve any symbolic references. The specified dynamic linker then is responsible for loading the executable's memory segments to the process image, adding shared object memory segments to the process image, performing relocations and closing the object file's file descriptor. To assist the dynamic linker in achieving these tasks, several data structures provided by the compiler are used. Namely, the `.dynamic`, `.got`, `.plt`, among other various sections (these are additional special sections discussed in §A.1).

The function of these special sections is to provide a roadmap for mapping position-independent code to absolute virtual memory addresses. Using the `.got` section – the global offset table – and the `.plt` section – the procedure linkage table – the dynamic linker `memcpy`s static and shared program text and makes respective updates to the data structures.

Bibliography

[1] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7:321–359, November 1989.

[2] Emery D. Berger and Robert D. Blumofe. Hoard: A fast, scalable, and memory-efficient allocator for shared-memorymultiprocessors. Technical report, Austin, TX, USA, 1999.

[3] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35:117–128, November 2000.

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

[5] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 15–26, New York, NY, USA, 1990. ACM.

[6] Willy Zwaenepoel, John K. Bennett, John B. Carter, and Pete Keleher. Munin: distributed shared memory using multi-protocol release consistency.

IEEE Comput. Soc. Tech. Comm. Newsl. Oper. Syst. Appl. Environ., 5:11–, December 1992.

[7] Robert C. Steinke and Gary J. Nutt. A unified theory of shared memory consistency. *J. ACM*, 51:800–849, September 2004.

[8] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th annual international symposium on Computer architecture*, ISCA '87, pages 234–243, New York, NY, USA, 1987. ACM.

[9] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. *SIGARCH Comput. Archit. News*, 14:434–442, May 1986.

[10] James Goodman. Cache consistency and sequential consistency. In *IEEE Scalable Coherent Interface (SCI) Working Group*, 1989.

[11] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28:690–691, September 1979.

[12] Daniel J. Scales and Kourosh Gharachorloo. Towards transparent and efficient software distributed shared memory. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 157–169, New York, NY, USA, 1997. ACM.

[13] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th annual international symposium on Computer architecture*, ISCA '92, pages 13–21, New York, NY, USA, 1992. ACM.

[14] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui

Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29:18–28, February 1996.

[15] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 10–10, Berkeley, CA, USA, 1994. USENIX Association.

[16] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: distributed shared memory based on type-specific memory coherence. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, PPOPP '90, pages 168–176, New York, NY, USA, 1990. ACM.

[17] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of munin. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, SOSP '91, pages 152–164, New York, NY, USA, 1991. ACM.

[18] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: a low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ASPLOS-VII, pages 174–185, New York, NY, USA, 1996. ACM.

[19] Sandhya Dwarkadas, Peter Keleher, Alan L. Cox, and Willy Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. In *Proceedings of the 20th annual interna-*

tional symposium on computer architecture, ISCA '93, pages 144–155, New York, NY, USA, 1993. ACM.

[20] Leslie Lamport. Ti clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[21] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9:21–65, February 1991.

[22] Eyal de Lara, Y. Charlie Hu, Honghui Lu, Alan L. Cox, and Willy Zwaenepoel. The efeect of contention on the scalability of page-based software shared memory systems. In *Selected Papers from the 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, LCR '00, pages 155–169, London, UK, 2000. Springer-Verlag.

[23] Zhou Yuanyuan, Iftode Liviu, and Li Kai. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *USENIX 2nd Symposium on OS Design and Implementation (OSDI '96)*, jan 1996.

[24] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Trans. Comput.*, 36:388–395, April 1987.

[25] Sivarama P. Dandamudi. Reducing hot-spot contention in shared-memory multiprocessor systems. *IEEE Concurrency*, 7:48–59, January 1999.

[26] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. In *Proceedings of the*

15th ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '10, pages 269–280, New York, NY, USA, 2010. ACM.

[27] Tor E. Jeremiassen and Susan J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pages 179–188, New York, NY, USA, 1995. ACM.

[28] Elana D. Granston and Harry A. G. Wijshoff. Managing pages in shared virtual memory systems: getting the compiler into the game. In *Proceedings of the 7th international conference on Supercomputing*, ICS '93, pages 11–20, New York, NY, USA, 1993. ACM.

[29] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 281–289, New York, NY, USA, 2003. ACM.

[30] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[31] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 272–280, New York, NY, USA, 2003. ACM.

[32] Haizhi Xu and Steve J. Chapin. Improving address space randomization with a dynamic offset randomization technique. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 384–391, New York, NY, USA, 2006. ACM.

[33] Aparna Kotha Matthew Smithson, Kapil Anand. Binary rewriting without relocation information. In *Technical Report, University of Maryland, November 2010*, 2010.

[34] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[35] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Trans. Program. Lang. Syst.*, 27:882–945, September 2005.

[36] Jeff Hollingsworth Barton Miller. Dyninstapi offical type @ONLINE, June 2011.

[37] Jeff Hollingsworth Barton Miller. *ParseAPI Programmers Manual*. Paradyn.

[38] Jeff Hollingsworth Barton Miller. *SymtabAPI Programmers Manual*. Paradyn.

[39] Jeff Hollingsworth Barton Miller. *InstructionAPI Programmers Manual*. Paradyn.

[40] Jeff Hollingsworth Barton Miller. *StackwalkerAPI Programmers Manual*. Paradyn.

[41] Jeff Hollingsworth Barton Miller. *DynC Programmers Manual*. Paradyn.

[42] Jeff Hollingsworth Barton Miller. *ProcControlAPI Programmers Manual*. Paradyn.

[43] E. Speight and J.K. Bennett. Using multicast and multithreading to reduce communication in software dsm systems. In *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, pages 312 –322, feb 1998.

[44] Yannis Smaragdakis and Don S. Batory. Mixin-based programming in c++. In *Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers*, GCSE '00, pages 163–177, London, UK, 2001. Springer-Verlag.

[45] Emery David Berger. *Memory management for high-performance applications*. PhD thesis, 2002. AAI3108460.

[46] Jeff Hollingsworth Barton Miller. *Portable Formats Specification Version 1.1*. Tools Interface Standards (TIS).

[47] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Trans. Comput. Syst.*, 13(3):205–243, August 1995.

[48] Ricardo Bianchini, Mark E. Crovella, Leonidas Kontothanassis, and Thomas J. LeBlanc. Alleviating memory contention in matrix computations on large-scale shared-memory multiprocessors. Technical report, Rochester, NY, USA, 1993.

[49] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The midway distributed

shared memory system. In *Compcon Spring '93, Digest of Papers.*, pages 528 –537, feb 1993.