INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700  800/521-0600
RICE UNIVERSITY

I/O-Oriented Applications on a Software Distributed-Shared Memory System

by

Timothy Paul Parker

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Dr. Alan Cox, Chairman
Associate Professor
Computer Science

Dr. Peter Druschel
Assistant Professor
Computer Science

Dr. Devika Subramanian
Associate Professor
Computer Science

Houston, Texas
February, 1999
Abstract

I/O-Oriented Applications on a Software Distributed-Shared Memory System

Timothy Paul Parker

This thesis evaluates the use of a software distributed shared memory system, Treadmarks, as a platform for supporting an I/O-intensive application, specifically the database Postgres. Software distributed shared memory (DSM) systems allow applications to run on cheap and powerful networks of workstations without the complexity of explicit message-passing. Such systems are usually used for computationally-intensive scientific applications. I/O-intensive applications have significantly different characteristics.

Despite this, Postgres needed only minimal changes to run on Treadmarks. This is partially because we wrote emulation layers for many APIs Postgres already used. We created additional support pieces for Treadmarks to handle the problems that arose because of the different application characteristics. These were divided into three areas. Some issues were related to usage of forms of communication other than shared memory, some to Treadmarks, and some to the UNIX API. We discuss and evaluate solutions to these problems.
Acknowledgments

There are several people I wish to thank for their assistance. Obviously, the person most responsible for this document other than myself is my advisor, Alan Cox, to whom I wish to express sincere thanks for kind and patient support as well as for his direct contributions. The suggestions provided by the other members of my committee made this a much better document than it would otherwise have been. I also wish to thank fellow students Kai Zhang and Chen Ding, as well as Intervarsity Christian Fellowship, for their encouragement during difficult times. My officemate Eyal de Lara put up with the mess I made of much more than my half of our office and other idiosyncrasies. Lastly, I might have been completely unable to do this without the support of my loving family.
## Contents

Abstract ......................................................... ii
Acknowledgments .................................................. iii
List of Tables .................................................... vi
List of Illustrations ............................................. vii

### 1 Introduction

1.1 Background .................................................... 2
   1.1.1 Treadmarks .................................................. 2
   1.1.2 Postgres .................................................... 3
1.2 Thesis Statement and Contributions .......................... 4
1.3 Outline of Thesis ............................................. 5

### 2 Postgres

2.1 Design and Pieces ............................................ 7
2.2 Query processing example .................................... 8

### 3 Treadmarks

3.1 General Overview ............................................. 10
   3.1.1 Interface ................................................ 11
   3.1.2 Implementation ......................................... 11
3.2 Pertinent Details ............................................ 12

### 4 Postgres on Treadmarks: PG-TMK

4.1 High-Level Design .......................................... 14
4.2 Emulation Layers ............................................ 15
   4.2.1 Process creation: transfer agent and stub .......... 15
   4.2.2 System V semaphore and shared memory emulation .... 16
4.3 Issue Categories ............................................ 18
   4.3.1 Treadmarks-implementation-related problems ..... 18
4.3.2 Non-shared-memory-based communication ................................ 20
4.3.3 UNIX API issues ........................................................................ 21
4.4 Performance Cost .......................................................................... 22
  4.4.1 Periodic barriers and safe locks .................................................. 23
  4.4.2 Wrappers .................................................................................. 23
  4.4.3 Transfer Agent ........................................................................... 24
4.5 PG-TMK Performance ...................................................................... 24

5 Related Work .................................................................................... 26
  5.1 Database-Specific ......................................................................... 26
    5.1.1 Global memory systems ......................................................... 26
    5.1.2 Distributed-shared memory ..................................................... 27
  5.2 Related Distributed and Parallel OS Work ...................................... 29

6 Conclusions ....................................................................................... 31
  6.1 Contributions ................................................................................. 31
  6.2 Future Work .................................................................................. 33

Bibliography ......................................................................................... 34
Tables

4.1 Safe locks vs. conventional Treadmarks locks on 166MHz Pentium Pro
Illustrations

2.1 Basic design of Postgres. Lines represent network connections. Note that the postmaster and engines are all on the same machine. The postmaster exists primarily to start up engines. ........................................ 7

2.2 The Postgres Backend. Adapted from Momjian [13] ............................... 9

4.1 Basic design of PG-TMK. All lines represent network connections. Only the postmaster and the transfer agent need execute on the same machine. .......................................................... 15
Chapter 1

Introduction

In this thesis, we evaluate the use of software distributed shared memory as a platform for supporting I/O-intensive applications, specifically a distributed database. Software distributed shared memory (DSM) provides a shared memory abstraction on a network of computers, even though the computers do not physically share memory. Historically, software DSM systems have been used for scientific applications requiring little network or disk I/O. Commercial workloads, such as databases, differ from scientific applications in that there is heavy use of multiple forms of I/O, often simultaneously. This difference exposes several problems in the existing DSM implementation.

The contributions of this thesis include the following. We first present a categorization of the problems encountered in adapting our software DSM to this class of applications. We then present issue solutions and demonstrate that it is possible to run I/O-intensive applications on a software DSM with minimal changes to the application code. This is desirable because modifications to the application code would have to be performed for each new application. Some augmentation of the software DSM system was required, but these additions can be reused or modified in the future.

We chose to use a database system for two major reasons. First, databases are parallelizable and can productively use large amounts of memory. A cluster of computers makes significant aggregate CPU power and memory bandwidth available to the database at a reasonable price. The shared memory provided by the software DSM system gives the database a larger memory space for caching data. This is effective even when the data is in another machine's memory because fetching data pages from a remote machine is considerably faster than from disk.

The second reason we chose to use a database is that databases are a widely used I/O-intensive application. Databases make heavy use of multiple forms of I/O, perhaps more than any other common application. From a performance perspective, database systems are extremely demanding of the systems on which they run.
Databases are probably the most common substrate for commercial applications and the major reason commercial users purchase expensive hardware systems.

1.1 Background

We chose Treadmarks as our software DSM. Treadmarks is an efficient page-based software DSM system from Rice University. For a DBMS that is sufficiently powerful to be representative of serious commercial databases we chose Postgres. Postgres is likely the most advanced DBMS whose source code is freely available.

1.1.1 Treadmarks

Many applications today require more computing power than uniprocessor workstations can deliver. Enterprise-class servers are not as expensive as supercomputers, but they are far from inexpensive. The small SMP machines in the commodity price range are often not powerful enough, and are not as widely available as common workstations or PC-class machines. Thus, computing on a network of workstations is still a desirable alternative.

Writing programs to run on a network of workstations by hand is complicated and tedious, because data has to be explicitly communicated by message passing. Software DSMs can simplify this by providing the simpler paradigm of shared-memory computing on commodity workstations and PCs. These systems, such as Treadmarks, are typically used for scientific applications. Scientific applications are computationally-intensive but involve little use of system calls or forms of communication other than the shared memory.

Broadly speaking, the benefit of a software DSM system from an application's perspective is to greatly increase the amount of computer power available for a given price. Rather than buy a single expensive box with as much performance as possible, software DSM facilitates running the application over a cluster of commodity systems. These can be chosen so as to be at the point of greatest value on the price-performance curve. Because the individual machines have their own processors and memory buses, an application can make use of the aggregate memory bandwidth and CPU power. Memory bandwidth is a particularly important factor for I/O-intensive applications such as databases.

We chose Treadmarks as our software DSM system. Treadmarks is a page-based DSM that uses the lazy release consistency relaxed memory model, which allows it
to reduce the amount of network communication. It uses an invalidation protocol that defers the communication of the application's data until necessary and has a multiple writer protocol that allows different processors to write to the same page simultaneously. Treadmarks incorporates other optimizations to decrease expensive communication and increase performance. It runs on many common UNIX flavors and does not require modification of the kernel.

1.1.2 Postgres

We chose the DBMS Postgres as a representative example of a powerful database. It is a full-featured database originally developed by Michael Stonebraker's group at the University of California–Berkeley. Although it is not a commercial database, it is the only open-source database powerful enough to be similar to the commercial systems.

For Postgres, the goal of the port was two-fold. First, we hoped Treadmarks would enable a larger database to be memory-resident, by utilizing the aggregate memories of the workstations as a single memory. For some time it has been true that a page could be brought from a remote memory faster than from disk. A larger database could be made memory-resident than would fit in the physical RAM of a given machine by using Treadmarks to access remote memory.

Secondly, we hoped the parallelization would increase performance by increasing the throughput of the database system. This can happen without decreasing the response time for a query considered in isolation. According to Olson et al. [14], a DBMS can exhibit several types of parallelism. The original Postgres architecture, which will be explained in detail in Chapter 2, supports only inter-query parallelism, which allows queries to operate simultaneously. We did not attempt to significantly modify this architecture. Instead we merely allowed the database processes to run on separate machines. Thus we do not support intra-query parallelism, which allows a single query to be parallelized, nor inter-function and intra-function parallelism, which allow user-defined procedures to be parallelized. Because this is the only form of parallelism we support, our parallelized system cannot make queries faster. We can only increase throughput, which is the number of queries that can be executed in a given period of time.

Note Treadmarks allows Postgres to run on a shared-nothing architecture (meaning a cluster of machines which do not physically share memory or disk resources, e.g., networks of workstations or personal computers). According to the Asilomar
report on database research [3], shared nothing systems have better scalability characteristics for database systems, as first argued by Michael Stonebraker [20]. In a later paper he estimates a network of smaller machines such as personal computers (as we used) would be 2 orders of magnitude cheaper than a large single machine—one of two reasons virtually all database systems will eventually be distributed [22].

1.2 Thesis Statement and Contributions

Our thesis is that although I/O-intensive applications are significantly different from the applications that software DSM is designed to support, it can effectively support such applications without significant modification to them. To show this, we port the DBMS Postgres to Treadmarks to learn about the issues involved in this fundamentally different type of application. I/O-intensive applications are focused more on the input and output of data than on performing computation based on the data. As such, they are less likely to be numerically-intensive and more likely to be operating-system intensive. An example of a difference with significant effect is that while many scientific applications use the single-program, multiple-data (SPMD) form of parallelism, Postgres does not use this style of parallelism. The interaction of this with our software DSM Treadmarks caused significant complications, as will be discussed. Our experimental work provides a framework for the future porting of I/O-intensive applications.

Since Postgres uses the System V APIs for shared memory and semaphores, as well as the standard UNIX calls for process creation, we wrote an emulation layer for these. After this we dealt with a number of problems of different types. Some problems were caused by the nature of the application itself, such as problems relating to the heavy use of system calls for disk and network I/O. Other problems were related to the design of Treadmarks, such as some complications arising from the need to perform periodic global synchronization. Some problems were related to the nature of the UNIX API, including difficulties caused by the semantics of signals.

We employed a number of solutions for these problems. We built a new lock mechanism to address some Treadmarks-related issues that could cause deadlock. Wrappers we wrote for some I/O system calls dealt with problems from all three categories. We added a new process to the system, which we called the transfer agent, in order to deal with problems related to UNIX and the forms of communication.
1.3 Outline of Thesis

We begin the thesis with an overview of Postgres, an open-source database first developed at the University of California-Berkeley. The next chapter focuses on Treadmarks, the software distributed-shared memory system we used. The following chapter describes PG-TMK, our version of Postgres that runs on Treadmarks. We examine related work before concluding.
Chapter 2

Postgres

Postgres is a relational database system that provides typical database services, including transactions. It was originally developed at the University of California at Berkeley by a group led by Michael Stonebraker. The name derives from the fact that it's an intellectual successor to Ingres, which was developed by the same group. The original Postgres used a non-standard query language called Postquel; Postgres95 was a rewrite to support SQL, and PostgreSQL is the open development version.

The Postgres DBMS introduced several innovations. Rules, procedures, and object-relational concepts were some of them [12]. Object-relational databases not only provide the ability to store complex objects, but also allow these objects to be queried. Often, these queries use user-defined procedures to operate on a specific user-defined type. Since these procedures can be inherited by subtypes, they are methods in the sense of a conventional object-oriented programming language. Michael Stonebraker describes the concept and motivation of an ORDBMS (object-relational DBMS) in an Informix white paper [23]. Postgres also supports a storage system that abstracts the storage devices and their properties. The original design did not use write-ahead logging, which is commonly used for transaction integrity, but a different technique that also allows for access to past states of the database [21]. This access to historical states, called "time-travel," is being removed from the open-source PostgreSQL version.

More relevantly for our purposes, Postgres is one of the premier open-source databases, so we could obtain the source code. Moreover, it had previously been used for a study of memory behavior on a multiprocessor, so we knew it could run TPC-D, an important read-intensive database benchmark, in parallel [24].
Figure 2.1 Basic design of Postgres. Lines represent network connections. 
Note that the postmaster and engines are all on the same machine. The 
postmaster exists primarily to start up engines.

2.1 Design and Pieces

Postgres is based on a "process-per-client" model, meaning that for each client there 
is a corresponding database process. It allows simultaneous queries via running si-
multaneous processes.

The first piece of Postgres started up is the postmaster. This program waits 
for clients to connect, usually via TCP. A client session is created by passing this 
connection off to a newly spawned engine. Since the UNIX process creation calls 
fork and exec do not close file descriptors, allowing them to be inherited instead, 
the postmaster simply leaves the TCP connection socket open for the child to use.

Engines listen to their inherited connection to the client, and process any queries 
or other SQL commands issued by the client. Engines communicate with each other 
via shared memory and semaphores, using the corresponding System V APIs.

Shared memory is used to create a shared cache of disk buffers. This allows a 
process to read from cached data used by itself or another database process without 
invoking the operating system. A single lock, BufMgrLock, controls access to these 
shared buffer structures. All reads and writes of database data use the shared buffers. 
Shared memory also contains the database data locks, which are used to prevent 
conflicting accesses to the same tuples. Although read locks may be done at the
granularity of a tuple, write locks cannot be done at a granularity finer than a relation. This limits transaction performance, and so the PostgreSQL development team is implementing a new locking system with finer granularity [12].

2.2 Query processing example

We examine the processing of an individual query to demonstrate the operation of Postgres in more detail. The query begins at a client, which must have TCP/IP or local connectivity with the server. Suppose someone starts up a Tcl front end and requests a listing of all graduate students. Tcl communicates with the DBMS processes using libpq, a Postgres client interface.

First, a database session is created by sending a startup packet to the postmaster, via TCP/IP. The postmaster creates an engine process to handle this new client. The engine does not make its own TCP connection back to the client but simply inherits the TCP connection originally made to the postmaster.

Tcl now gives the SQL query "SELECT * FROM STUDENTS WHERE CLASS = GRAD" to the client part of the libpq library, which sends a query packet over the inherited TCP connection. The engine parses the received query, identifying it as non-trivial in the traffic cop module. The form of the results is determined, while checking for views or related rules. The optimizer creates an optimized plan whose exact sequencing is determined by the path module. Finally, the plan is executed by the executor module. Since the database has its own internal disk cache buffers, some pages may be read directly from those without invoking the operating system. If pages are not already in the cache, they are first read into the cache before reading them from the cache as before. Thus all database data is read from the cache and never directly from disk. The results are available to the client, which fetches them. [13]
Figure 2.2 The Postgres Backend. Adapted from Momjian [13]
Chapter 3

Treadmarks

Treadmarks is a leading software distributed-shared-memory system. Distributed shared memory systems provide the illusion that multiple processors share a memory region even though they may not be physically wired to the same banks of RAM [1]. Shared data structures can be referenced in the same way as local data structures, requiring only the locking normally required for safe access to shared structures. This programming model is simpler for the programmer than requiring the explicit communication of data by including message-passing calls in the code. Message-passing requires the programmer to decide when to communicate, what to communicate, and to whom it should be sent. [1]

A software DSM allows shared-memory programming on clusters of workstations without the special hardware support found in hardware multiprocessor machines. Instead, Treadmarks traps into the virtual memory subsystem, so that it can fetch remote copies of pages only when a processor actually uses the corresponding memory locations. We first give a general overview of Treadmarks then focus on some particularly relevant details.

3.1 General Overview

Treadmarks uses a release consistency relaxed memory model [1]. This means that memory accesses are only guaranteed to be consistent when guarded by an acquire-release pair. Most programmers are used to the sequential consistency memory model, where there is some consistent total ordering of memory accesses. This is what is found on a multiprogrammed uniprocessor. In Treadmarks, an access to a shared structure must be preceded by an acquire on a corresponding lock, and may not become visible until after a release on that lock.
3.1.1 Interface

The Treadmarks API contains a small number of functions. `Tmk_malloc` allocates shared memory (note that all memory allocated via `malloc` will be private memory). The `Tmk_lock_acquire` and `Tmk_lock_release` calls acquire and release critical section locks, which are exclusive locks. Another synchronization operation, `Tmk_barrier`, is used for system-wide synchronization and also allows garbage collection of version data. Because this synchronization is system-wide, no process which has entered a barrier proceeds until all processes complete the barrier.

Programs access shared memory via the normal programming language operations. Any page which the local processor does not have the most recent version of (according to the semantics of the release-consistency model used) is marked inaccessible. Whenever a processor attempts to access memory in that page, a page fault will occur, which Treadmarks receives because it registers itself as handler for the corresponding UNIX signal. Treadmarks programs normally communicate mainly via this shared memory. There is another call, `Tmk_distribute`, that can be used to push data into a specified private memory location on all processes.

Normally, the user specifies the number and location of processes for a Treadmarks program by listing machines in the program invocation. There is a mechanism for creating processes at runtime, called `Tmk_spawn`. Locks and barriers have a manager process, which is where messages about those synchronization operations are sent.

3.1.2 Implementation

Treadmarks uses a form of the release-consistency relaxed memory model called lazy release consistency (LRC) which delays the change notification until the next acquire on the same lock, rather than notifying all processors on the lock release. Moreover, Treadmarks uses an invalidate protocol, so that this change notification is merely used to mark pages as no longer valid. Actual data updates (called diffs because they are the difference between the previous version of the page and the new one) are only distributed on demand, i.e., when a processor attempts to access the stale page. So a processor invalidates pages when it does an acquire on a lock corresponding to a data structure, and then actually fetches data only when it accesses a particular page.

Treadmarks has a mode for the multiple-writer case, where multiple processors are writing to a given page. Multiple-writer mode significantly decreases the performance cost of false sharing, a problem that can occur when two processors are both writing to
data items on the same page. Note that if they are actually writing to the same datum, the program has a memory race. Treadmarks adaptively selects multiple-writer mode or single-writer mode for individual pages based on the program's behavior [2]. Most of these optimizations are designed to reduce the amount of communication, which represents the primary cost of Treadmarks.

A processor which acquires a lock previously held by another processor needs to know which memory locations were modified by the other processor between its acquisition and release of the lock. This time period is called an interval, and can be thought of as a description of the changes to the world as a given processor knew it while holding the lock. A sequence of intervals, beginning from a known state, describes a unique world. A 16-bit timestamp is used to identify intervals. This timestamp can be reset when a barrier happens. Since a barrier synchronizes all processes, they can agree on a single worldview and not have to remember which intervals are associated with which locks (i.e., they garbage-collect the interval data).

Not only does Treadmarks run on commodity hardware, but it also runs under unmodified common UNIX operating systems without special compiler support. The implementation is merely a user-level library with signal handlers that serve as "hooks" into the UNIX operating system. Programs run on Treadmarks by linking against the library and making the appropriate calls to the Treadmarks interface.

3.2 Pertinent Details

We will discuss four aspects of Treadmarks that were particularly relevant to porting Postgres. One, Treadmarks requires that applications periodically execute a barrier for reasons described below. It also makes use of certain signals that can interact with the application's operation. Third, lock and barrier managers are allocated at system start time. Fourth, spin locks are not efficient under lazy release consistency.

Treadmarks requires applications that make heavy use of locks to barrier periodically. One reason for this is the finiteness of the 16-bit timestamp; if it rolls over there are then distinct intervals with duplicate timestamps. The existence of communication cliques can also necessitate periodic barriers. Such cliques occur when a subset of the processes does not interact with another subset for some time. Each subset can use relatively short messages to exchange interval data among its members. However, when a member of one subset tries to communicate with a member of the other, it must inform the other of each of the many intervals which have occurred since their
last exchange. For speed and simplicity, Treadmarks uses a single UDP packet to communicate this information, but this fails if there are more intervals to list than can fit in a single packet.

Treadmarks relies on intercepting certain signals to operate effectively. Previously we mentioned that Treadmarks uses a signal handler, either for SIGSEGV or for SIGBUS depending on the platform, to intercept page faults. Another signal, SIGIO, is used to handle asynchronous events, such as a request from another process for a lock or for consistency information. This allows Treadmarks to handle these events on demand without a separate thread of execution. SIGALRM is used to provide a timing “heartbeat” that interrupts certain blocking networking calls to provide a timeout. All of these signals can occur during the execution of the application code.

There are a couple of additional details about Treadmarks that are particularly relevant. One, managers for locks and barriers are statically allocated at system start time, so dynamically created processes do not share the burden of managership. This could lead to load imbalance. Secondly, some applications use spin locks, which repeatedly check a memory location to see if the contents has changed. Since LRC only guarantees memory updates when acquiring a lock, the lock acquire and release must appear inside the spin loop. This is inefficient in a Treadmarks program, because it can cause polling over the network.
Chapter 4

Postgres on Treadmarks: PG-TMK

We call our Treadmarks version of Postgres PG-TMK. As Postgres is approximately a quarter-million lines of code by itself, a major goal for the Treadmarks port was minimal modification of the Postgres code. Knowing that Postgres used the standard fork-and-exec combination for creating engine processes, and the System V API for shared memory and semaphores, we began by creating emulation layers for these.

This chapter begins with a high-level design overview of the PG-TMK system, then details the creation of the emulation layers. The various problems that we had after writing the basic infrastructure are discussed in the context of their categorization. Lastly, we look at the performance cost of the new infrastructure pieces we wrote, and then the performance of PG-TMK.

4.1 High-Level Design

Given the multiprocess nature of Postgres, where each engine runs as a separate process, we decided the appropriate design for a Treadmarks version was to allow each engine to run on its own machine. This would allow it to use as much of the machine's resources as necessary without having to contend with another database process. The shared memory cache that had been shared between processes on the same machine could be provided by Treadmarks instead so there was a shared cache between the engines on different machines. This allows us to take advantage of all the memory in the cluster as our cache.

Rather than rewriting Postgres, it was simpler to write emulation layers for system calls that it used. We wrote an emulation layer for the process creation calls Postgres used to create engines. This included adding a new piece to the system, a proxy process called the transfer agent. Postgres used the System V API for shared memory to provide the shared cache, and the semaphore API for locking to prevent conflicting accesses to the data, so we wrote an emulation layer for these.
Figure 4.1 Basic design of PG-TMK. All lines represent network connections. Only the postmaster and the transfer agent need execute on the same machine.

4.2 Emulation Layers

4.2.1 Process creation: transfer agent and stub

Remember that in the normal Postgres system, engines inherit the connection to their clients from the postmaster when they are created via fork and exec. This is important because this is how the client communicates with its database engine to send queries and receive results. A single UNIX system supports passing file descriptors from parent to child this way or between arbitrarily-related processes via a mechanism built into UNIX domain sockets. However, conventional UNIX systems do not include any kind of support for passing file descriptors between UNIX systems.

In order to remedy this, our emulation layer for process creation includes a separate program called the transfer agent, or “ta.” The ta acts as a proxy, forwarding SQL commands to the newly-spawned engine via a TCP connection and receiving results over that same connection. Our replacement for exec operates as follows. When the postmaster calls exec, our code passes the file descriptor meant for the engine to the ta process. The ta interprets this as a request to spawn off a new engine, connected
to this file descriptor. Instead of spawning the new engine directly, a stub program
is invoked which makes a TCP connection back to ta and then dup's that socket
descriptor into the appropriate descriptor index. The stub then uses the normal exec
to invoke the engine, which now has a socket descriptor that it can communicate on
to the engine normally, oblivious of the new link in the chain of TCP connections.
The only change to the postmaster application code that was required was a call to
join the Treadmarks system. The fork system call is unchanged. Note the ta process
is a likely bottleneck if there are many processes communicating much data over these
descriptors.

There are alternative solutions to the problem that we did not implement. Rather
than having a separate proxy process and program, an application could incorpo-
rate a proxy thread. However, there are complications associated with converting a
single-threaded OS-intensive application to a multi-threaded model, for example when
using non-thread-safe system calls. We didn’t want to attempt to rewrite Postgres.
However, in order to avoid the ta bottleneck completely, it is necessary to change
the application or the kernel. The application could be changed by modifying the
start-up protocol so that a client receives the TCP address and port number of the
engine process from the postmaster, so that it can make a direct connection. A more
general way to replace ta would be to add kernel support for “passing” file descrip-
tors between machines. For disk files, a distributed file system might provide the
functionality of being able to share a reference to a file. A more elegant solution
for our particular case would have been to “hand-off” the TCP connection from the
postmaster machine to the engine machine. This requires intricate manipulation of
the TCP protocol, see Pai [15] for use of this for a distributed Web server.

4.2.2 System V semaphore and shared memory emulation

Since Postgres uses the System V semaphore and shared memory APIs to provide
the shared buffers and their locking, we wrote an emulation layer to make these
APIs work in a Treadmarks system. An important design point was to avoid the use
of polling in the following case. When a semaphore counter does not have sufficient
balance to fulfill a request, semantics require the operation to block until enough units
are available. The common Treadmarks synchronization primitives, mutex locks and
barriers, do not provide any way to asynchronously notify a process to wake up.
Instead, a process would have to either block on a particular lock or poll by acquiring
a lock, checking the memory value guarded by that lock, and releasing it. Since the semaphores could have values greater than the minimum decrement unit, they couldn't be implemented simply as a mutex. (Unfortunately, newly-added support for condition variables couldn't alleviate the problem, because blocking on the conditional variable wait caused deadlock in the same manner as blocking on a lock acquire, as will be discussed in 4.3.1.1.)

We designed two semaphore implementations. The simpler one is best for semaphores which are binary in value (for example, ones used for a mutex lock) or are rarely used. Because we could not block on locks (or condition variables), which were the only mechanism we had to put an individual process to sleep, we had to resort to a modified busywait. Access to the semaphore's value is in a critical section protected by a Treadmarks lock for that semaphore. To block inside semaphore wait, a process spins on an array entry corresponding to it, sleeping for a brief period of time and checking to see if it has been notified to barrier. The releaser of the semaphore uses Tmkl.distribute to change the array entry, telling the process to wake up.

The more complicated design is appropriate for frequently used semaphores whose maximum value is significantly greater than the normal delta. This is most easily explained by examining the motivation. Postgres used semaphores for the readers-writers problem. These semaphores were initialized to 255. A process requested a read lock by asking for a single unit of the semaphore, and a write lock by asking for all 255 units. This ensured that multiple readers could have the lock at once, but any writer would hold the lock exclusively. However, a particular one of these semaphores, the previously-mentioned BufMgrLock, was very heavily used; in fact, it was acquired for every read of a tuple. Conventional semaphore designs, such as the former one, would end up trading a Treadmarks lock back and forth for every semaphore operation not on the same machine as the previous operation, even when the cause was just two simultaneous readers.

We devised a distributed semaphore system that could allow consecutive read locks without communication. The idea was essentially to allow each machine to hold part of the semaphore balance locally. A machine could then grant a read lock from the units in the local balance. Write locks were gained by acquiring the balance from all machines. Since a writer had to get all the units, and a reader had to have some unit, no writer could exist while a reader did. In fact, because units were neither created nor destroyed, we preserved normal semaphore semantics fully. Whenever a machine required more units than it had, it would acquire the exclusive thievery lock
and then attempt to steal units from other machines. The thievery lock serialized theft to prevent deadlock and simplify the implementation.

The emulation layer for shared memory was much simpler. Most of the System V shared memory calls were implemented as a small wrapper around the corresponding Treadmarks calls. Some shared state did have to be maintained, as the API allows the user to associate each piece of shared memory with a unique key.

4.3 Issue Categories

After building the emulation layers, we dealt with a number of other problems. The issues that we addressed in making Postgres function properly on top of Treadmarks can be divided into three categories. Some of the problems were related to some of the design and implementation choices in the current version of Treadmarks. Others were a consequence of the heavy use of non-shared-memory communication and file I/O. A few problems arose because of the design of the UNIX API.

4.3.1 Treadmarks-implementation-related problems

The design and implementation choices in the current version of Treadmarks caused some difficulties for this kind of application. The relevant issues were our application's heavy use of locks, dynamic process creation and termination, and having distinct process binary images.

4.3.1.1 Barriers and deadlocks

The most complicated problem related to the implementation of Treadmarks was caused by the need to use locks to implement semaphores. Remember that Treadmarks requires processes that use locks to periodically perform a barrier. This garbage collects interval data to prevent the timestamp from overflowing. It also prevents message overflow caused by the communications clique that the engines in Postgres form.

Adding such barriers in most scientific applications and computationally-intensive applications is not difficult, because they typically use the SPMD (single program, multiple data) model of parallelism. Adding such barriers in PG-TMK is more difficult, because the system is comprised of three independent programs which choose to cooperate. External synchronization had to be imposed on this collection of unrelated processes. So we decided to have barriers happen with a fixed frequency. We
designed a system where the ta process kept track of time and notified other processes to perform a barrier via Tmk.distribute.

This caused deadlock in two ways. The first way it induced deadlock was with the use of Treadmarks locks. If a process blocked on a Treadmarks lock acquire, and the process holding the lock went into a barrier, neither could progress. Since barriers update the same data that locks generate and rely on for consistency, Treadmarks could not be re-entered from the lock acquire to join the barrier. Nor could the lock acquire fail. Deadlock was induced in a more insidious manner through blocking I/O calls. An engine process could block on a read from ta, perhaps because it was waiting for a command. Then ta could block on a barrier, guaranteeing it wouldn’t feed the engine, which would not barrier until after the read succeeded.

Each deadlock-inducing condition required a separate solution. Solving the blocking lock-acquire problem required us to eventually drop any Treadmarks locks unless they provably don’t block. Locks are only used around short critical sections which can be guaranteed to finish. Checks for barriers are kept outside those sections. To ameliorate the blocking I/O call deadlocks without liberally modifying the application, we had to provide wrappers around the blocking I/O calls. Rather than actually blocking on a read or write syscall, the wrappers call select to evaluate readability or writability. When the select times out, they check for the need to barrier before selecting again. This code had to have a check to avoid re-entering itself, since the barrier code itself uses I/O calls.

4.3.1.2 Dynamic process creation and termination

Some of the issues we addressed dealt with dynamically creating and destroying Treadmarks processes. Treadmarks has a simple mechanism for creating additional processes at runtime, but in some ways it is designed more for special tasks than to create (and destroy) full peer processes.

One problem is that the C library function exit terminates the Treadmarks system, because there is no mechanism for a process to leave the system. So when an engine is ready to terminate, it simply sleeps forever, because if it actually called exit the system would terminate.

Another problem is that lock and barrier management are not dynamically allocated during system runtime. Processes created by spawning, which is the dynamic process creation facility, are not eligible to be managers. PG-TMK only has one orig-
inal process; the postmaster is spawned immediately and the engines are spawned
on demand. Because of this, all locks and barriers are managed by the first pro-
cess, which performs reasonably in PG-TMK because we do not rely on ta to do any
computation.

4.3.2 Non-shared-memory-based communication

Many of the problems we had to solve resulted from the use of various types of commu-
nication other than shared memory. This included file I/O, networking, and standard
forms of interprocess communication, and less obvious communication mechanisms
such as the previously-mentioned inheritance of file descriptors across a fork-and-
exec combination.

4.3.2.1 Spin locks

A common form of interprocess coordination is spin locks, which "spin" checking for
a particular memory location to change value (i.e., they poll the checked location's
value). Spin locks do not provide synchronization under memory models which are
only release-consistent, unless the location is guarded with acquire-release. However,
this produces polling that requires a message for each poll. Postgres contains a semi-
phore implementation of spin locks, since the direct implementations it has to support
multiprocessor systems are machine-dependent. So we modified the defines to not rec-
ognize the direct implementation and revert back to the semaphore implementation
of the spin-lock functionality. This was simple, but increased our dependence on
the semaphores. Note that since the semaphore implementation requires some com-
unication, these spin locks are not nearly as cheap in PG-TMK as the hardware
implementations.

4.3.2.2 Environment propagation

A common way for UNIX processes to communicate information, usually static set-
tings, is via the use of environment variables. These variables are propogated from
parent to child during exec. However, the environment is not propogated by most net-
work process-creation techniques, and Postgres uses environment variables for several
purposes. We handled this problem by hardcoding most of the applicable environment
variables, and passing a few when starting up the network connection. Occasionally
the database code had to be changed to deal with a static value for some of these
variables. For example, when the database system experiences some types of fatal engine errors, it kills all old engines and starts new ones with a different key for the session so that old engines won’t mistakenly be allowed to continue to participate. A better solution would have been to allow the programmer to specify which variables to pass, or perhaps even all (allowing existing host-specific settings to be retained).

4.3.3 UNIX API issues

Some of the issues we dealt with were most directly a consequence of the UNIX API. These were related to the operation of signals with respect to each other and some system calls.

4.3.3.1 Signals

One signal-related problem is that ”slow” system calls get interrupted by signals, notably SIGIO and SIGALRM. The slow system calls are those which can block for some period of time, typically on I/O, and include read and write. When a signal arrives during one of these calls, the call returns prematurely indicating signal interruption as an error. This is not intrinsically a problem, except that if the application did not anticipate receiving a signal during the call, it may react as though the call failed. Many of the read calls in Postgres do exactly this, during the occurrence of SIGIO and SIGALRM signals generated by the Treadmarks routines. The wrappers call the system call again in this case. (Normally an application can specify that the slow system calls should automatically restart themselves if interrupted. However, this would cause the program to effectively block on those calls. We could not allow this, since we had to periodically check whether it was time to barrier (see section 4.3.1.1 on barriers and deadlock), so we had to write wrappers which retried in the case of interruption and did the check before each retry.)

Clearly, it would be better to entirely avoid the use of signals visible to the application thread. Unfortunately, we could not do this efficiently. Treadmarks sets the timer which signals SIGALRM to go off periodically, irrespective of whether in Treadmarks library code or not. (This is used to cheaply make network calls timeout to retransmit UDP packets; it avoids the overhead of setting the timer every time.) The use of SIGIO to handle asynchronous events (other processes asking for a lock or diff) could be avoided by having a separate thread to handle these. However, it
could not opt to necessarily receive SIGIO itself, and synchronization of the threads would be expensive.

Another problem is that signals are not compositional. It only makes sense for a process to register a single handler for a signal. When different pieces of a single application wish to use the same signal for different purposes, they must have some mechanism to disambiguate the different meanings of the signal. Signals themselves do not provide this information. In our case, when Postgres and Treadmarks wished to use the same signal, we simply disabled one of the uses of the signal. This meant we lost some deadlock detection via timeout in Postgres. Had Postgres actually required SIGALRM or SIGIO to function, we would have had a complicated problem, probably requiring us to rewrite the application or rewrite Treadmarks in a less efficient manner.

4.3.3.2 Pages for system calls must already be local

All pages involved in system calls must be local first. In other words, user-level page-fault handlers can’t be used to fetch pages remotely inside the system call. The kernel disables the Treadmarks user-level page-fault signal handlers when entering kernel mode. While this is reasonable kernel behavior for security and efficiency, it means that I/O calls on shared memory indicate a bad memory address operand when the memory isn’t already local. In our case, this frequently happened when data was being read into a buffer in shared memory.

The solution to this problem was to wrap the appropriate system calls to touch all of their pages prior to actually making the system call. Then we invoke the system call directly using the syscall() mechanism. This became a race condition. In some "slow" system calls the call could be interrupted by a Treadmarks signal handler that might make the page inaccessible again. This would happen because of the receipt of new consistency information indicating the page was not valid. So if the call fails because of an invalid memory address, we try again. A possible optimization, depending on the common memory behavior of a program, would be to not touch pages until the call failed the first time.

4.4 Performance Cost

We added several mechanisms to the base Treadmarks in order to support our non-traditional application. We analyze the costs of these mechanisms, to consider recommending their inclusion in the base Treadmarks system.
4.4.1 Periodic barriers and safe locks

We imposed a periodic-barrier mechanism, as discussed in section 4.3.1.1. The cost is negligible unless a process actually does have to use it to barrier periodically. Since Treadmarks requires the process to pay this barrier overhead anyway, it does not represent any significant cost increase over the normal Treadmarks system.

To prevent the deadlock associated with using a normal blocking Treadmarks lock acquire in conjunction with the periodic barrier mechanism, we created the safe lock implementation. The acquire and release primitives incorporate standard Treadmarks locks to provide release consistency. Because they are built using standard Treadmarks primitives, they are significantly more expensive than those calls themselves. The measured times are in Table 4.1; they effectively represent the worst case.

The distributed semaphore implementation is built on the safe lock primitives. The safe locks provide all synchronization used by the distributed semaphores. Since the distributed semaphores have no algorithmic complexity of their own and very little communication, their cost is not significantly greater than that of the safe locks. Measuring the distributed semaphores themselves is an extremely difficult problem, because the timing would have to account for skew between the clocks of different computers and varying network latencies.

4.4.2 Wrappers

Several system calls, such as read and write, were wrapped to provide varied additional functionality. Examples of the functionality provided are in sections 4.3.1.1, 4.3.3.1, and 4.3.3.2. Any process which uses these calls would have to pay this overhead. We tested 100 writes and 100 reads of 4k on a disk file whose file pointer was

<table>
<thead>
<tr>
<th>Distance</th>
<th>Type</th>
<th>Tmk Time</th>
<th>Non-Block Time</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>remote</td>
<td>release</td>
<td>49 usec</td>
<td>859 usec</td>
<td>17x as expensive</td>
</tr>
<tr>
<td>remote</td>
<td>acquire</td>
<td>277 usec</td>
<td>607 usec</td>
<td>2.2x as expensive</td>
</tr>
<tr>
<td>local</td>
<td>acquire</td>
<td>7 usec</td>
<td>31 usec</td>
<td>4x as expensive</td>
</tr>
<tr>
<td>local</td>
<td>release</td>
<td>7 usec</td>
<td>15 usec</td>
<td>twice as expensive</td>
</tr>
</tbody>
</table>

Table 4.1 Safe locks vs. conventional Treadmarks locks on 166MHz Pentium Pro
reset between each operation, ensuring the data remained cached. This eliminated physical disk I/O so that we measured the time for the OS call itself. The 200 normal calls took 37 microseconds each on our Pentium Pro 166-MHz platform. The 200 calls using the wrappers took 45 microseconds each, about 25% longer. Since I/O calls are often used to actually do uncached disk I/O, or network I/O, the typical cost factor will be smaller.

4.4.3 Transfer Agent

The cost of the transfer agent to programs which do not pass file descriptors across fork-and-exec is limited to the startup cost, which is normally ignored. Recall the transfer agent is required because file descriptors normally inherited across fork-and-exec cannot be passed between machines, as discussed in section 4.2.1. For programs that do use inherited file descriptors, the cost is the slowdown for communication passed across that descriptor, which is forwards via TCP. This latency increase is basically the cost of adding an additional TCP connection in the communication chain.

We measure this cost on our cluster of Pentium-II machines, which has a switched 100Mbps Ethernet, on an unloaded system. Latency was measured by sending a packet large enough to exactly fill the Ethernet frame one direction, answered by a small reply message. We timed the interval from the sending of the first message until the reception of the reply as 85 microseconds. We sent the reply message because otherwise it is very difficult to get a number from TCP representing the interval required to send the packet, particularly since TCP does not guarantee the remote machine will immediately acknowledge receipt. Because this is a round-trip time, it is an upper bound on the latency for data flowing one direction.

4.5 PG-TMK Performance

We tested the performance of PG-TMK by comparing it against the performance of a standard uniprocessor version of Postgres on the TPC-D database benchmark [6]. On a single query stream, PG-TMK must perform slower than Postgres because it has additional overhead, such as Treadmarks and the transfer agent, but not any ability to parallelize (because the queries are run consecutively on a single engine). So PG-TMK is 15% slower than Postgres for a single query stream.
However, PG-TMK takes three times as long to run a pair of query streams as it does to run one stream. Normally this would represent a 50\% slowdown. However, PG-TMK is now running on two engine machines, so that if it had perfect speedup two query streams would take exactly the same amount of time as a single query stream on a single engine. Thus, PG-TMK has a slowdown factor of 3 when running two streams.

We attribute this slowdown to two factors. Postgres uses a single exclusive lock for all access to tuples. In other words, only one process can access data from the database at any point in time. Since accessing tuples is an extremely frequent operation for a database, contention for this lock badly impairs scalability. Worse, not only does this lock experience heavy contention, but it must be traded back and forth remotely, meaning that each tuple access has a significant probability of requiring communication.
Chapter 5

Related Work

This chapter examines the most similar work to ours that we could find. Little work exists on running databases or other I/O-intensive applications on software DSM systems specifically. We begin with work that is specific to databases, which is generally conceptually similar but may not be in terms of implementation. We then discuss efforts that have purely implementation similarities in making operating system calls work on top of clusters.

5.1 Database-Specific

We examine two types of related database-specific work. Global memory systems deal with making effective use of the memory of a cluster considered as a whole, but this work usually has significantly different implementations from ours. Software DSM systems provide the abstraction of shared memory over a cluster, as previously discussed.

5.1.1 Global memory systems

From the perspective of the application, our work bears some fundamental similarities to global memory system support for DBMS systems work done at the University of Wisconsin-Madison. Originally they considered the problem of dealing with caches in client-server systems without making replacement decisions based on the content of other caches [10]. Franklin et al. [8] deals with global memory management in client-server systems, where the clients can access pages from each other as the third level in the memory hierarchy, after the local and server memories. Their simulation does sharing at the page level, as we do; but because they tie more closely into the DBMS they cache locks (optimistically or with the possibility of invalidation) as well as the data itself. In a previous work [10] they had noticed the efficacy of simple caching schemes was significantly reduced by duplication between the client and server or individual clients. Thus they examined global memory schemes that could avoid
such difficulties, which generally improved performance. These schemes are based on the concept of minimization of replication, which is not currently integrated into Treadmarks because it is designed for applications which are more concerned with fast access to smaller amounts of data than trying to manage a larger amount of data globally. This client-caching model has some similarities to our architecture since we have a one-to-one correspondence between engines, where caching is done, and clients, although since these are the server processes we do not have a separate server level as well.

A more applicable model is a multi-server model with caching done on each server, which is examined in Venkataraman [27]. The dissertation uses relatively simple global memory management policies to provide improved performance in a simulation. These policies reduce duplication while attempting to fully utilize idle memory and ameliorate the network contention that was problematic in the previous work. They also investigate using the local disk of the clients as another level in the memory hierarchy [9]. PG-TMK can use the local disk for caching, but it happens only when the VM system picks a shared page as the best candidate for eviction to disk.

Chen and Roussopoulos [5] simulates the adaptive allocation of buffers using feedback from query access patterns. Their method outperforms the methods they compare it to, including being significantly better for random access reference patterns.

5.1.2 Distributed-shared memory

While there has been a considerable amount of general work on software DSM systems in the recent past and some work on database memory performance on multiprocessor systems, not much work has been done on running database systems or other applications that make intensive use of operating system functionality on DSM. We located two works closely related to what we did.

Feeley et al [7] discuss making a transactional DSM system. They begin with CMU's Recoverable Virtual Memory package [16], which provides support for a persistent store in memory backed by disk logs for recovery. They add distributed locks and broadcast log updates to all machines in the system to provide coherency and create a DSM. Their work is similar in that it can be viewed as operating on a local cache of data which is accessed directly in the virtual memory space of a process. They do not provide full database functionality. There are no real similarities in
terms of the actual issues they dealt with; just the overall conceptual similarity of their application.

The work by Scales and Gharachorloo [18] does address many of the issues we dealt with from a slightly different perspective. Much of the perspective difference is due to the substantially different design of their software DSM, Shasta. Shasta implements fine-grained memory consistency, at the cache block-level, using a directory-based invalidation protocol [17]. It rewrites hardware multiprocessor binaries, placing a miss check at each memory reference (which can often be optimized in practice). Shasta supports the Alpha’s memory model and can use the low-latency Memory Channel architecture to pass messages without invoking the OS. No kernel support is required. Shasta includes support for efficiently using the hardware shared memory on individual SMP machines.

Their goal was also to run a parallel database over a cluster with their DSM. Since they didn’t need source code, they used the commercial Oracle database, widely recognized as a standard. They ran some queries from TPC-D.

Scales and Gharachorloo divided their support for Oracle (and similar applications) into three parts. First they dealt with system call arguments. The protocol code had to obtain the memory referenced by system call arguments before the kernel was entered, since they did not attempt to modify the kernel (remember they have to place a miss check at each memory reference). They cannot guarantee they will retain exclusive (i.e., write) access to lines once they request them, so on the next entry into protocol code they reissue any stores whose lines reverted state. PG-TMK simply touches each page involved with a read and write. Since we can’t lock pages to a given state either, we repeat the system call if it fails due to a protection violation (which would happen if the page is stolen away from us).

Their second issue was making system calls function properly on clusters. An emulation layer handled process management calls, such as fork, wait, kill, pid_block, pid_unblock, and getpid. The fork emulation copied the memory state of a process. Unique global process ids were generated to avoid duplication of process ids between nodes. PG-TMK did not require this much complexity because there were no non-trivial uses of the process id and process control calls. PG-TMK did require supporting the inheritance of file descriptors across fork-and-exec, which Oracle did not. The System V shared memory emulation was similar in both systems. There is no mention of a semaphore layer in their work; perhaps the Alpha’s memory barriers (fences) are sufficient when combined with the atomic read-write-modify in-
structions. As an approximation of a coherent distributed file-system, both systems NFS-mounted the same volume at the same location.

"Handling complex process graphs" [18] is the third issue they deal with. They first describe an ideal policy for their system, which entails having a "protocol process" on every node. This process can answer requests if there are no other processes on the node and accepts consistency data bequeathed when a process terminates. Their actual implementation is a simplified version of this, having one Shasta process per node which communicates with dynamically created processes using essentially the same mechanisms they use for processes on an SMP. The PG-TMK solution is better in that nodes don't have to have a process until necessary, but worse in that processes are not ever allowed to actually die until the system terminates.

Differences between our approaches are that Scales and Gharachorloo did not address signals such as SIGIO and the page-fault signal, because Shasta does not require them. Nor was it necessary for them to deal with barriers and the related deadlock problems. Oracle did not require the transfer agent, presumably because there was no inheritance of file descriptors, nor the semaphore layer.

5.2 Related Distributed and Parallel OS Work

Other researchers have made I/O and other operating system calls, or even entire operating systems, function across clusters. Broom [4] overviews implementing I/O calls on a Fujitsu AP1000 multi-computer, including a distributed file cache (whereas our distributed cache is at the application level) and a plan to investigate using weaker memory models such as used in software DSM systems. Another AP1000 project at the Australian National University does parallel process management [25] under AP/Linux. Here their "root node" had a bottleneck potential similar to our transfer agent, since it accepts all connections.

Sun Labs has implemented a cluster version of virtually the entire Solaris kernel, known as the Solaris MC project [11]. Their CORBA-based prototype is compliant with the Solaris Application Binary Interface (i.e., it runs binaries unmodified) and can run many Solaris executables. It provides the illusion that the cluster is a single system. Load-balancing is supported at process creation, but migration was not supported at the time of writing. Similar to our work, they did not fully implement file descriptor sharing and the full semantics of fork. Additional details about the distributed process management framework may be found in Shirriff [19]. The GLUnix
project [26] at U.C. Berkeley was to provide a cluster operating system with a single image that did support process migration.
Chapter 6

Conclusions

In order to evaluate the use of software DSM as a platform for I/O-intensive applications, we ported the database Postgres to the software DSM Treadmarks. Only minimal modifications were required to the source of Postgres. Postgres on Treadmarks successfully runs a TPC-D workload on multiple computers.

6.1 Contributions

We demonstrate that it is possible to run an I/O-intensive application with minimal changes to the application code. Since any changes we had to make to this application might have to be made to other applications as well to port them, it is important to minimize the required application changes. Less than one percent of the Postgres code required modification, and many of these changes were repetitively placed around C library I/O. Our changes usually deal with readily-identifiable sections of the code, such as the program beginning and specific system calls. Because of this, extensive knowledge of the inner workings of the application should not be necessary. This simplifies porting, especially for developers who are not already intimately familiar with the application.

We did have to add some new pieces to Treadmarks, but most of these can be reused in the future. The transfer agent might require modification to handle multiple inherited file descriptors. It is a potential bottleneck for applications that intensively use the inherited file descriptors. For these it should written out of the system entirely by changing the application code.

We created a categorization of the problems encountered in adapting our software DSM to the class of applications. Some of the issues we dealt with were related to the Treadmarks implementation. Others were because of the heavy use of forms of communication other than shared memory. Finally, some are a consequence of the UNIX API.
We employed a number of solutions for these problems. We created a periodic-barrier mechanism to address some Treadmarks-related issues, as well as non-blocking locks to avoid related deadlock. Wrappers we wrote for some I/O system calls dealt with problems from all three categories. We added a new process, the transfer agent, to forward a file descriptor's traffic between UNIX nodes, solving problems related to UNIX and the forms of communication.

We examine experimental results that show our modifications could be added into the base Treadmarks, at some cost. The I/O wrappers slow down I/O system calls by about 25% of the minimum cost of such a call. The safe lock operations, on which the distributed semaphores are built, can be as much as two to 17 times more expensive than normal Treadmarks lock operations. The transfer agent adds a less than 100 microsecond latency each time data is sent across a file descriptor it handles. TPC-D runs about 15% slower on the Treadmarks version of Postgres. However, it exhibits a slowdown when running engines on multiple machines, due to a single exclusive lock that prevents simultaneous access to data in the database.

We learned several lessons about making this type of application work on a software DSM system. Porting Postgres to Treadmarks was significantly more difficult than we had anticipated. We did not expect to have to write a transfer agent piece. For applications which do not use the inheritance of file descriptors over fork and exec, this piece is not necessary. Applications which have extensive traffic on these inherited file descriptors would be better served by a rewrite than a proxy piece like the transfer agent. However, rewriting applications can be difficult and time-consuming. Alternatively, a modified OS kernel could support intermachine descriptor passing or even the "handing off" of a live TCP connection to one machine from another.

We also did not expect to spend more than a month grappling with deadlock induced by the requirement for barriers. This problem can be entirely avoided by reworking Treadmarks so as to allow system-wide interval garbage collection initiated by a single processor to happen automatically on others. This is currently being investigated. We also recommended propagating the values of all (non-machine-specific) environment variable settings in the exec emulation, rather than only passing the values of a few specific variables as we did.
6.2 Future Work

We discovered several opportunities for future work in the course of the project. An obvious extension for Postgres itself is to rework it so as to not use an exclusive lock for the buffer manager. However, this would require reworking undocumented locking requirements for data structures spread across several modules and large quantities of code. We abandoned this after determining it would require a significant reworking of the database code. Another application-specific opportunity would be to add a level of indirection for queries. They could then be redirected to engines which had handled queries over same tables in the recent past and are likely to have those tables cached in local memory. It would be desirable to rework Treadmarks so as to garbage collect interval data without requiring the system-wide blocking synchronization a barrier requires, thus completely avoiding the deadlock problems. Lastly, it might be useful to examine a broad set of I/O-intensive applications to determine representative characteristics.
Bibliography


