

UNIVERSITY OF CALIFORNIA
IRVINE

Spontaneous Process Migration with Global Pointers

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of
DOCTOR OF PHILOSOPHY
in Information and Computer Science

by

Koji Noguchi

Thesis Committee:
Professor Lubomir F. Bic, Co-Chair
Professor Michael B. Dillencourt, Co-Chair
Professor Alex Nicolau

2007

© Koji Noguchi, 2007

The dissertation of Koji Noguchi is approved
and is acceptable in quality
and form for publication on microfilm:

Committee Co-Chair

Committee Co-Chair

University of California, Irvine

2007

Contents

List Of Figures	vii
List Of Tables	xi
Acknowledgments	xii
Curriculum Vitae	xiv
Abstract	xv
1 Introduction	1
1.1 Target Problems and Architectures	1
1.2 Message Passing	3
1.3 Distributed Shared Memory (DSM)	5
1.4 Process Migration	6
1.5 Research Objective	8
2 Language Construct for Distributed Dynamic Data Structures	10
2.1 Autonomous Implicit Strong Migration	11
2.1.1 Autonomous vs. Passive Migration	11
2.1.2 Explicit vs. Implicit Hop	12

2.1.3	Weak vs. Strong Migration	12
2.2	Location-aware Pointer Allocation	15
2.3	Location-transparent Pointer Handling	15
2.4	Process Creation	17
2.5	Communication and Synchronization through Global Shared Heap	18
3	Spontaneous Process Migration	20
3.1	Daemon	22
3.2	Global Shared Heap	27
3.2.1	Sharing Memory Space among Processes	27
3.2.2	spm_malloc and spm_free	30
3.2.3	Remote Pointer Dereference	31
3.3	Agent Process	31
3.3.1	Agent Startup	32
3.3.2	State Transfer	33
3.3.3	State Snapshot	37
3.3.4	Signal Handler	38
3.3.5	Locating a Remote Memory	39
3.4	Migration Example	39
3.5	Miscellaneous Enhancements	42
3.5.1	Tracking Process/Data Movement	42
3.5.2	Ghost Process	45
3.5.3	Parallelism	47
3.5.4	User Interface	47
3.5.5	Program Numbering	49
3.6	Requirements and Limitations	51

3.6.1	Architecture	51
3.6.2	Limitation on Transparency	52
3.6.3	Snapshot Efficiency	54
4	Remote Read/Write	57
4.1	Ping-Pong Effect	58
4.2	System Implementation	59
4.2.1	Single Line Interpreter	60
4.2.2	Parallelism on Remote Write	61
4.3	Runtime Heuristic	64
4.3.1	Compile-time vs Run-time	64
4.3.2	Daemon vs Agent Process	65
4.3.3	Heuristic Log	66
4.3.4	Remote Write over Remote Read	72
5	Page Migration	76
5.1	Thrashing	77
5.2	System Implementation	77
5.2.1	Basic Algorithm	79
5.2.2	Sharing Page Mapping	81
5.2.3	Pending Request Queue	88
5.2.4	Page Update History	89
5.2.5	Locating the Page	92
5.2.6	Page Table Look aside Buffer	93
5.2.7	Avoiding Deadlock	94
5.2.8	Non-Migratable Page	95

5.3	Runtime Heuristic	96
5.3.1	do_page_request	97
5.3.2	examine_page_req	97
5.4	Modified Malloc and Free	99
5.4.1	Cut and Re-link	100
5.4.2	Multi-page Block	102
5.4.3	Lazy free	103
5.4.4	Immediate <i>free</i> vs. Lazy <i>free</i>	106
6	Performance Evaluation	108
6.1	Overhead of Pointer Checking	108
6.1.1	Explicit Pointer Checking	108
6.1.2	Signal Handler	109
6.2	TreeAlloc & TreeAdd	111
6.3	Remote Read/Write	114
6.4	Bitonic Sort	117
6.4.1	Advantage of Page Migration	119
6.4.2	Distributed Sequential Code	120
6.4.3	Parallel Code	120
7	Related Research & Future Work	122
7.1	Process Migration for Locality of Access	124
7.2	Distributed Dynamic Data Structure	125
7.3	Selecting Data and Process Migration	126
8	Conclusions	129

List of Figures

1.1	Linked Data Structures	2
1.2	Loosely Coupled Processors	3
1.3	Message Passing	4
1.4	Distributed Shared Memory	5
2.1	Explicit vs. Implicit Migration	13
2.2	Migration Points	14
2.3	Creating a Linked List	16
3.1	Execution Model	21
3.2	Prototype	24
3.3	Daemon Network	25
3.4	Flow Diagram of a Daemon	26
3.5	Global Shared Heap	28
3.6	Sharing Address Space	29
3.7	<i>spm_malloc</i> keeping track of available holes	30
3.8	Compiling Agent Process	32
3.9	Injecting Agent Process	34
3.10	Migrating Agent Process	34

3.11	Overhead of process migration over NFS	36
3.12	Process Snapshot	38
3.13	Linked List on 3 daemons	41
3.14	Linked List Traversal: C Code	43
3.15	Compiled Machine Code	43
3.16	Following the remote pointer	44
3.17	Tracing Execution	46
3.18	Reusing Agent Process	48
3.19	.spm_profile	49
3.20	spmstart options	50
3.21	simple shell script	50
3.22	Agent Process: User Space and Kernel Space	53
3.23	Synchronizing within a single daemon	54
3.24	Migrating in the middle of the system semaphore call	55
3.25	Migrating before the system semaphore call	55
4.1	Steps of Remote Read	59
4.2	Steps of Remote Write	60
4.3	Waiting for the Remote read/write Reply	62
4.4	Inconsistency case 1	63
4.5	Inconsistency case 2	63
4.6	Overlapping Computation and Remote Write	64
4.7	Visualized Trace of Linked List Creation	68
4.8	Linked List Creation Example	69
4.9	Pseudo Code for migration heuristic of an agent process	70
4.10	Visualized Trace of Linked List Creation with Remote Write	71

4.11	Shifting the value	72
4.12	Opportunities for both Remote Read and Write (source code at Fig 4.11)	73
4.13	When Remote Read is used	75
4.14	When Remote Write is used	75
5.1	Accessing two separate arrays	78
5.2	Accessing left and right subtrees	78
5.3	Pseudo Code of the daemon	80
5.4	Granting a Page Migration Request	82
5.5	Rejecting a Page Migration Request	82
5.6	Initial Page Bitmaps	84
5.7	Page Mapping State Diagram	86
5.8	Daemon handling page mapping update	87
5.9	Agent Process handling page mapping update	87
5.10	Request on a page that is in a pending list	88
5.11	Request on pages in the <i>waiting_page</i> list	90
5.12	Page migrating one after another	93
5.13	System calls <i>write</i> and <i>read</i>	95
5.14	Moving a page inside the Global Shared Heap	99
5.15	Sequentially sorted linked list	101
5.16	Linked List of Linked List	101
5.17	Allocating a block larger than a page	104
5.18	Problem of Multi-page blocks	105
5.19	Example of Lazy Free	107

6.1	Code for causing Segmentation Fault	110
6.2	Tree Structure	111
6.3	TreeAlloc and TreeAdd	112
6.4	ParallelTreeAdd	113
6.5	Testing remote read/write using linked list	114
6.6	Program that can benefit from remote write	115
6.7	Program that can benefit from remote read	115
6.8	Result from remote write	116
6.9	Result from remote read	116
6.10	Result from remote read/write	117
6.11	Bitonic Sort Initial Data Mapping	118
6.12	Bitonic Sort: Sequential Code and Distributed Sequential Code (4hosts)	121
6.13	Bitonic Sort: Sequential Code and Distributed Sequential Code (4hosts) Tree Size = 2^{27} nodes	121

List of Tables

4.1	Load and Store instruction set	62
4.2	Deciding on remote read/write	65
5.1	Potential Incorrect Page Mappings	84
5.2	Overheads of <code>spm_malloc</code> and <code>spm_free</code>	102
6.1	Overhead of Explicit pointer checking. Traversing 600,000 nodes for 2000 iterations.	109
6.2	Overhead of Signal Handler: 1000 Segmentation Faults	110
6.3	TreeAdd time with TCP/IP process migration and ghost processes	113
6.4	TreeAdd time with NFS process migration and without ghost processes	114
6.5	Bitonic Sort Tree Size 2^{23} Leaf Size 2^{17} running on 2 Hosts	119
6.6	Bitonic Sort: Parallel code	120
7.1	Migration Types	125

Acknowledgments

First and foremost, I would like to express my appreciation to Professor Lubomir Bic and Professor Michael Dillencourt who have supported me in every stage of my graduate study. This dissertation would not have been possible without their continuous encouragement, patience, and numerous discussions throughout the years. Thank you so much for not giving up on me.

My thanks also to my other committee members, Professor Alex Nicolau and Professor Lichun Bao for the time and guidance they have provided. Thanks to the School of Information and Computer Science for supporting me through the Teaching Assistant Fellowship.

My overwhelming thanks to Munehiro Fukuda and Suzanne Schaefer who have been my mentors, supporters and friends. They have motivated and guided me especially when I was under stress and struggled in my research.

Many thanks to my colleagues in our MESSENGERS group, Yev (Eugene) Gendelman, Hairong Kuang, Jiming Liu, Richard Utter, Adam Chi-Lun Chang, Lei Pan, Ming Kin Lai, Javid Huseynov, Wendy Zhang, Min Wu, Olga Rivero, and Yosen Lin. In particular, Ming, I enjoyed every enlightening discussion we had at our office over the years.

I would like to thank my parents and all my family members for supporting

me and keeping touch with me from overseas. And finally, I owe special thanks to my wife, Mami, for believing in me all the way through.

Curriculum Vitae

Koji Noguchi

March 1999 B. Eng. in Information Engineering
Tohoku University, Japan

December 2001 M.S. in Information and Computer Science
University of California, Irvine

March 2007 Doctor of Philosophy in Information and Computer Science
University of California, Irvine
Dissertation: Spontaneous Process Migration with Global Pointers

Abstract of the Dissertation

Spontaneous Process Migration with Global Pointers

by

Koji Noguchi

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2007

Professor Lubomir F. Bic, Co-Chair

Professor Michael B. Dillencourt, Co-Chair

Pointers in C provide a powerful and flexible approach for designing dynamic data structures whose sizes and shapes change at run-time, e.g. linked lists, trees, stacks, hashed tables, directed acyclic graphs (DAG), general graphs, etc. But this advantage could become an obstacle when running on a loosely coupled cluster using message passing or DSM systems. Their dynamic nature, such as incremental allocation and irregular access pattern, makes it hard for the compiler or the programmer to distribute both the data and the code in a balanced fashion.

We will describe a process migration system that supports system-wide pointers and global dynamic data structures. The system lets the processes to 1) distribute the data structures among hosts, 2) implicitly migrate when

referencing remote data, 3) handle pointers in a location-transparent manner, 4) fork and synchronize through the native UNIX calls.

In addition, we tackle the granularity issues when the process migration becomes inefficient. The system is enhanced by an ability to pull or push a very small amount of data (word) or to migrate a large virtual address page(s) instead of migrating the process. Our system makes sure that the correctness of the algorithms is preserved and adds the runtime heuristic for selecting between process and data migration based on the past access pattern of the processes.

Chapter 1

Introduction

Dynamic data structures play a major role for providing flexible and adaptive programs in C. However, its use on a distributed computing environment especially for loosely coupled cluster is rather rare. Because of its intricate nature (e.g. incremental allocation and irregular access pattern), distributing and managing these structures on cluster systems have been a challenging task. The two common approaches for distributed computing, Message Passing and DSM (Distributed Shared Memory), would involve a significant change to the original sequential code and also require a runtime overhead.

We introduce our process migration system with global dynamic data structures and propose as an alternative system for coding and running pointer-intensive applications on a distributed environment.

1.1 Target Problems and Architectures

Our system is not a silver bullet for parallelizing all kinds of scientific applications. Instead, our system focuses on applications with intensive use of

dynamic data structures and provides runtime support that distributes computation and data efficiently based on the migration pattern. Linked lists (a), trees (b), directed acyclic graphs (c), general graphs (d), or any combination of these would be good examples of a dynamic data structure (Fig 1.1). Its application scope extends from computer geometry and graphics to individual based simulation and fluid dynamics.

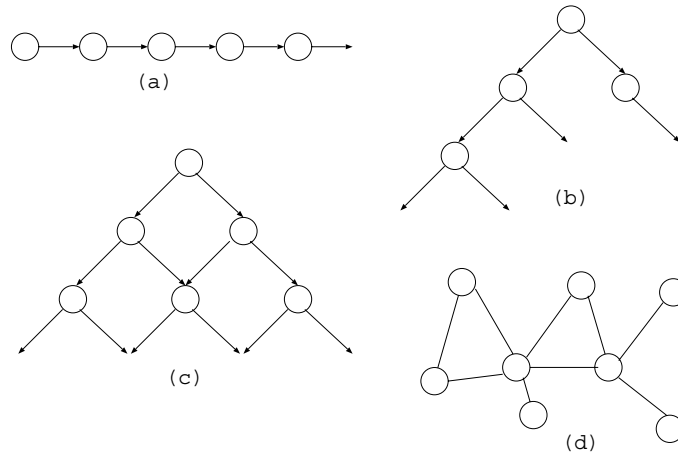


Figure 1.1: Linked Data Structures

What distinguish pointer-intensive codes from array-based codes are the means of allocation and manner of access. Arrays are allocated once at the start up and their shapes and sizes generally do not change during the rest of the execution. On the contrary, linked data structures are allocated incrementally, and their sizes and shapes can change throughout its execution. In addition, arrays are accessed regularly by loops, whereas dynamic data structures have relatively irregular access patterns.

Because of this intricate nature of pointer intensive codes, distribution of dynamic data structures cannot be determined until runtime, which makes the conventional approaches to require complex communication pattern or extra

care for data and computation mappings.

Throughout this thesis, we will be using loosely coupled processors (Fig 1.2) for our running environment. It is the common configuration in today's cluster computing. Each processor contains its own memory and the speed of remote memory access is magnitude of order slower than the local memory access.

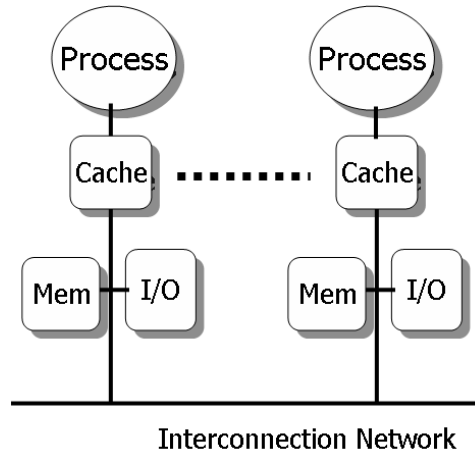


Figure 1.2: Loosely Coupled Processors

1.2 Message Passing

Message Passing (Fig 1.3) is undoubtedly the dominant approach used on distributed-memory systems. Full control over the data and the computation mapping is given to the programmers. It is the programmers' responsibility to make sure that all the data are transferred promptly to the necessary hosts. MPI [49] is used very often because of the availability and performance of its highly tuned implementations on various architectures and for several major programming languages.

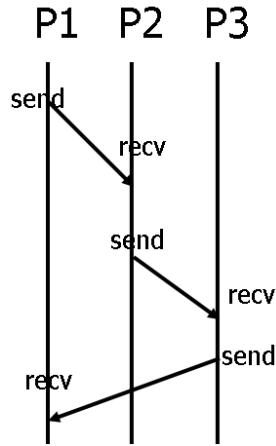


Figure 1.3: Message Passing

In order to build the dynamic data structure on top of MPI environment, programmers have to partition the data structures and place the computation in-between the communication calls. The straightforward approach would require two phases: first locally constructing a data structure at each host, and then having an explicit message to imitate the remote pointer dereference across the two hosts. However, there are several restrictions to consider. To create and delete the link without constraint, or to be able to *traverse* the link, e.g. to move the locus of computation, both the sender and the receiver have to agree on the action (two-sided communication) they are performing. Uncertainty of the data mapping prevents the receiver from promptly calling the *receive* function. It is particularly complicated when the access is conditional.

Even one-sided communication introduced in MPI 2.0 [50] is unfit for our purpose, especially not allowing concurrent conflicting one-sided get/put accesses to the same memory location [12].

RPC [9] or any similar services (RMI [51], CORBA [65]) do not have this receiver problem, but still face the difficulty of separating the computation into

source and remote functions. This itself requires the programmers to know the shapes and the sizes of the dynamic data structures at compile-time.

1.3 Distributed Shared Memory (DSM)

DSM (Fig1.4) provides an abstraction of a single address space spanning over the hosts. Originally designed for tightly couple system where the access to any part of global memory was considered equally fast, this simple design provided an advantage of ease of programming over Message Passing systems but suffered from scalability problem. Protic [58] provides a good overview of the history of DSM. In early developments of DSM, such as IVY [44], data mapping were completely transparent to the programmer and sequential consistency [43] was enforced. Later, consistency model was relaxed [25] (Midway [6], Munin [13], TreadMarks [1]) and more data allocation control was given with the compiler support (UPC [16], Titanium [69], Cilk [11]) resulting in higher performance.

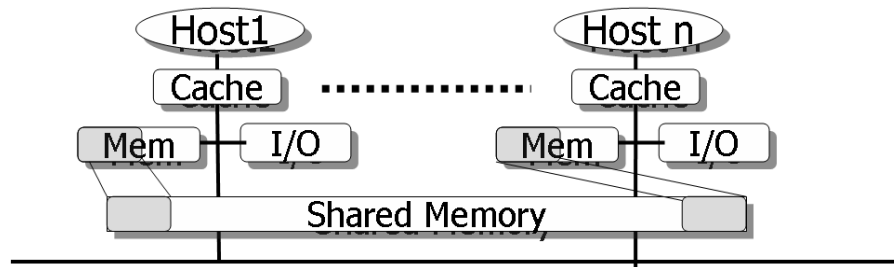


Figure 1.4: Distributed Shared Memory

There are enormous types of DSM systems varying in consistency, algorithms and architectures, but the main idea remains the same, *to pull the remote data*. This simple concept preclude its method from being used for our

application (e.g. programs traversing the dynamic data structures).

1.4 Process Migration

Unlike message passing or DSM, process migration systems [48] let the processes to move the computation instead of the data. Some of the advantages of process migration are:

Dynamic Load Balancing Runtime system can decide to move some of the processes on overloaded host to a less crowded host.

Data Access Locality Large size of remote memory access can be avoided by moving the process to the remote site and thus achieving local access to the data.

Algorithmic Integrity Less code modification is required (compared to message passing), thus preserving the original code structure [56].

Fault Tolerance In case of a system crash, checkpointed process could be used for recovery.

When applications run on shared workstations, using process migration for load balancing is an attractive feature. However, nowadays, most cluster systems provide a way to reserve and use dedicated CPU(s) in order to run their applications. In these cases, the compiler or the programmer may be able to balance the data and computation at compile time thus not requiring process migration for load balancing and access locality. It is easy to imagine that the amount of overhead associated with process migration is much larger

than sending a single message in message passing systems. Thus it is natural for people to ask

“Why process migration?”

Perhaps a better question to ask is

“When does the benefit outweigh the cost of using process migration?”

We believe that the answer to this question is “Uncertainty at compile time”. This “uncertainty” could be

- Amount of computation
- Shape and/or size of the data structure
- Load of the network

If any of the information is missing, it becomes extremely difficult for the compiler or the user to equally distribute data and computation. This is when the process migration’s ability to migrate becomes handy. The process can adapt to the new finding at runtime and change its behavior accordingly.

These “uncertainties” may arise in the following occasions.

Applications using Dynamic Data Structures As we have already mentioned in the previous sections, incremental allocation and irregular access pattern prevent us from predicting the correct shape and size of the data structures at compile time.

Indirect Arrays Access, e.g. $A[B[i]]$ Unless contents of array B is somehow known at compile time, the compiler cannot determine where to

place the computations. Inspector-executor technique [62] is a common approach to this problem and our system could be used to adapt to the data access pattern at runtime.

Application running on GRID [22] environment In GRID, speed of the CPUs and network delays among hosts are unknown and unstable since GRID makes use of the available resources on the network. For this reason, programmers might prefer distributing the data at runtime, consequently requiring process migration for placing the codes.

Throughout this thesis, we are focusing on applications with **dynamic data structures**, but it does not preclude our system from being applied to other problems. They are left for our future research.

1.5 Research Objective

The goal of our research is to provide

- Program design suitable for distributed computing using dynamic data structures (chapter 2). Distributing the data, following the data, handling the pointers, creating and synchronizing the processes.
- Runtime support enabling implicit process migration and parallelism without requiring major change to the original sequential code (chapter 3). Support the program design but also keeping the uniprocessor performance of the sequential code.
- Enhancement by runtime selection of process migration, remote read/write (chapter 4) and virtual address page migration (chapter5). Preserving

the correctness of the code when moving the page.

The rest of the thesis is organized as follows: performance data (chapter 6), related and future research (chapter 7) and conclusion (chapter 8).

Chapter 2

Language Construct for Distributed Dynamic Data Structures

In the previous chapter, we have discussed how the process migration can be extended to programming dynamic data structures over multiple hosts. In this chapter, we will look at the characteristics of our process migration system and examine how the global pointers and dynamic data structures can be integrated into the whole system.

The goal of our system is to provide a computing process the ability to allocate and utilize dynamic data structures across distributed machines by process migrations and task parallelism. The main features can be grouped into two categories from the programmers' point of view: pointer handling and parallelism.

- Pointer Handling

- Autonomous Implicit Strong Migration
- Location-aware Pointer Allocation
- Location-transparent Pointer Handling
- Parallelism
 - Process Creation
 - Communication and Synchronization through Global Shared Heap

2.1 Autonomous Implicit Strong Migration

When dealing with process migration, we need to answer the three basic questions.

Who decides whether a process migrates?

When is the migration point decided? At compile time or at run time?

Where in the code can the process migrate?

In this section, we will look at each point.

2.1.1 Autonomous vs. Passive Migration

Depending on the goals, the decision on process migration may be made by the process itself or by the outside observer. For dynamic load balancing and fault tolerance, a central process manager or an administrator can relocate the processes to different hosts. On the other hand, if the main objective is for locality of access, each process automatically makes its decision.

In our system, we take a hybrid approach that each process makes the decision for migration and the daemons will decide to grant the migration request or to bring the data back to where the process resides.

2.1.2 Explicit vs. Implicit Hop

If the programmer knows exactly when to hop inside the code, *explicit* HOP statements can be inserted at compile time. Based on this information, a compiler can analyze and insert packing and unpacking code to achieve efficient and heterogeneous migration [64]. However, it is not always possible to determine the migration point inside the code. Unlike array based code, granularity of the dynamic data structures is much finer. Basically, any of the pointer dereference could be a potential migration point and inserting explicit hops for each data access is not realistic (Fig 2.1). In this case, we can let the system handle the mappings and let the process migrate *implicitly* following the place of the data.

2.1.3 Weak vs. Strong Migration

Weak migration carries only the process' data state. Its execution state has to be manually restored at the remote site. In most cases, it can only be migrated at predetermined points (e.g. at beginning or end of functions or blocks).

On the other hand, strong migration carries both its data state and execution state for transparent migration [26]. It does not require programmer intervention for restoring execution state so migration usually can happen between in any statements. Since we want implicit migration for remote pointer dereferences, our system has to be able to handle migrating the process any-

```
1 int foo() {
2     int a;
3     a = 2;
4     HOP (hostid2); /migration point/
5     a = a + 1;
6     printf...
7 }
```

Explicit Hop

```
1 int foo(node_t *head) {
2     node_t * ptr;
3     ptr = head;
4     while( ptr != NULL ) {
5         ptr->val = 3 * ptr->val; /migration point/
6         ptr = ptr->next;
7     }
8 }
```

Implicit Hop

Figure 2.1: Explicit vs. Implicit Migration

where in the code with strong migration. This, of course, includes migrating in the middle of the expression (Fig 2.2).

```
1 int foo() {  
2     int a;  
3     HOP;  
4     a = 2;  
5     a = a + 1;  
6  
7 }
```

Top of the function

```
1 int foo() {  
2     int a;  
3     a = 2;  
4     a = a + 1;  
5     HOP;  
6     a = a + 1;  
7 }
```

Any statements

```
1 int foo() {  
2     int a;  
3     a = 2;  
4     a = a + 1 + HOP + 3;  
5  
6  
7 }
```

Anywhere

Figure 2.2: Migration Points

2.2 Location-aware Pointer Allocation

Distribution of the dynamic data structures is the essential part of the entire application. Not only does it control the amount of hops, it also regulates the potential parallelism. Our system relies on the programmers to tell the system where to initially allocate the space through a function call:

```
spm_malloc(daemon_id, size);
```

Afterwards, our enhancement enables the data to be moved to nodes with higher affinity (chapters 4 and 5). In addition to the size of the allocation, `spm_malloc` takes the `daemon_id` as an extra argument.

Fig 2.3 shows the source code for creating a linked list such that the list elements are grouped in groups of ten and each group is allocated to a different machine in a round-robin manner. The only difference from the sequential C code is that the `malloc` call takes two arguments instead of one. Note that in addition to explicit hop by the `malloc` call, there are implicit hops at line 16.

2.3 Location-transparent Pointer Handling

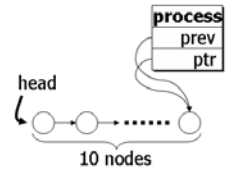
For pointer-intensive applications, speed of pointer dereference is the key to overall execution time. In order to obtain the equivalent speed as the pure sequential code, extra software-level checking for each pointer dereference would become the huge obstacle.

Many systems, such as Cid [53], Earth [30], Olden [59], MCRL [35], provide ways to annotate and differentiate between local pointers which point only to local data, and global pointers which can point to both global and local data.

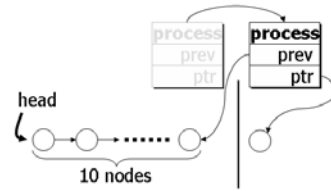
```

1 node_t * head;
2 struct node_t{
3     int val;
4     struct node_t* next;
5 }
6 int create_linkedlist() {
7     node_t *ptr, *prev;
8     head = malloc(0, sizeof(node_t));
9     head->val = 0;
10    prev = head;
11    for( i = 1; i < MAX; i++ ) {
12        ptr=spm_malloc(
13            (i/10) % num_daemons,
14            sizeof(node_t));
15        ptr->val = i;
16        prev->next = ptr;
17        prev = ptr;
18    }
19    ptr->next = 0;
20 }
21

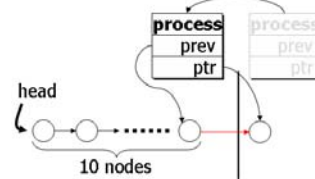
```



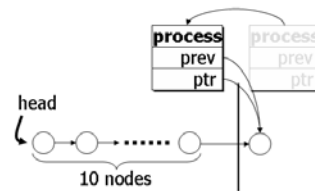
Line 19 (i=9)



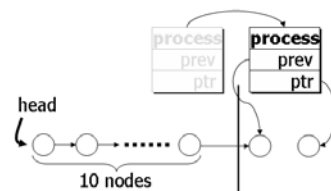
Line 12 (i=10)



Line 16 (i=10)



Line 17 (i=10)



Line 12 (i=11)

Figure 2.3: Creating a Linked List

In some cases global pointers could be inferred to point to local data by a compile analysis [45] [59]. However, there are the following drawbacks:

- Additional burden on the programmers. For each pointer declaration, the programmer has to examine whether it may ever point to a global area which is not straightforward.
- Not applicable to some data structures. For example, if a linked list is distributed, the pointer has to be declared global.
- Prohibits data migration. No data pointed to by a local pointer can be moved to remote nodes (chapters 4 and 5).

In our system, we decided not to take this approach but treat both local and global pointers as general pointers. We rely on the hardware virtual address checking mechanism to avoid the extra overhead of pointer dereference (chapter 3). With this option, the users do not need to worry about different types of pointers; and our system can freely move the data without restriction.

2.4 Process Creation

To exploit parallelism in application programs, processes have to be able to create computing entities. Since all the computations are done around the dynamic data structures, data parallelism [34] may seem to be the natural choice. It partitions the program based on the owner-computes rule and optimizes on communication. This is tempting but actually very hard to adapt to our target applications. Our main problem, not knowing the shapes and sizes of the dynamic data structures until runtime, prohibits us from using the

owner-computes rule at compile time. Instead, we chose task parallelism using the fork-and-join model. By having the processes migrate to data (chapter 3), we achieve locality of access (owner computes) and by having the data with high affinity to migrate (chapters 4 and 5), we achieve better communication.

This model also fits our system since each computation entity is a native UNIX process in our implementation and the features from native *fork* and *exec* calls can be used. Some systems only allow the creation of new computation entities at the top of the function but we will not have this limitation thanks to these system calls. Just like UNIX multi-process programming on a single host, the processes can duplicate itself anywhere in the code.

For example, if the process is traversing a tree, it is quite easy to duplicate itself so that each process could cover the given branch.

```
1  if( fork() != 0 )
2      cur = cur->right;
3  else
4      cur = cur->left;
5      ...
```

2.5 Communication and Synchronization through Global Shared Heap

Once we have multiple processes on the system, all the processes are managed and scheduled by the operating system. Just like UNIX shared memory provides shared space for processes on a single node, our system provides a global shared space spanning over all the hosts (section 3.2). Any processes can communicate through this shared area transparently. Processes will im-

PLICITLY migrate to the data or the data will implicitly migrate to where the process resides.

To resolve race condition, we rely on the native UNIX semaphores. The semaphore will be allocated on this global shared space and it allows the processes to perform computation atomically by blocking critical sections. It is also relatively straightforward to implement multiple readers and multiple writers [19] through these semaphores.

Chapter 3

Spontaneous Process Migration

In this chapter, we will introduce the details of our system that supports the programming model covered in chapter 2. Once the data is explicitly distributed among hosts, the process will automatically migrate to the data to achieve locality of access. It provides transparent handling of pointers to the application programs. Other forms of migration are covered in chapters 4 and 5.

As illustrated in figure 3.1, there are three basic entities that constitute our Spontaneous Process Migration system: *daemon*, *global shared heap*, and *agent process*.

Daemon is an entity that maps the agent processes to a physical host. The main purpose of the daemon is to isolate the application programs from the details of the physical network topology. It maintains fully connected TCP/IP connections with other daemons. The application program only sees the daemons but not the physical hosts. Any kind of agent migration requests go through a daemon and the daemon forwards the request to

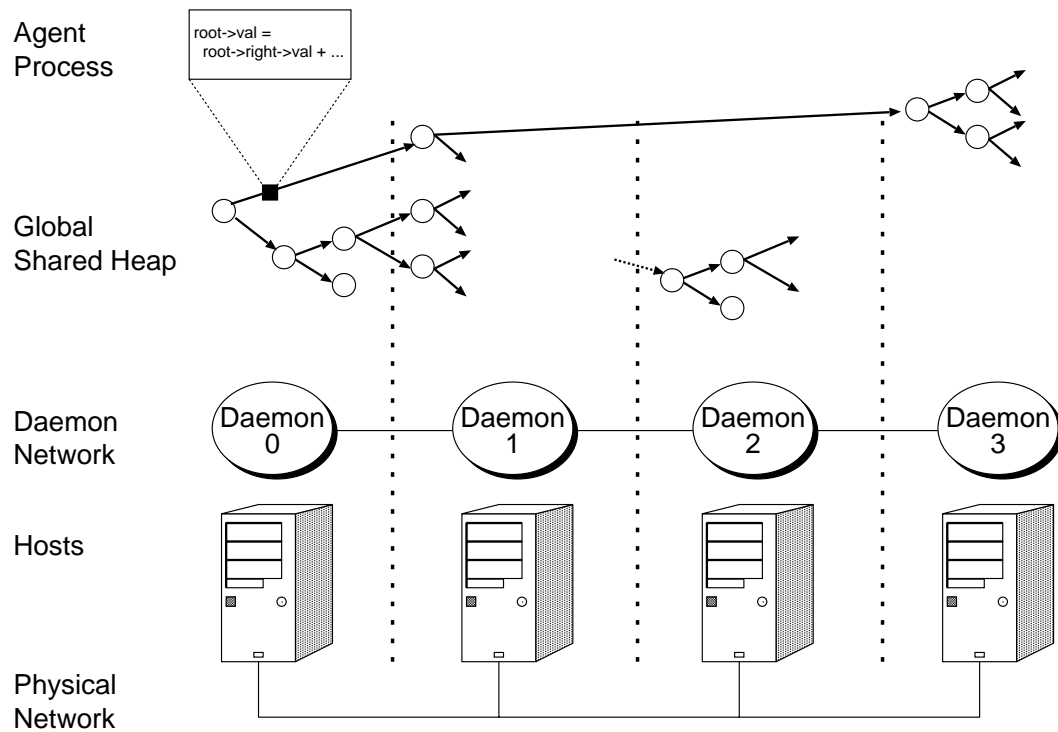


Figure 3.1: Execution Model

the correct destination. Usually there is a one-to-one matching between a daemon and a physical host.

Global Shared Heap is the only media that agent processes use to share data. (No message passing.) It is evenly distributed among all daemons and a process has to (implicitly or explicitly) hop to the specific daemon to access its content. All the agent processes share the single view and dynamic data structures are allocated on this space. Any process can create a space on a global shared heap through the *spm_malloc* library call.

Agent Process is a computing entity programmed and compiled by the user. It is a UNIX process linked with our SPM library that will let the process access the global shared heap and exchange information with the local daemons. At any given time, each agent process is owned by a daemon. Processes will run on a physical host where the daemon exist.

In this chapter, we will first describe the implementation of these three basic components followed by a process migration example that shows how these components interact with each other. We will then show some enhancements and tools made for our system and also discuss the requirement and limitation for using our methods.

3.1 Daemon

Daemon is the central component of our system that

1. starts agent processes,

2. manages process communications, and
3. directs process and data migrations.

The most complicated part, directing process and data migrations, is mainly covered in chapter 5. Here, we concentrate on only the first two tasks. In the presence of the ability to migrate a process, the question as to how to share the work between the daemon and the agent process [67] arises. Daemon can cover all the work necessary to set up the agent's runtime environment and let the agent concentrate on the computation. On the other hand, an agent can cover as much work as possible and leave the daemon as merely a simple communication layer. In our system, daemons and agent processes are separate UNIX processes. This means that a daemon has no control over agent process' data or execution. For this reason, we decided to take the latter approach where agent processes execute most of the migration steps.

Initially, our prototype did not even have a daemon as illustrated in figure 3.2. Our agent process had all the capability of packing and unpacking (section 3.3) and depended upon the ssh server to create a new agent on a remote host. The main disadvantage of this design was the huge overhead associated with this ssh connection.

To obtain a migration speedup, we replaced the ssh server with our own daemon. Figure 3.3 shows how the daemons interact with each other and talk with the local agent processes. At start up, daemons establish a fully connected TCP/IP network followed by a FIFO queue for receiving messages from local agent processes and a user. Although the initial design of the daemon was quite simple, it grew as features such as remote read/write (chapter 4) and

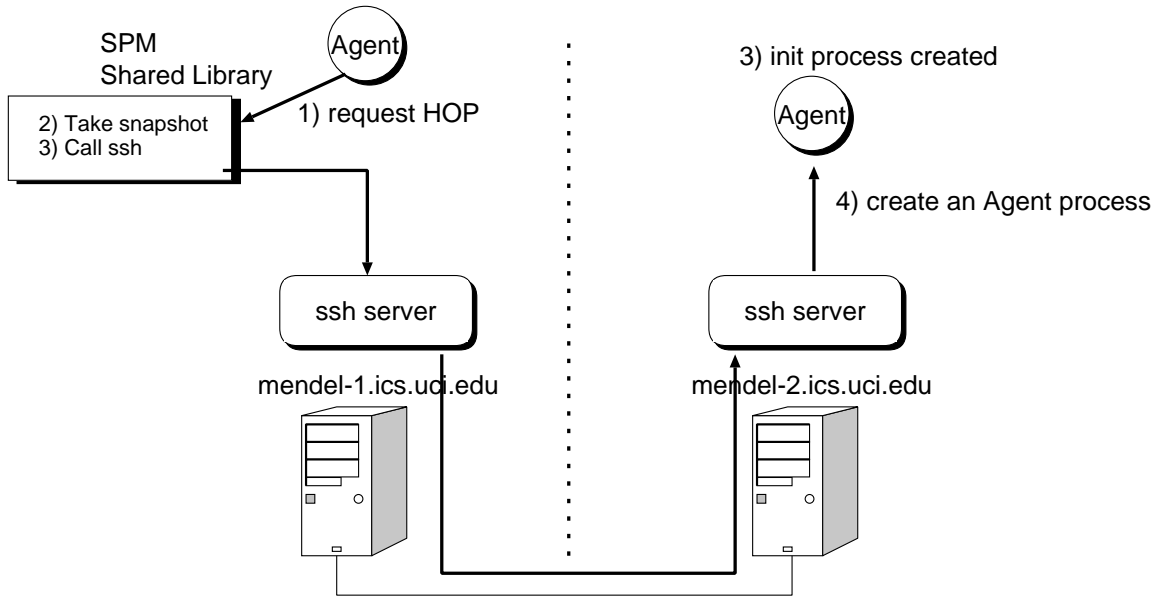


Figure 3.2: Prototype

page migration (chapter 5) were added to the overall system. In this section, we will only cover the basic element. Please refer to the corresponding chapters for more detail.

Daemon is an iterative server that listens to a FIFO and all the sockets to handle requests from the following.

Local Agent Processes Whenever agent process decides to migrate, it asks the daemon to forward its request and create a new agent process at the remote site.

Message from other Daemons Receives the request from remote agent process, fork a new process and *exec* the requested program.

Request from the user (Master Daemon Only) User may inject a program into the system at any time. For reason covered in section 3.5.5, it is restricted to Master daemon only.

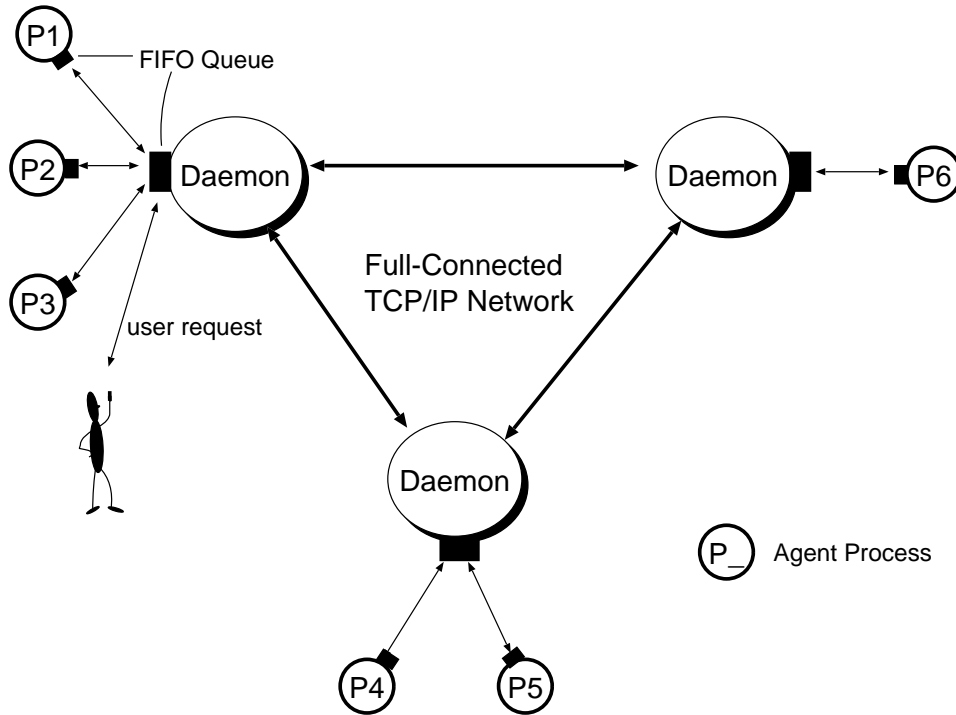


Figure 3.3: Daemon Network

Figure 3.4 uses a flow diagram to show how the daemon interacts. Whenever a daemon has no data coming from other daemons or agent processes, it will simply sleep to avoid wasting CPU power.

All the computation and communication intensive work are conducted by each agent process (section 3.3). Information exchanged among daemons and processes, called **pmi** (*process migration information*) by us, are all less than 100 bytes. This allows us to simplify the daemon design by using an iterative server instead of a concurrent server. It will have less overhead per request, and enables the daemon to handle complex I/O (by no race conditions).

Several local agent processes can contact the daemon at the same time. Atomicity of each request is guaranteed through a UNIX FIFO channel.

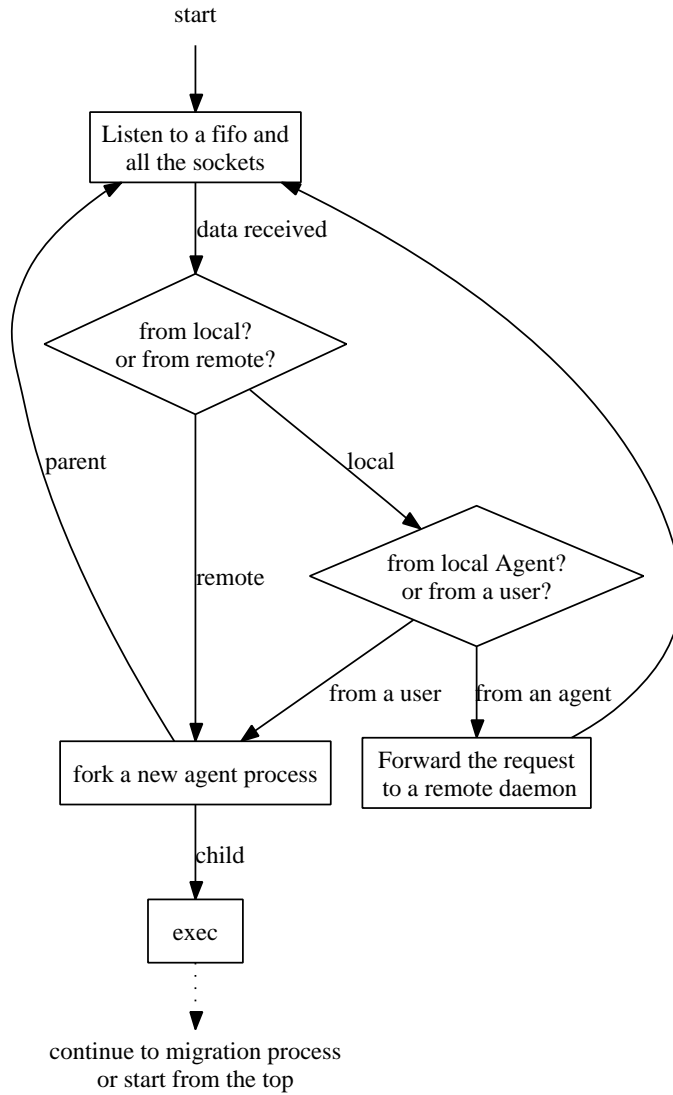


Figure 3.4: Flow Diagram of a Daemon

Since each agent process is a native UNIX process, the daemon does not need to keep track or control the processes once they are injected. As covered in section 2.5, it is up to the application programmer to correctly synchronize the agent processes for protecting the critical sections using the UNIX system calls.

3.2 Global Shared Heap

Global Shared Heap is the space distributed over the hosts and is used to store the dynamic data structures for the agent processes to share. An abstract view of a single shared heap space to the application program is provided without requiring software-level pointer checking. We can divide the work of the global shared heap into 3 key features.

- How to create and extend shared space
- How to create dynamic data structures on the space
- How to detect remote data access

3.2.1 Sharing Memory Space among Processes

Basically, what we want is a space that all the processes can read from and write to from any hosts. If we limit this focus to a single host, then there is already a solution provided: UNIX shared memory (*shm_open etc*). By using the shm system calls, we now extend the space from a single host to multiple hosts distributed over the networks.

First, each process will reserve the virtual address space for the entire global heap between the stack and the heap area by the *mmap* call (Fig 3.5). This

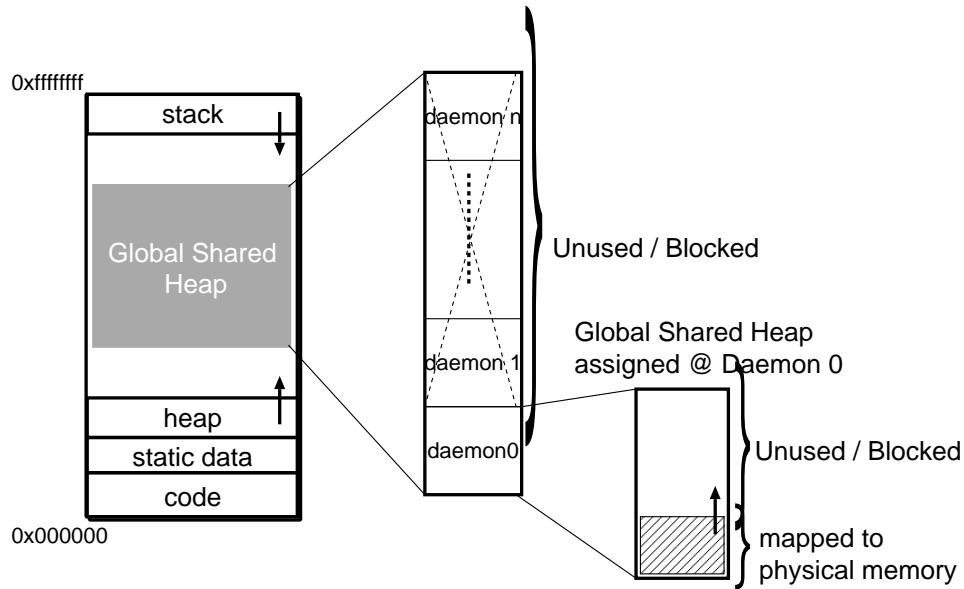


Figure 3.5: Global Shared Heap

reserved space is split into blocks and each block is assigned to a daemon. Note that each daemon only maps the virtual memory to physical memory only for the memory block being assigned. The remaining space is not mapped to any physical memory, thus no memory is wasted.

When an agent process is injected, the shared page mappings are carried out inside the initialization function so that all the agent processes on the local daemon share the same page mapping (Fig 3.6). When an agent migrate from one daemon to another, mapping of the global shared heap changes before resuming execution at *spm_init_process* function call (section 3.3.1). Thus, used with automatic process migration on remote data access (sections 3.2.3 and 3.3), processes can access this global shared space transparently without any special function calls.

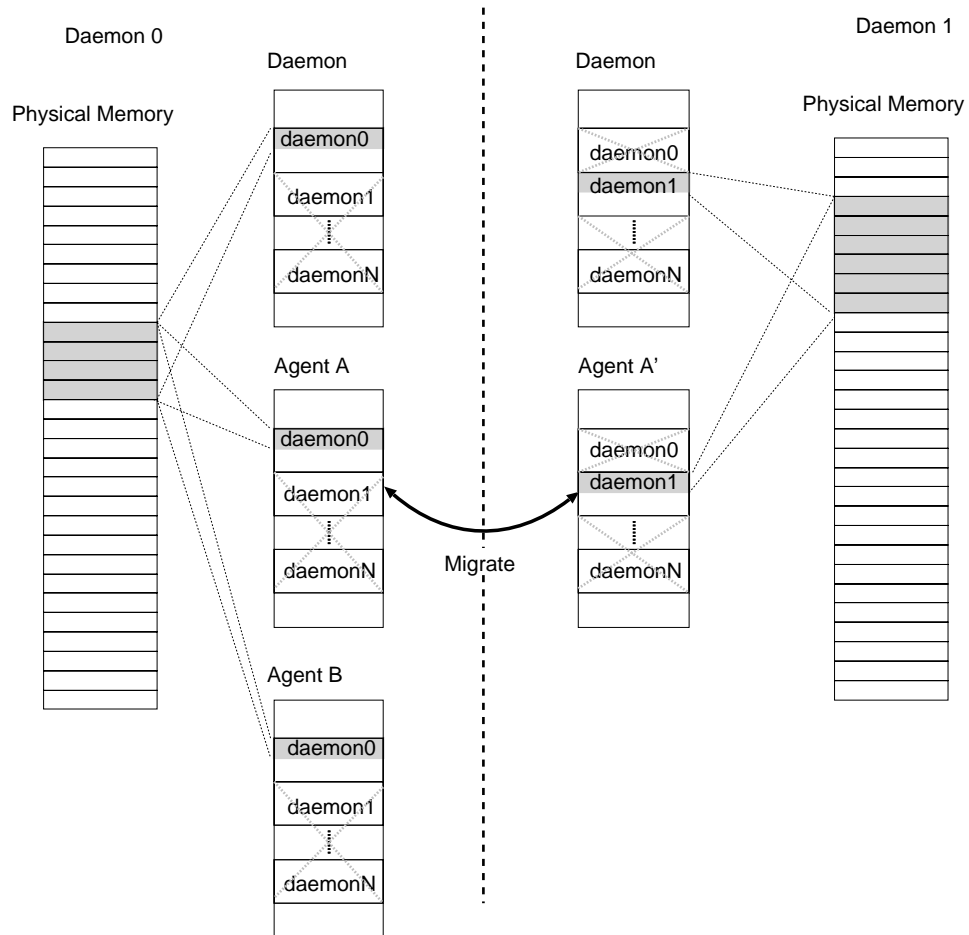


Figure 3.6: Sharing Address Space

3.2.2 `spm_malloc` and `spm_free`

So far, we only have a single flat shared space for the agent processes to play with, but our goal is to place dynamic data structures which have much finer granularity. In C, people create dynamic data structures through *malloc* system calls. We provide our own *spm_malloc* function which acts just like UNIX *malloc* that returns a pointer to an allocated memory. The difference is that, in addition to the requested size, it asks for the daemon number in order to specify which daemon block to allocate from. Initially, each block starts with a few pages and the *spm_malloc* incrementally maps more physical memory pages as needed.

For our allocation method, we employed a linked list implementation from *Operating Systems Principles* [7] chapter 7 with First-Fit. On each daemon, all the available holes are followed by doubly-linked-lists and free block is merged with neighboring holes (Fig 3.7). Holes do not need to be sorted by its physical address.

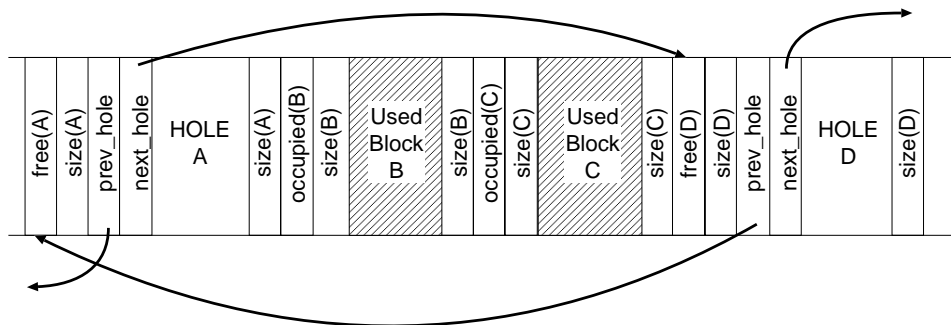


Figure 3.7: *spm_malloc* keeping track of available holes

An agent process will migrate to a requested daemon before searching for

an available hole. Note that actual allocation is not done by the daemon but by the individual agent process. If concurrent *spm_malloc* or/and *spm_free* calls are expected, our system would require synchronization just like multi-threading introduces certain synchronization overhead for malloc and free. If a user or a compiler [60] can guarantee that there would be no race condition, they have a choice of using the lock-free *spm_malloc* and *spm_free*.

In order to allow page migration, this design is extended in section 5.4.

3.2.3 Remote Pointer Dereference

Our target applications involve a great deal of time in traversing the (distributed) dynamic data structures. If each pointer dereference is being explicitly checked by the software, it will slow down the entire code significantly (section 6.1.1). To avoid this situation, we took an alternative solution by using the UNIX hardware virtual address locking mechanism (*mprotect*). Since the hardware-level checking is done for regular C pointer dereference, we are not adding any overhead as long as the agent processes access the local objects. When a pointer pointing to a blocked virtual page is dereferenced, SIGSEGV (segmentation fault signal) is thrown to our signal handler (registered in the process initialization function). It will then handle the process migration as described in the next section.

3.3 Agent Process

Agent process is the main component which executes the computation programmed by the user. The agent process mainly consists of two parts: C

program written by the user and the SPM (Spontaneous Process Migration) shared library linked at compile time as illustrated in figure 3.8. As long as the computation is accessing local data, it runs just like the pure C code without any overhead.

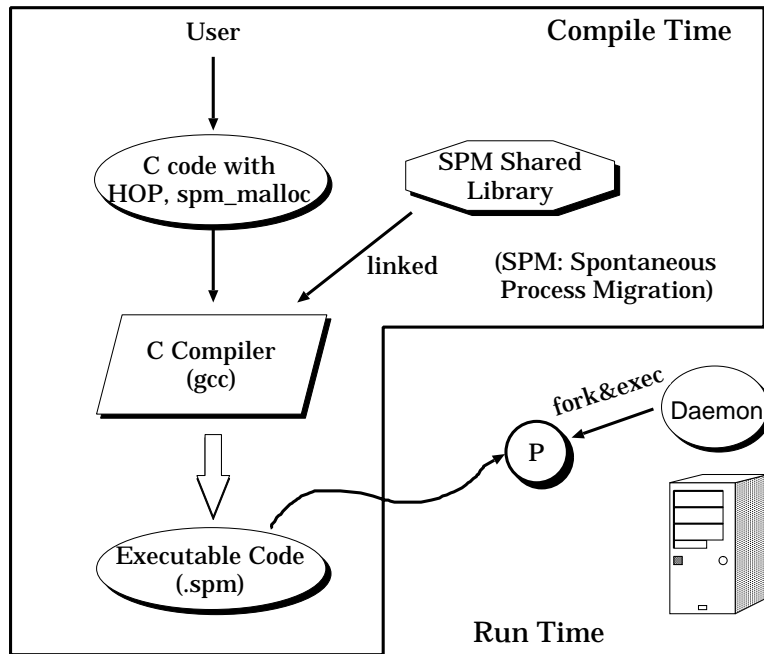


Figure 3.8: Compiling Agent Process

Once the remote pointer dereference is detected (section 3.2.3), it will start the migration process. To prepare for migration, process has to be able to take a snapshot of itself and restore its state at the remote host. It is a OS-dependent implementation requiring detailed care. Here, we will briefly go over the basic snapshot mechanism we used on the Solaris system.

3.3.1 Agent Startup

Agent process is always started by the local daemon. Depending on the arguments passed to the agent program, the agent process will know if it should

start the program from the top or to restore the state from the remote agent process and resume execution. For this selection process, the user is asked to change the original sequential code so that *spm_init_process*, process initialization function, is called at the top of the main function. Figures 3.9 and 3.10 show the steps for agent process startup and migration.

Injecting Agent Process A user or an agent process sends a request to a local daemon to create a new agent process. A local daemon then forks itself, and the child process starts a program by an *exec* system call. It also passes “INIT” as an argument to indicate that process should start from the top (main function).

Migrating Agent Process An agent process sends a request to a local daemon for process migration and it writes its snapshot to the communication channel. Local daemon will forward its request to the target daemon. Target daemon will then fork and exec the program with arguments that tell exactly where the communication channel is to receive the snapshot. This communication channel could be in the forms of NFS or TCP/IP (section 3.3.2).

3.3.2 State Transfer

As shown in figure 3.10, agent process uses direct communication channel for state transfer so that daemon-to-daemon TCP/IP connection is open for other requests.

In the initial design, we relied on NFS (Network File System) for the state transfer. The source agent process writes its snapshot to one single file and

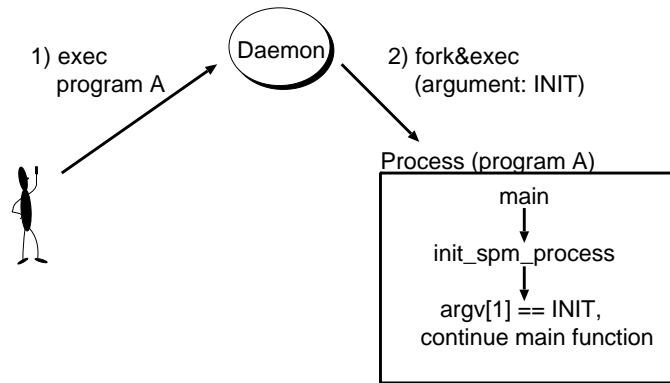


Figure 3.9: Injecting Agent Process

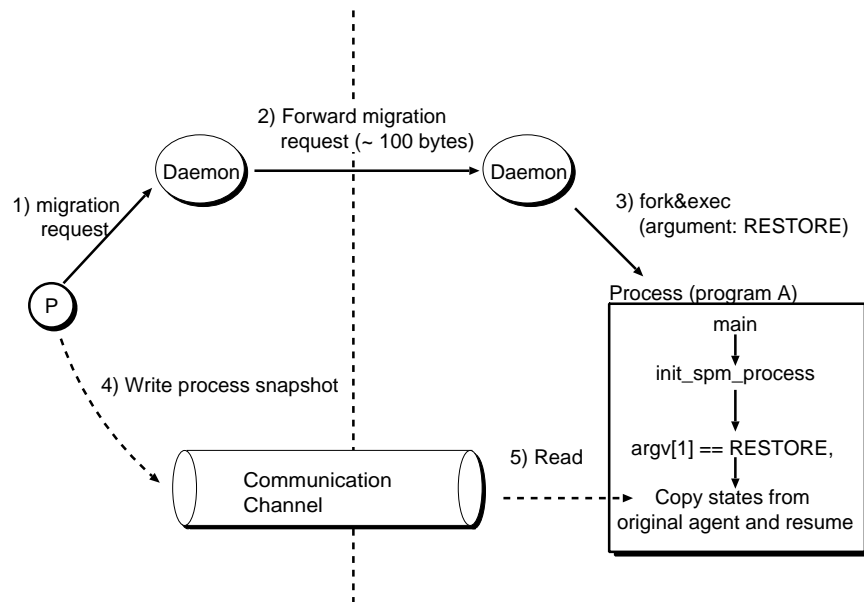


Figure 3.10: Migrating Agent Process

forwards the filename through the daemon network. A remote agent process receives the filename through argument passing and it simply overwrites its states from that file. This choice had the benefit of a simple design with some drawbacks.

Overhead of writing and reading from a shared file Besides on overhead of *opening* and *closing* a file, each *read* or *write* call generates traffic between the host and the NFS server.

NFS server can become the bottleneck As the number of active agent processes increases, the performance of each migration declines. This becomes critical seeing that our typical applications contain thousands of migrations.

***write* and *read* cannot be overlapped easily** If the *read* reaches end-of-file before all the *writes* are done, it needs a protocol to wait and re-read. Figure 3.11 shows our initial design where an agent process inefficiently writes its entire state to a file before sending the request.

To achieve better overall transfer speed, we replaced NFS with TCP/IP. For each migration, two agent processes create a new TCP/IP connection. The source agent process forwards the port number which it waits on. The target agent process receives the address(IP address and port number) and initiate TCP/IP connection. Once the connection is established, we transfer the entire process state. This would eliminate the overhead of going through the NFS server. Actual performance result is covered in section 6.2. Note that we still have some overhead of 3 way handshake for TCP/IP connection set up and closing but it is much less compared to the overhead of NFS transfer.

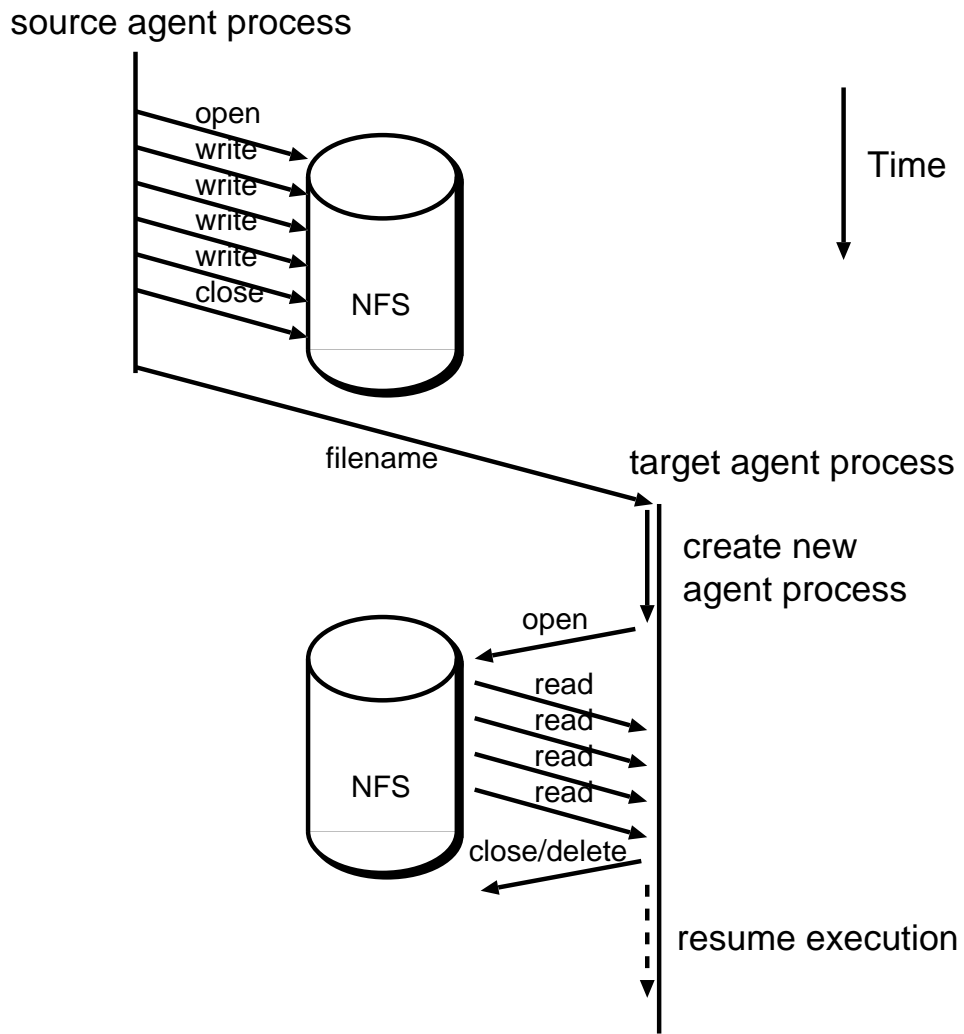


Figure 3.11: Overhead of process migration over NFS

3.3.3 State Snapshot

Roughly speaking, a process consists of stack, heap, static data, code segment and CPU registers. In order to take a snapshot of the entire process state, we need to know exactly where each state begins and ends (Fig 3.12). *_end* and *_edata* are global variables set by the compiler and *sbrk()* is a system call. Those shaded areas in the figure, stack, heap, part of the static data area and CPU state, are transferred for migration. Code segment, temporary stack and private data area are not transferred. Some of the fields might require some explanation.

CPU State Contents of the registers. Copied through the signal handler.

Code Segment Executable code. It is automatically read from NFS when the process is first executed by the daemon.

Temporary Stack Part of the SPM library and reserved especially for copying over the process state. We cannot use the regular stack since process migration involves copying the stack itself.

Private Data Area This is reserved for information relevant only to the actual process (e.g. process id, port number, fifo file descriptor, etc). When ghost process (section 3.5.2) is used, this space is used to save the previous states of the process and avoid redundant system calls.

We have adopted a user-level checkpointing similar to the Condor system and refer the readers to the Condor technical report [46] for comprehensive description of the state snapshot mechanism.

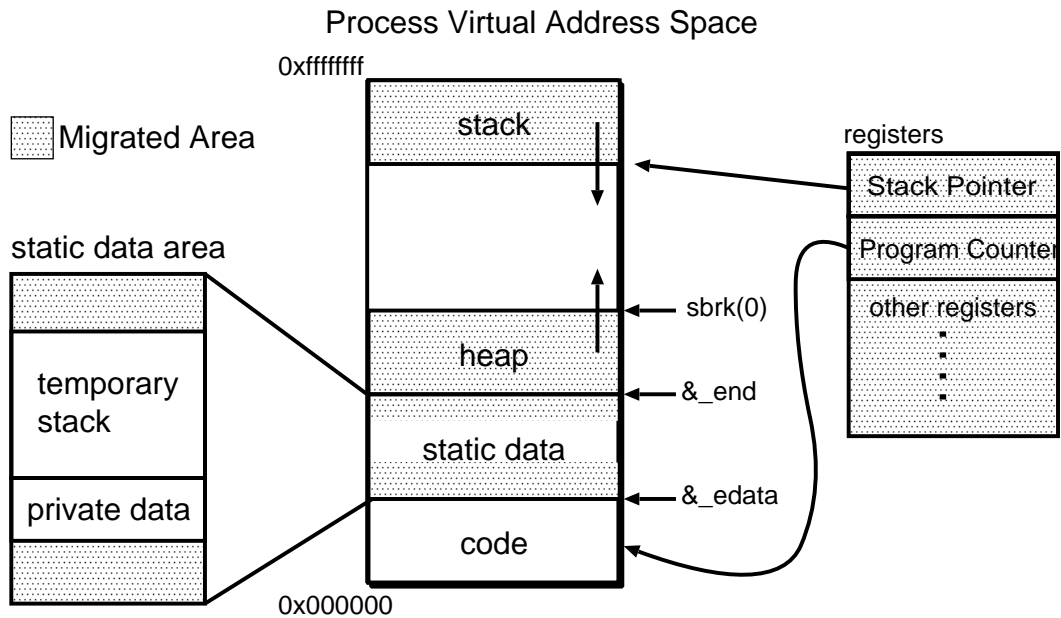


Figure 3.12: Process Snapshot

3.3.4 Signal Handler

One of our goals is to have as little overhead as possible when an agent process is running on a single machine. To avoid the extra cost, we rely on a signal handler to handle all the necessary work for the migration. Basically, the agent process runs just like a sequential code until there is a remote pointer dereference and the signal is being caught. Agent process may also choose to explicitly hop by sending a `SIG_USR1` signal to itself to activate the same signal handler. The signal handler takes 3 argument which are filled by the operating system when the signal is caught.

```
void (*signal_handler) (int sig, siginfo_t * sip, ucontext_t *ucp)
```

sig signal number

sip siginfo structure tells extra information on why the signal is generated.

`sig->si_addr` holds the address of a faulting memory reference.

ucp `ucontext` structure defines the context of a control within an executing process. It holds machine registers and all the implementation specific data when the signal event occurred. By looking at the Program Counter, we can also learn the code address when the SIGSEGV is raised.

```
(ucp->uc_mcontext.gregs[REG_PC])
```

At this point, we have enough information to take the full snapshot of the agent process itself and migrate (Fig 3.10).

3.3.5 Locating a Remote Memory

When signal handler receives the faulting memory address, it has to be able to know where that remote address exists. Since we divide the entire global heap space equally to each daemon (*reserved_size*), it is simple arithmetic.

```
remote_id = (faulting_memory_addr - top_of_global_heap) / reserved_size;
```

When page migration is introduced (chapter 5), this equation would no longer hold. It would then require a separate data structure to keep track of where the pages have moved.

3.4 Migration Example

In sections 3.1, 3.2, and 3.3, we have discussed how individual components interact as part of a system. In this section, we will see how the system works as a whole by using a simple linked-list example.

Figure 2.3 shows an agent process' code which creates a linked list of *MAX* nodes: bridging daemons for every 10 nodes. Note that the only difference from C sequential code is the two *spm_malloc* calls. This program, linked with our *spm* shared library, will distribute the linked list over multiple daemons (hosts). It involves two kinds of hopping: explicit and implicit. At line 12-14, *spm_malloc* call would migrate the process when the specified daemon number differs from its place. After this migration, *prev* pointer would be pointing to a node allocated at previous daemon, meaning it is pointing to a blocked page. Thus, at line 16, SIGSEGV signal is thrown and signal handler will be activated. Within the handler, it would migrate back to the previous daemon and assign the variable to the correct place. Immediately after that, *spm_malloc* call would explicitly migrate the process again. This is repeated for every 10 nodes.

Variable *num_daemons* (at line 13) is a global variable set inside the *spm_init_process* call so that a single program could be used to run on various number of daemons. Figure 3.13 shows how the linked list would appear when the agent process is injected to a three daemon-network.

We have used the C source code example, but at runtime, migration break point is determined at machine code level. The next example gives C code that traverses the linked list created above (Fig 3.14) and also the compiled machine code (Fig 3.15). Optimization option is turned off for ease of reading. Note that there is no special code added to the C code. Once the data is distributed, an agent process implicitly migrates to the location of the data to obtain the local data access. In our example, an agent process will migrate at line 6 (code address 0x11ff0) where it loads up the content of *ptr->val*. When

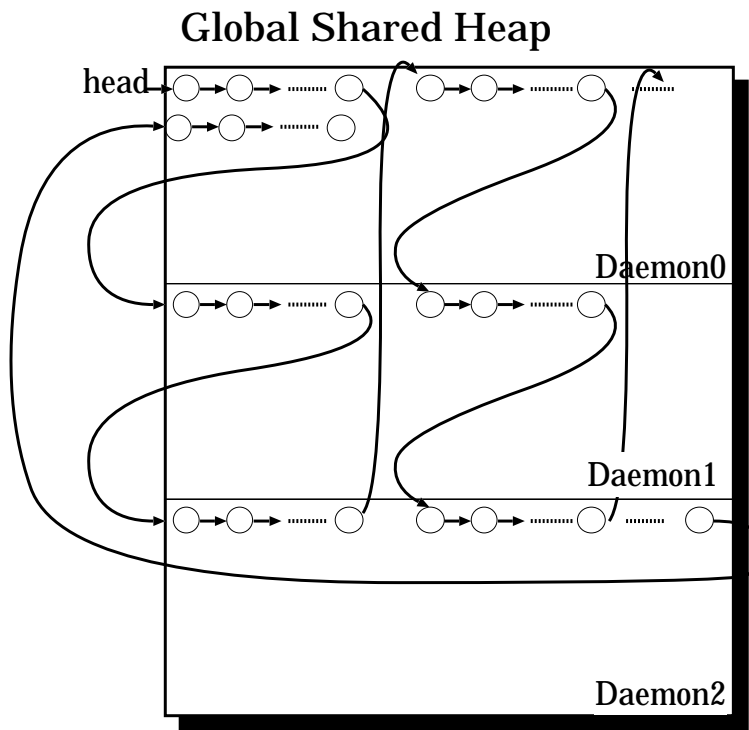


Figure 3.13: Linked List on 3 daemons

ptr is pointing to a blocked page, ‘ld’ command would fail with SIGSEGV. Now, once the migration succeeds, the agent process will repeat the same code (0x11ff0), but this time it will not stop since the *ptr* is now pointing to a valid page. Figure 3.16 shows how the agent process circulates the daemons until it hits the end of the linked list. Whenever it ask for the content of a non-shaded (remote) area, it will initiate a migration.

3.5 Miscellaneous Enhancements

The two major enhancements, remote read/write and virtual page migration, which occasionally migrate small data or virtual pages instead of an agent process, are covered in the next two chapters. In this section, we focus on minor features and tools which contributed to the speed or the usefulness of our SPM system.

3.5.1 Tracking Process/Data Movement

Debugging message passing code is more challenging than debugging a sequential code. Now, our system allows the process, a single data element (remote read/write), and virtual pages to migrate from host to host. This decision is made by our system at runtime and it provides a transparency to the application programmer with regards to the type of migration. However, some users might want to know or worse have to debug/trace their migrating programs. To help such programmers, we have implemented a trace log feature that visualizes the migration, forking, and computation time with the help from the graphviz library [24]. Figure 3.17 shows an example of the trace. Each agent

```

1 int add_all_nodes (node_t *head) {
2     int total;
3     node_t *ptr;
4     total = 0;
5     for( ptr = head; ptr != NULL; ptr = ptr->next ) {
6         total += ptr->val;
7     }
8     return total;
9 }

```

Figure 3.14: Linked List Traversal: C Code

```

11fc4 <add_all_nodes>:
11fc4: 9d e3 bf 88   save %sp, -120, %sp
11fc8: f0 27 a0 44   st  %i0, [ %fp + 0x44 ]
11fcc: c0 27 bf ec   clr  [ %fp + -20 ]           //total = 0
11fd0: c2 07 a0 44   ld  [ %fp + 0x44 ], %g1     // ptr = head
11fd4: c2 27 bf e8   st  %g1, [ %fp + -24 ]     // |
11fd8: c2 07 bf e8   ld  [ %fp + -24 ], %g1     // ptr == NULL ?
11fdc: 80 a0 60 00   cmp  %g1, 0                // |
                                   // if equal jump to 12010
11fe0: 02 80 00 0c   be  12010 <add_all_nodes+0x4c>
11fe4: 01 00 00 00   nop
11fe8: c2 07 bf e8   ld  [ %fp + -24 ], %g1     // load ptr
11fec: fa 07 bf ec   ld  [ %fp + -20 ], %i5     // load total
11ff0: c2 00 40 00   ld  [ %g1 ], %g1           // load ptr->val
                                   // total = total + ptr->val
11ff4: 82 07 40 01   add  %i5, %g1, %g1
11ff8: c2 27 bf ec   st  %g1, [ %fp + -20 ]     // |
11ffc: c2 07 bf e8   ld  [ %fp + -24 ], %g1     // load ptr
12000: c2 00 60 04   ld  [ %g1 + 4 ], %g1       // load ptr->next
12004: c2 27 bf e8   st  %g1, [ %fp + -24 ]     // ptr = ptr->next
                                   // goto 11fd8
12008: 10 bf ff f4   b   11fd8 <add_all_nodes+0x14>
1200c: 01 00 00 00   nop
12010: c2 07 bf ec   ld  [ %fp + -20 ], %g1
12014: b0 10 00 01   mov  %g1, %i0
12018: 81 c7 e0 08   ret
1201c: 81 e8 00 00   restore

```

Figure 3.15: Compiled Machine Code

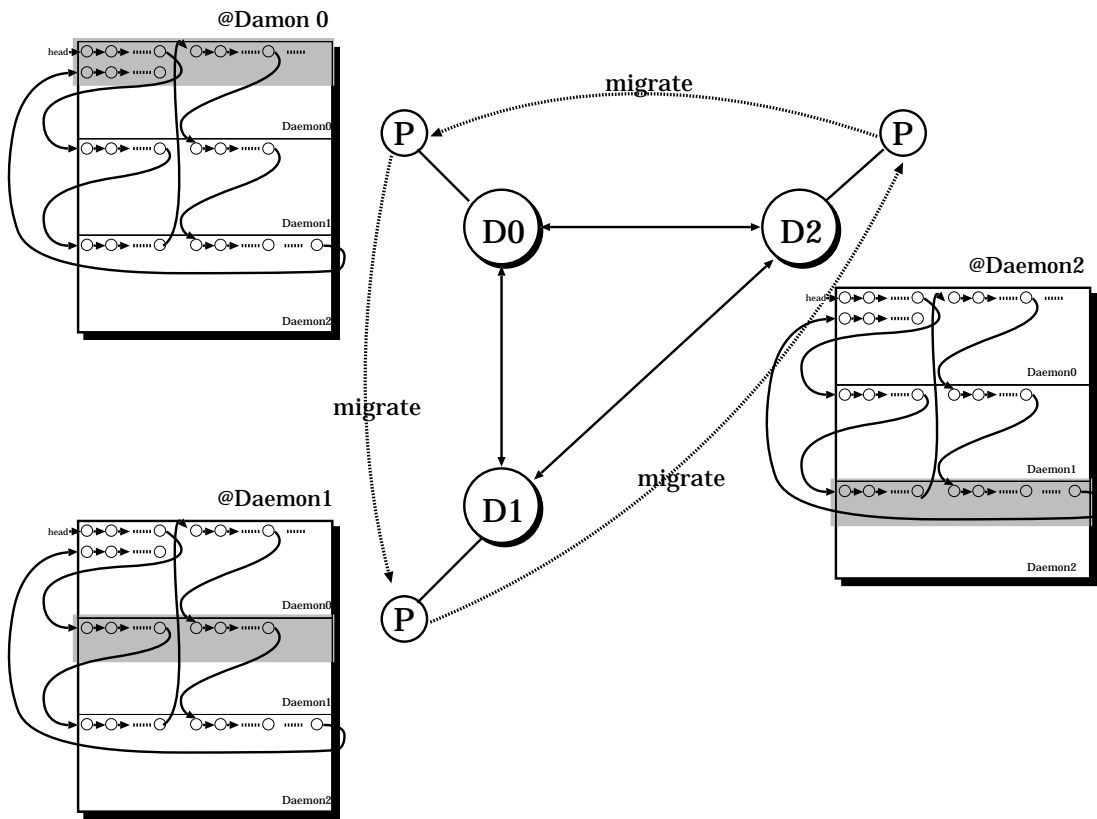


Figure 3.16: Following the remote pointer

is assigned a distinct ID by a pair of originated daemon number and a process number. Time associated with each node is relative to the startup time of each daemon and cannot be used to order some events on different daemons. For sorting the events, we used lamport's logical clock [42]. For implicit hop and remote read/write, the trace shows code and memory address causing a SIGSEGV and also 'st(store)' or 'ld(load)' information. This information is used by the agent process and the daemon to determine when to adapt a different migration type and is also helpful for debugging purposes.

3.5.2 Ghost Process

The differences between multi-threads and multi-processes are not only the overhead of context switching but also the cost of creating and destroying its computing entity. In message passing and DSM systems, the process lives for the entire lifespan of the application. In our SPM system, for each migration, a new process is created and the original (source) process is killed. There could be thousands of migrations which makes the overhead not negligible.

To overcome this overhead, the daemon keeps track of a fix number of discharged agent processes, and activates them when a new process migration request arrives. We call these processes, *ghost process*. Originally, necessary information for migration was passed as arguments to *exec* system call when the process is first created. Now, each agent process will have a FIFO opened for the daemon to write to (Fig 3.3). Whenever an agent process finishes sending its copy, it lets the daemon know by sending the process id and sleeps until the daemon contacts its FIFO. Figure 3.18 shows the lifespan of an agent process. To avoid having too many ghost processes, if there are more ghost

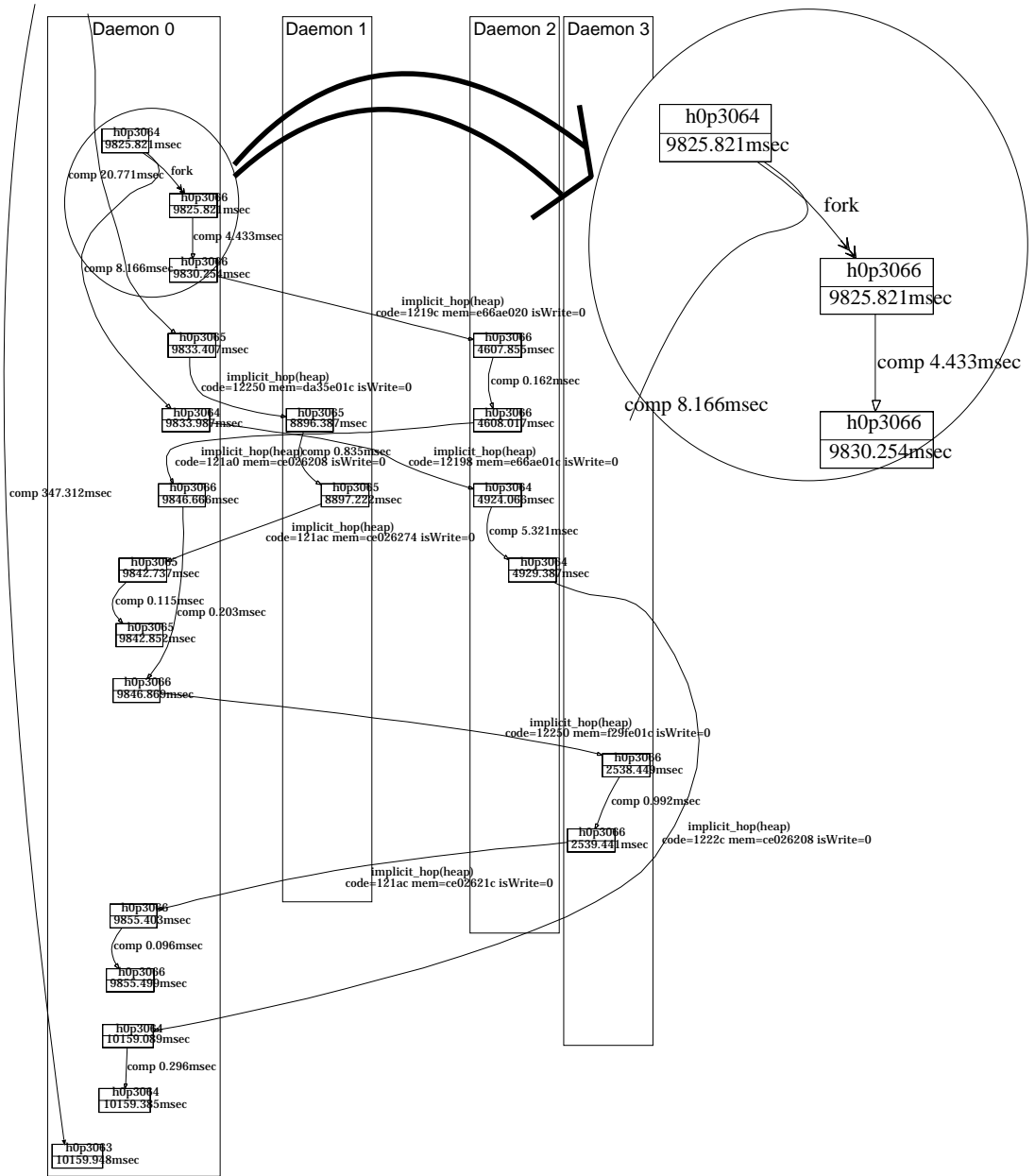


Figure 3.17: Tracing Execution

processes than a certain threshold, the daemon simply kills the agent process by sending SIGTERM signal. Actual performance result is covered in section 6.2.

3.5.3 Parallelism

This is merely a feature, but as a result of our design choice, the process can easily *fork* a new process and synchronize with other agent processes through UNIX semaphores. Each agent process and the daemon are all native UNIX processes so the scheduling and synchronization can be controlled by the operating system. This simplifies the system design and reduces the load of our daemon process.

Note that each agent process has some private data that is never migrated: file descriptors (for *accept* & *fifos*), ident key for tracing and so forth (Fig 3.12). Thus, *fork* system call has to be intercepted in order for the child process to reinitialize these values. Also, if the tracing is activated, the parent process will notify the daemon about the new child process.

3.5.4 User Interface

Usability of the system is also an important factor. It would be quite a burden if the user had to login to all the hosts to startup the daemons for each testing.

In our system, daemons are started from one command, *spmstart*. A user provides a list of hostnames for each daemon in *.spm_profile* file. For each node entry, a daemon is started on that host. In the example, *spmstart* would start 1 daemon on mendel-1, 1 daemon on mendel-2 and 2 daemons on mendel-3. The first node entry, on mendel-1, is considered as a master daemon. Once

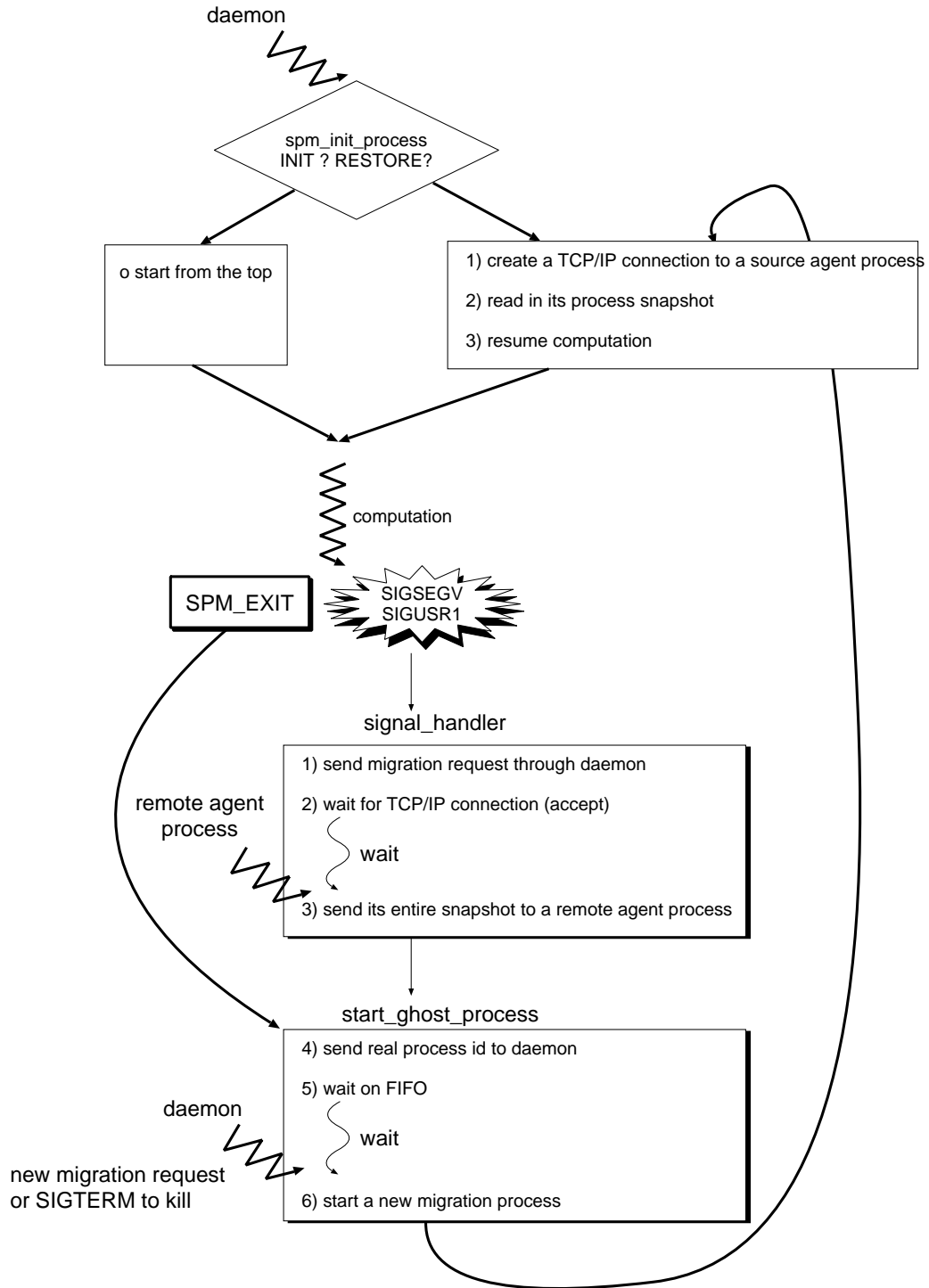


Figure 3.18: Reusing Agent Process

```
mendel-1%cat .spm_profile
node = mendel-1.ics.uci.edu
node = mendel-2.ics.uci.edu
node = mendel-3.ics.uci.edu
node = mendel-3.ics.uci.edu
```

Figure 3.19: .spm_profile

the daemons are up, they would create a fully connected TCP/IP network and wait for an agent program to be injected.

There are several options you can pass to *spmstart* (Fig 3.20). It can open a X-window for each daemon's output or it can tell the daemons to redirect their outputs to local disks. One of the useful option is '-f' which would keep the *spmstart* command on the foreground until the master daemon is killed. Combined with agent process function call, *spm_finish()*, that kills all the daemons, shell scripting becomes relatively easy. Figure 3.21 shows a short shell script, that tests different parameters.

3.5.5 Program Numbering

Depending on the OS, executable filename with the path could get as long as 255 bytes or more. As mentioned previously, we want to keep the size of the data exchanged between daemons as small as possible. Sending the entire filename for each migration goes against our intention.

To replace these filenames with a number, we have a simple routine when an agent program is first injected to the system. Whenever a new agent program is injected, a master daemon gives a new program number and broadcast the

```

mendel-8:~[89]%spmstart -h
spmstart options
-D: display_host
-d: debug version
-t: trace the processes and output in dot format
-x: open xterm windows
-w: wait for one key enter before exit
-s size_of_heap: specify the (total) size of the shared heap
-S remote_shell: specify which remote shell to use
-X remote_shell: specify which remote shell to use
-g number_of_ghost_processes (default 0)
-n number_of_hosts: takes #hosts from the .spm_profile
-p profilename: specify profilename (default .spm_profile)
-r:redirect its output to a file
-R directory:redirect its output to a file under specified
    directory
-f:leave the process on the foreground until daemon dies
-y:activate remote read/write

```

Figure 3.20: spmstart options

```

mendel-1% cat testbisort.sh
#!/bin/csh/ -f
pmexec -D bitonic.pm-parallel 26 17 1 &
pmstart -y -r -R /var/tmp/test_1 -g 1 -f
pmexec -D bitonic.pm-parallel 26 17 2 &
pmstart -y -r -R /var/tmp/test_2 -g 1 -f
...

```

Figure 3.21: simple shell script

filename and the number to all the other daemons. Once the acknowledgment is back, this program number can be used to represent certain agent programs. For this reason, a user can only inject a new agent process from a *master daemon*.

This is a very simple procedure, but is a necessary step to keep the overhead of migration low.

3.6 Requirements and Limitations

In this section, we briefly mention the running environment and limitations of our system.

3.6.1 Architecture

Our current version works on SUN SPARC machines running Solaris. There are several requirements, but the first 2 things to note would be

NFS support Initial version of our migration packed the entire state into one file and let the remote process pull back the file and unpack (section 3.3.2). Also, executable code is never transferred and assumed to be shared on all the hosts under the same directory.

Homogeneous Environment Not only the type of the Operating System needs to be identical, but also the version, release level (patch number), and even a processor type have to match. In order for the migration to work, all the states (stack, heap, text, static) have to start from the exact same address and content of the registers have to match.

The NFS support requirement could be lifted by providing an executable code transfer and file managing features. However, “Homogeneous Environment” is a strict requirement. It comes from the programming model design which allows migration at any point in the execution code. Heterogeneous process migration systems do exist [15] [63] [64], however, migration points are fixed at compile time and for each migration point they generate conversion codes to obtain heterogeneity. If we apply the same concept to our system, the code size would explode since each pointer dereference is a potential migration point. Furthermore, [15] shows that having a migration point inside the computation intensive section would slow down the code significantly and [63] mentions that their system cannot optimize across migration points. Heterogeneous migration systems with any migration points would be another open question for future research.

3.6.2 Limitation on Transparency

One of our goals is to ensure that the agent processes always produce the same result irrespective to how the data is mapped. Of course, race conditions, random number generation, and other classic distributed computing problems have to be resolved by the user. However, there is a more fundamental difficulty at our system level. It is coming from the fact that our SPM shared library runs on user-space (Fig 3.22). Our shared library has no mechanism of extracting data stored at the kernel space. In other words, some system calls which store data at kernel space have a risk of losing data when migration happens. Condor [46] gets over this problem by intercepting some of the system calls and replacing them with their own user-level library. For example, *open* can

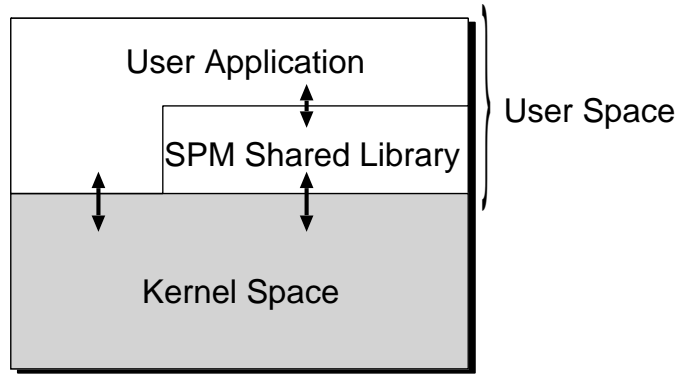


Figure 3.22: Agent Process: User Space and Kernel Space

be replaced by the shared library in order to trace the filename, mode, and offset so that it can be re-opened and adjusted at migrated site. However, as the authors of Condor write, “no way we can save all the state necessary for every kind of process.” So far, we have only replaced a limited number of system calls commonly used on our systems.

For example, the following two function calls were replaced.

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

These functions lock and unlock the semaphore referenced by *sem* pointer. Intuitively, it seems safe to simply pass a pointer pointing to a global shared heap. It would definitely work when two processes run on a single host environment (Fig 3.23). However, if run on a multiple daemons, in our environment, it crashes the agent process (Fig 3.24). Depending on the implementation of the system call, it might involve a ‘pre-work’ that stores some data at the kernel level. However, our migration only happens when the remote pointer is dereferenced. This means that the result of ‘pre-work’ might get lost on the way. In order to solve this situation, we need to intercept the system call

and migrate the agent process immediately to the place where the reference is pointing at (Fig 3.25).

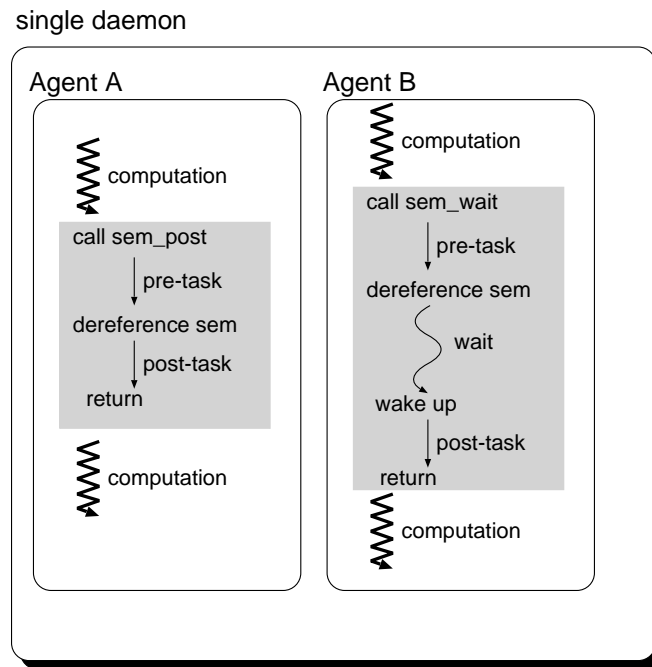


Figure 3.23: Synchronizing within a single daemon

3.6.3 Snapshot Efficiency

In process migration we do not need to transfer the entire virtual address space since only part of it is used by the application. However, it is not always easy to obtain the exact size of memory being used.

CPU state (registers) size Known at compile time. Fixed.

stack size Stack Pointer pointing to the top of the stack, thus easy.

heap size We rely upon native *malloc* to manage the heap area, thus there is no information on which pages are actually used in the heap area.

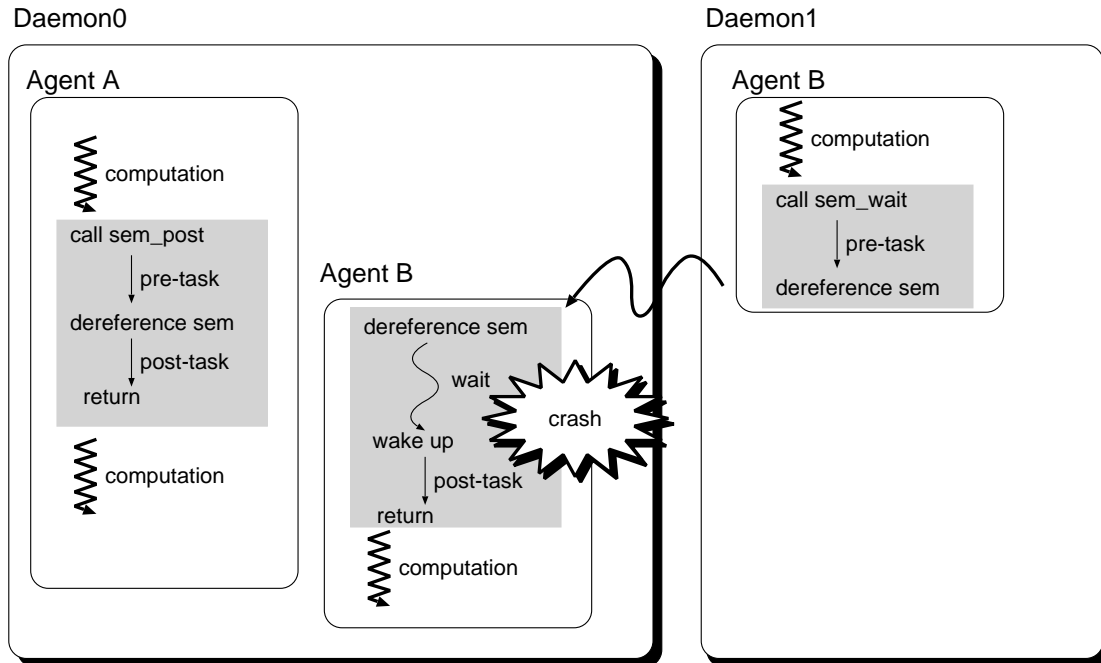


Figure 3.24: Migrating in the middle of the system semaphore call

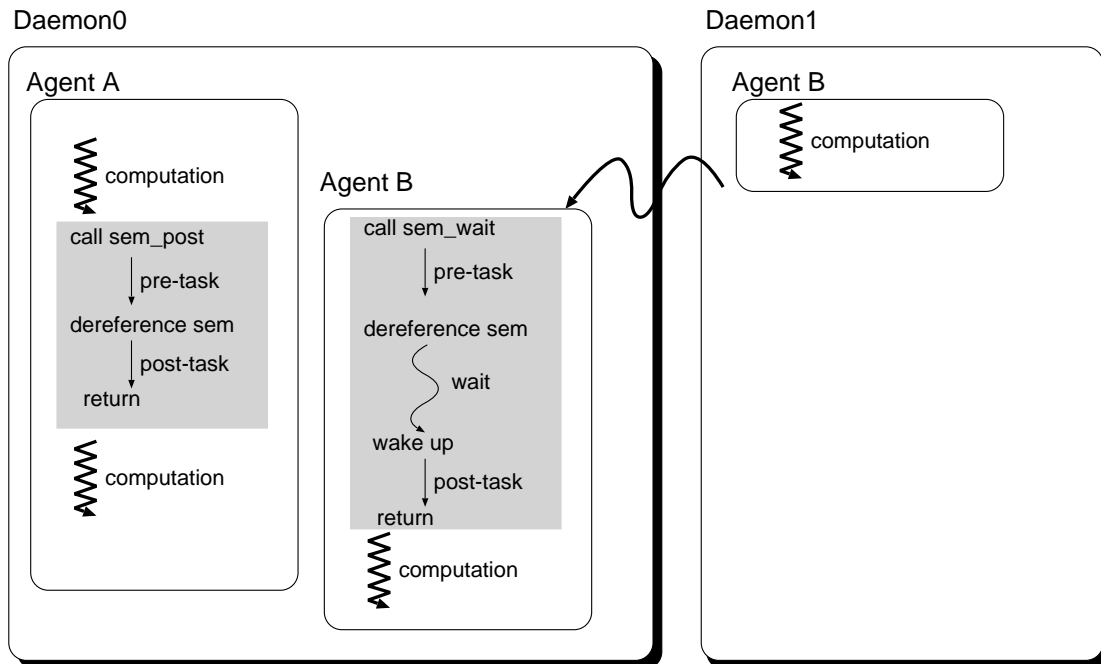


Figure 3.25: Migrating before the system semaphore call

Library implementation differs from system to system and we can only obtain the entire heap size by multiple of the virtual page size (8 Kbytes in Solaris). Note that this is different from the ‘global shared heap’. Local heap is used when an agent process wants to allocate a private space not shared with other processes.

static data size There is no constraint for C compiler and Linker to put all the static data into one block. Thus, static data could be scattered inside this area requiring the transfer for the entire static data space.

So depending on a compiler or a linker used, efficiency of our snapshot could vary significantly. One solution would be to use our own malloc implementation to keep track of the heap space and tweak the compiler and linker to pack the static data space.

Chapter 4

Remote Read/Write

In chapter 3, we have covered how the agent process can implicitly migrate from one host to another to obtain a local access and give a transparent view of a global shared heap. When we have computation intensive code that utilizes and traverses the dynamic data structures, process migration gives better execution speed over pulling and/or replicating the data in DSM. However, there is often a case where agent processes simply hop back and forth for accessing data at the border of the two daemons. In this chapter, we will show how this overhead can be reduced by a simple word-wise remote read and write and still provide the transparency of data access to the application program. Followed by a short example, implementation details are divided into two parts: *how* to remote read and write (section 4.3) and *when* to remote read and write (section 4.2).

4.1 Ping-Pong Effect

In chapter 1, we have argued the advantage of using process migration for applications traversing dynamic data structures. However, we do want to avoid the overhead of process migration whenever possible. One of the extreme cases we often observed is an agent process hopping to a remote host only to update a single variable/pointer and immediately hopping back. Even in a very simple data structure such as in a linked list (Fig 3.13), we can observe this problem where a process migrates back and forth to write to the *prev* \rightarrow *next* value. An simple expression such as

$$cur \rightarrow val+ = prev \rightarrow val;$$

would also make the process migrate back just to read a single value.

To resolve this problem, we will introduce a remote read/write feature that agent process uses to request to remotely read or write a word (1~8 bytes) and avoid unnecessary process migration. At the same time, we will keep the same level of transparency to the user, meaning the decision making and the actual data movement are going to be done by our system without the user's involvement.

Ideally, we want the compiler to decide when to remote read/write, but complicated layouts of dynamic data structures prevent us from figuring out the migration pattern at compile time. As a second option, our system makes the decision at runtime looking at how the process moved in the past.

4.2 System Implementation

Once the system decides to carry out the remote read/write based on the heuristic (section 4.3) the agent process will send its request through the daemons (Figs 4.1 and 4.2). To keep the daemon as stateless as possible, the information of the requesting agent process is encapsulated inside the request packet. When the daemon receives the reply, it forwards the data to the appropriate agent process through FIFO. Note that the agent process has to stop its computation for remote read but can continue executing for remote write (section 4.2.2).

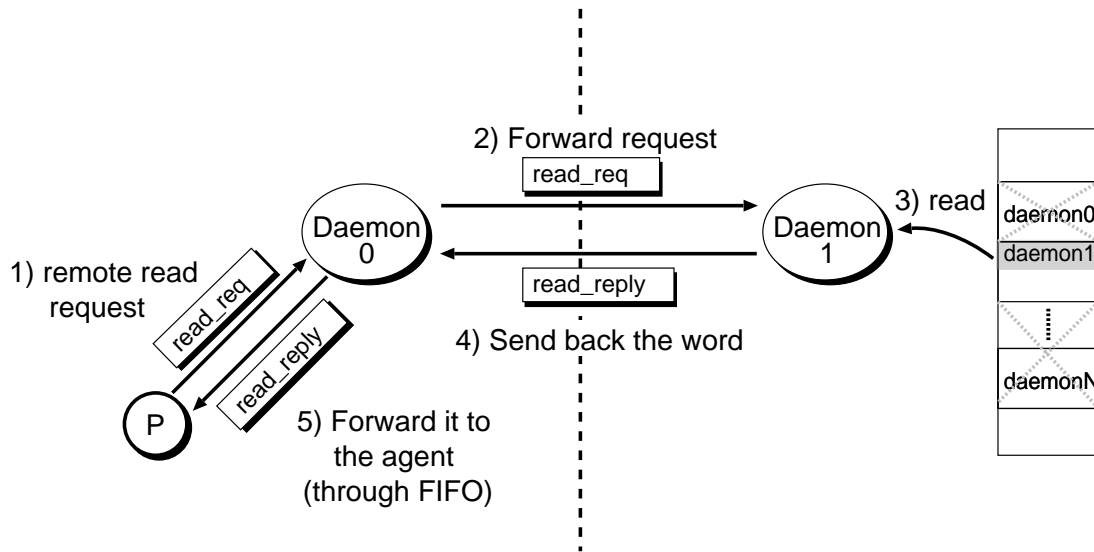


Figure 4.1: Steps of Remote Read

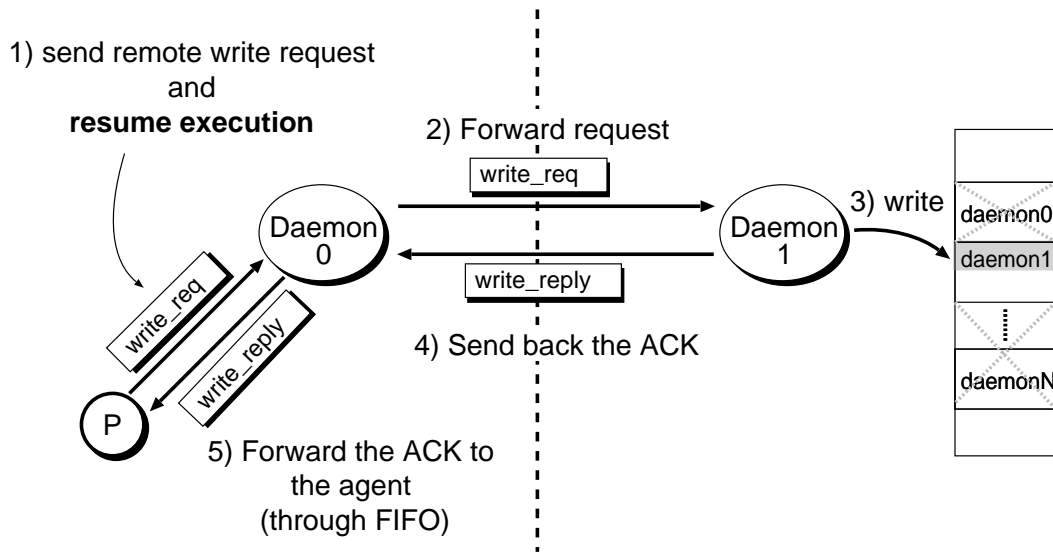


Figure 4.2: Steps of Remote Write

4.2.1 Single Line Interpreter

Simply put, by sending the request, agent process is asking the daemon to perform a task on its behalf. To illustrate, let's use the example of linked list traversal from figure 3.15. Assume that SIGSEGV was caught at code address 0x11ff0 where

$$ptr \rightarrow val$$

is being loaded to register g1.

```
11ff0: c2 00 40 00  ld [%g1], %g1 // load ptr->val
```

It is trying to load 4 bytes from where register g1 is pointing at and storing it to register g1. SIGSEGV occurred because register g1 (pointer) was pointing to a memory that had been blocked (section 3.2.3), which indicates that the shared memory is on a remote daemon.

Thus, an agent process could ask the daemon for data from `%g1` to `%g1 + 4` and write the value to `ucp->uc_mcontext.gregs[REG_G1]` (section 3.3.4). Remember `ucp` is the pointer to the `ucontext` structure that contains the full information on when the signal event occurred and also used when returning and resuming execution. So after incrementing the program counter to skip this code line and signal handler returns control, it is as if the line `0x11ff0` has been executed.

In order to perform this task, signal handler has to interpret the binary code `c2 00 40 00` and send the request to daemon accordingly. Followings items have to be understood.

Opcode(Load or Store) Reading or writing to memory

Memory Address Virtual Memory Address being referenced

Number of Bytes Single byte to double word (1-8bytes)

Register in Load: target register, in Store: source register

Table 4.1 shows part of SPARC Assembly Language for loads and stores that need to be interpreted at runtime by the agent process.

4.2.2 Parallelism on Remote Write

A straightforward way to implement the remote read/write is to always wait for the reply before resuming the computation (Fig 4.3). For remote read, this is reasonable since it needs the data to continue the computation. But this does not hold for remote write. Agent process is able to immediately resume

Opcode	Argument	Operation
ldsb	[address], reg	Load signed byte
ldsh	[address], reg	Load signed half-word
ld	[address], reg	Load word
ldd	[address], reg	Load double word
stb	reg,[address]	Store byte
stsh	reg,[address]	Store signed half-word
st	reg,[address]	Store word
std	reg,[address]	Store double word

Table 4.1: Load and Store instruction set

the computation from the next code line as long as we do not introduce an inconsistency.

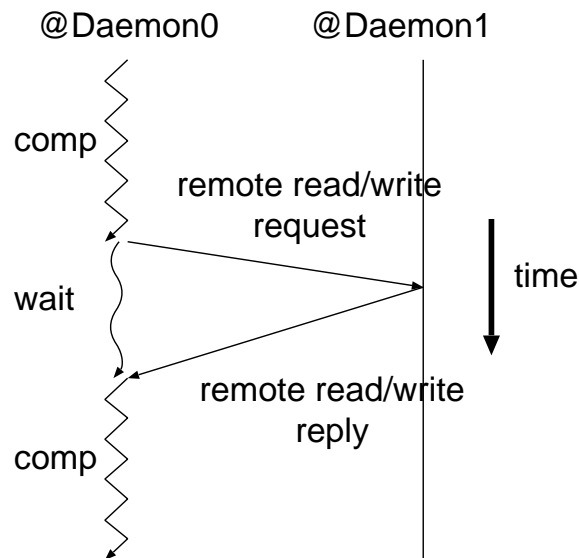


Figure 4.3: Waiting for the Remote read/write Reply

Inconsistency could only happen when the agent process or second remote read/write request packet overtake the first write request. (Fig 4.4). A subtle case would be when the forked child agent process gets ahead of the request packet (Fig 4.5). Even though there is only one TCP/IP socket connecting

the two daemons, the above inconsistencies still happen because the agent process could migrate along different paths and also the page migration feature (chapter 5) could break the order of request arrivals.

In order to protect from the above inconsistencies, each agent process waits for the remote write reply at both the top of signal handler and before the *fork* call. Figure 4.6 shows how the computation and remote write can be overlapped.

Performance of remote read/write is covered in section 6.3.

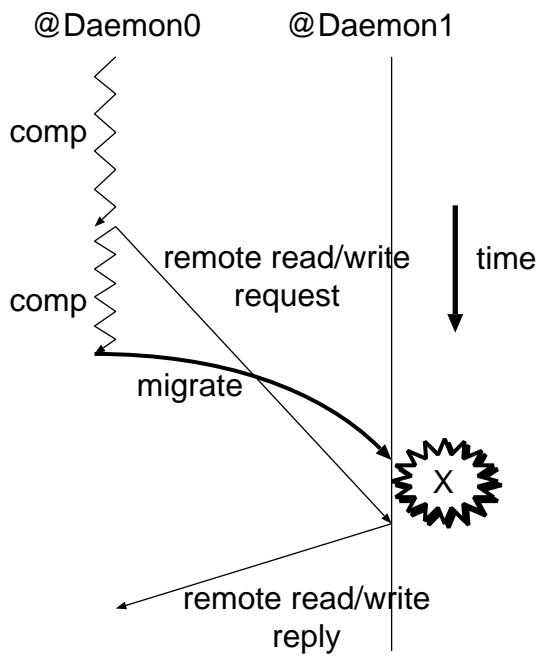


Figure 4.4: Inconsistency case 1

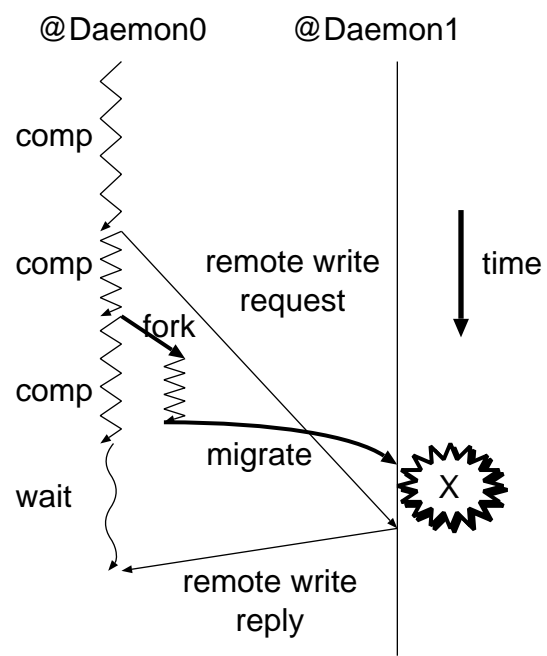


Figure 4.5: Inconsistency case 2

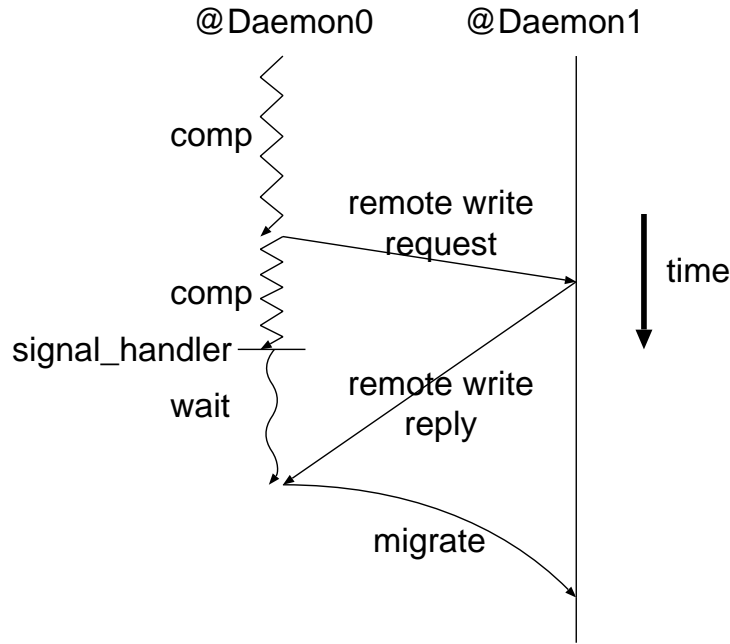


Figure 4.6: Overlapping Computation and Remote Write

4.3 Runtime Heuristic

Before sending a remote read/write request to a daemon, the agent process has to have a certain standard for making the decision. In this section, we will look into some of the idea behind our heuristic model.

4.3.1 Compile-time vs Run-time

To determine whether to remote read or write, the first question to ask is when to make the decision: at compile-time or run-time? As shown in table 4.2, compiler approach would be ideal if it can predict the data mapping and application's behavior at compile-time. However, the initial motivation of using the process migration over message passing was exactly the opposite: for programs with dynamic data structures, it is extremely difficult to determine

the data mapping at compile time. Also, with the page migration feature in chapter 5, data mapping can change dynamically. Thus it is essential for agent process to be able to change its behavior by looking at the actual data mapping at runtime.

Note that by making a wrong choice between migration and remote read/write, it would only hurt the overall speed of the application but the correctness of the program is still being preserved. This means that the system may take a try-and-error strategy to use the past data access pattern to make a better decision on the future migrations. Of course there is some overhead at first, but this running costs could be amortized over thousands of iterating process/data migrations.

	Pros	Cons
Compile-time	No overhead at run-time.	Cannot adapt to different data mappings. Requires users' involvement.
Run-time	Flexible. It can re-adjust when the data mapping changes	Overhead of logging and deciding

Table 4.2: Deciding on remote read/write

4.3.2 Daemon vs Agent Process

In section 4.3.1, we have decided to make the decision on the fly. Now, the next question: *who* is going to be responsible, a daemon or an agent process? Daemon can have a wider view of the system whereas an agent process only knows about itself but in more detail. By having the daemon make the decision, each agent process will not have to go through the same redundant

learning process. The drawback is the overhead of gathering information and the risk of having identical decision by all the agent processes. A dynamic data structure may have irregular data mapping that requires each agent process to behave differently. Moreover, there is one more type of migration, page migration, which complicates the heuristic. It is essential to have the overhead of heuristic as small as possible to avoid slowing down the application. To simplify the learning process, we split the responsibility for the daemon and the agent processes. Each component is responsible for one decision.

Agent Process Decide between remote read/write and process migration

Daemon Decide between granting the agent process' request and requesting a virtual page (detail in chapter 5)

4.3.3 Heuristic Log

Agent process keeps a log of its past behavior and utilizes it when migration decision has to be made. There is quite a latitude on what to store and how to use the information. This makes an interesting online scheduling algorithm especially when extended with page migration (chapter 5).

As the term "heuristic" suggests, our method will not provide the best decision, but gives reasonable selection&speed-up on our target applications (section 6.3). Our heuristic algorithm is based on the fact that a code with dynamic data structures often repeats some computation by a loop or recursive calls. Try-and-error approach attempts to discover this pattern in order to avoid the (nearly-)empty migration.

In order to come up with the information to log, we will first examine the

case when remote read/write helps. Figure 4.7 shows the trace of linked list creation example from section 3.4 (abridged code in Fig 4.8).

When the agent process is injected, it allocates 10000 nodes on daemon-0 (line7-10). Next, at line 7, an agent process explicitly migrate by the `spm_malloc` call (A). At the new daemon-1, it allocates a space for 1 node and line 9 causes the agent process to migrate back (B). Immediately after at line 7, agent migrates back to daemon-1 again by the `spm_malloc` call. Now, we can see from the trace that the computation time after migration (B) is very short (0.207 msec) and also it migrates back to where it came from. Later at (D), there is another implicit hop from the same code address with same migration pattern. Obviously, this migration (B) can be replaced by the remote write. If this seems to repeat, the agent process can decide to register the code address 0x12264 as a future candidate for a remote write call.

Based on the above observation, each agent process logs the following information inside the signal handler of the remote pointer dereference (section 3.3.4).

read or write A flag that indicates 'ld'(load) or 'st'(store). As covered in section 4.3.4, we give preference to remote write over remote read.

code address We want the previous machine code address that caused the process to migrate to the current position.

time Time spent since the last migration

source daemon id Where it migrated from.

target daemon id Where the referenced memory is located.

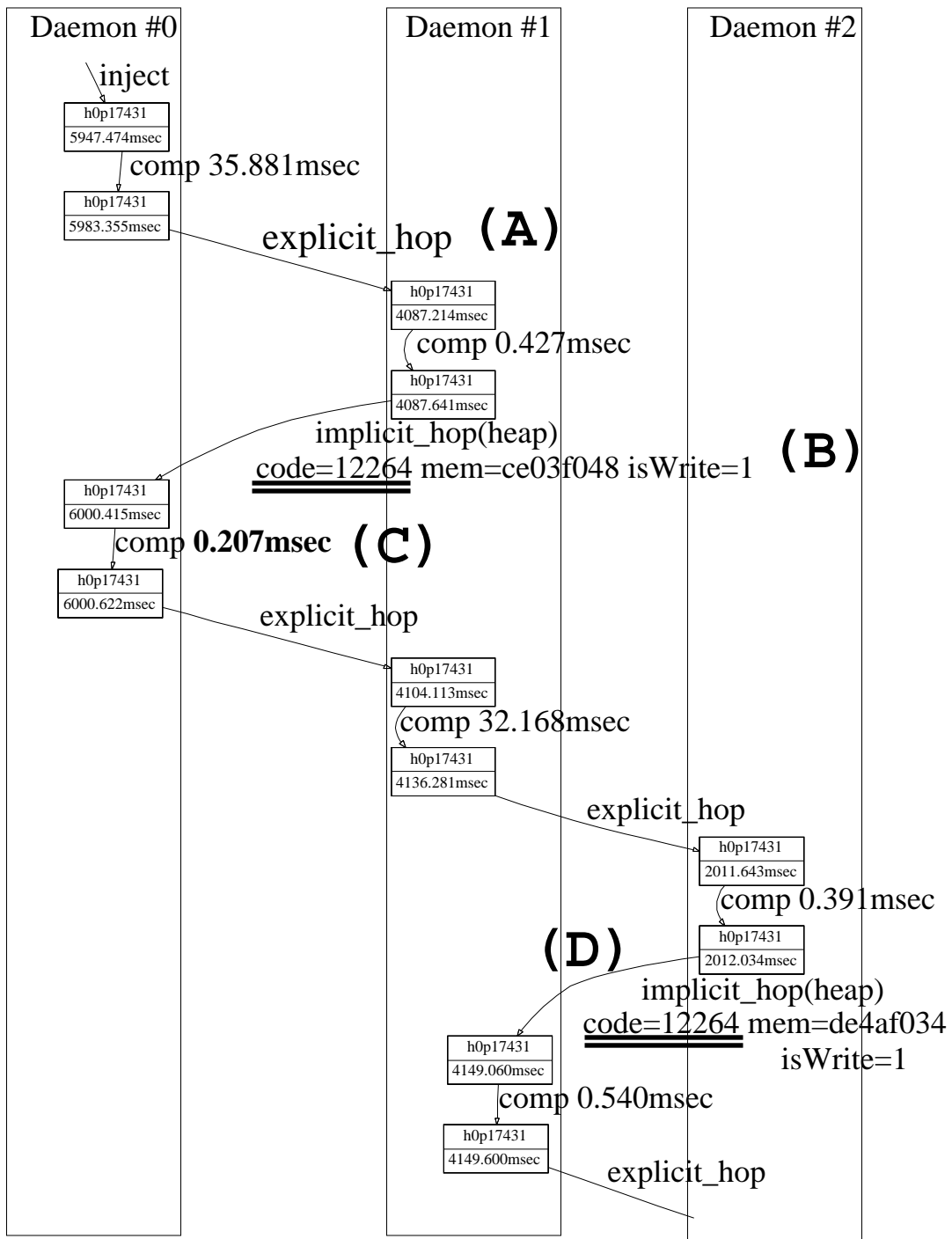


Figure 4.7: Visualized Trace of Linked List Creation

```

1 int create_linkedlist() {
2     node_t *ptr, *prev;
3     head = spm_malloc(0, sizeof(node_t));
4     head->val = 0;
5     prev = ptr = head;
6     for( i = 1; i < MAX; i++ ) {
7         ptr=spm_malloc( (i/10000) % num_daemons, sizeof(node_t));
8         ptr->val = i;
9         prev->next = ptr;
10        prev = ptr;
11    }
12    ptr->next = 0;
13 }

```

Figure 4.8: Linked List Creation Example

Pseudo code of our heuristic is shown in figure 4.9. In order to keep track of the frequency for these events, we use a fixed-size array to save the code address and the count pairs. Also, we have a separate fixed-size array for registering the code address for future remote read/write. In both cases, if the array gets full, it just throws away the oldest item. Premise behind this is that an agent process can always recover the data by going through the same trial-and-error process if necessary.

What makes this method unique is the fact that we use the agent process' machine code address for the heuristic. Just by looking at the memory address, we would not be able to determine the pattern when there are couple of different implicit migrations in the same loop. Figure 4.10 shows the trace of remote write after our heuristic decides to replace the implicit hop with remote write. If the remote write reply comes before the requesting agent waits for it, it will not block the daemon since it is communicating through the FIFO.

```

1  signal_handler(){
2     code_addr: machine code address that caused the SIGSEGV
3
4     if( explicit hop ) {
5         handle_migration();
6         return;
7     }
8     if( code_addr is registered for remote read/write ){
9         handle_remote_readwrite();
10        return;
11    }
12
13    if( previous hop was implicit
14        && previous source daemon == current_target_daemon
15        && computation time is very short )
16    {
17        increment the log counter for code_addr
18        if( counter > THRESHOLD )
19            register code_addr for future remote read/write
20    }
21
22    handle_migration();
23 }

```

Figure 4.9: Pseudo Code for migration heuristic of an agent process

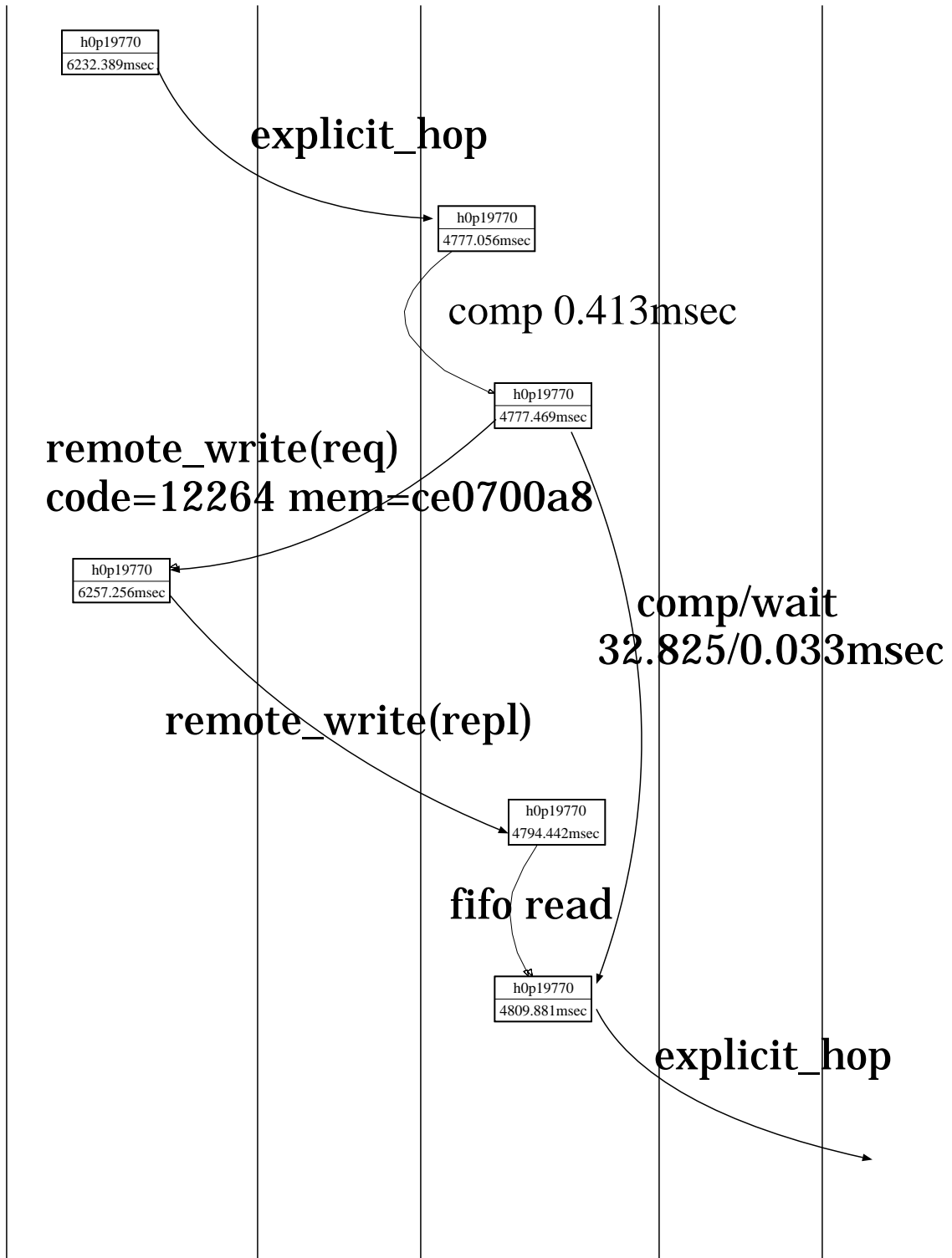


Figure 4.10: Visualized Trace of Linked List Creation with Remote Write

4.3.4 Remote Write over Remote Read

By avoiding the process migration, remote read and write both have the merit of reducing the amount of communication. In addition, remote write has the opportunity to overlap the computation with communication (section 4.2.2). As will be shown in section 6.3 overlapping computation with communication in remote write gives better speed up than remote read. In this section, we will go over an example where this situation happens and gives a simple modification to the heuristic for making a better decision.

Figure 4.11 shows a linked list code that shifts the values towards the head by one. (First value is thrown away and the last value is set to 0.) This code is silly but has some interesting attributes. In this example, we have opportunities for both remote read and remote write. Figure 4.12 shows the

```
1  prev = head;
2  cur = head->next;
3  while( cur != NULL )
4  {
5      prev->val = cur->val;
6      prev = cur;
7      cur = cur->next;
8  }
9  cur->val = 0;
```

Figure 4.11: Shifting the value

trace of the example code. There are 3 implicit hops involved.

- (A) At Line 5, load the value of `cur->val` to the register
- (B) At Line 5, save the value of the register to `prev->val`
- (C) At Line 7, load the value of `cur->next` to the register

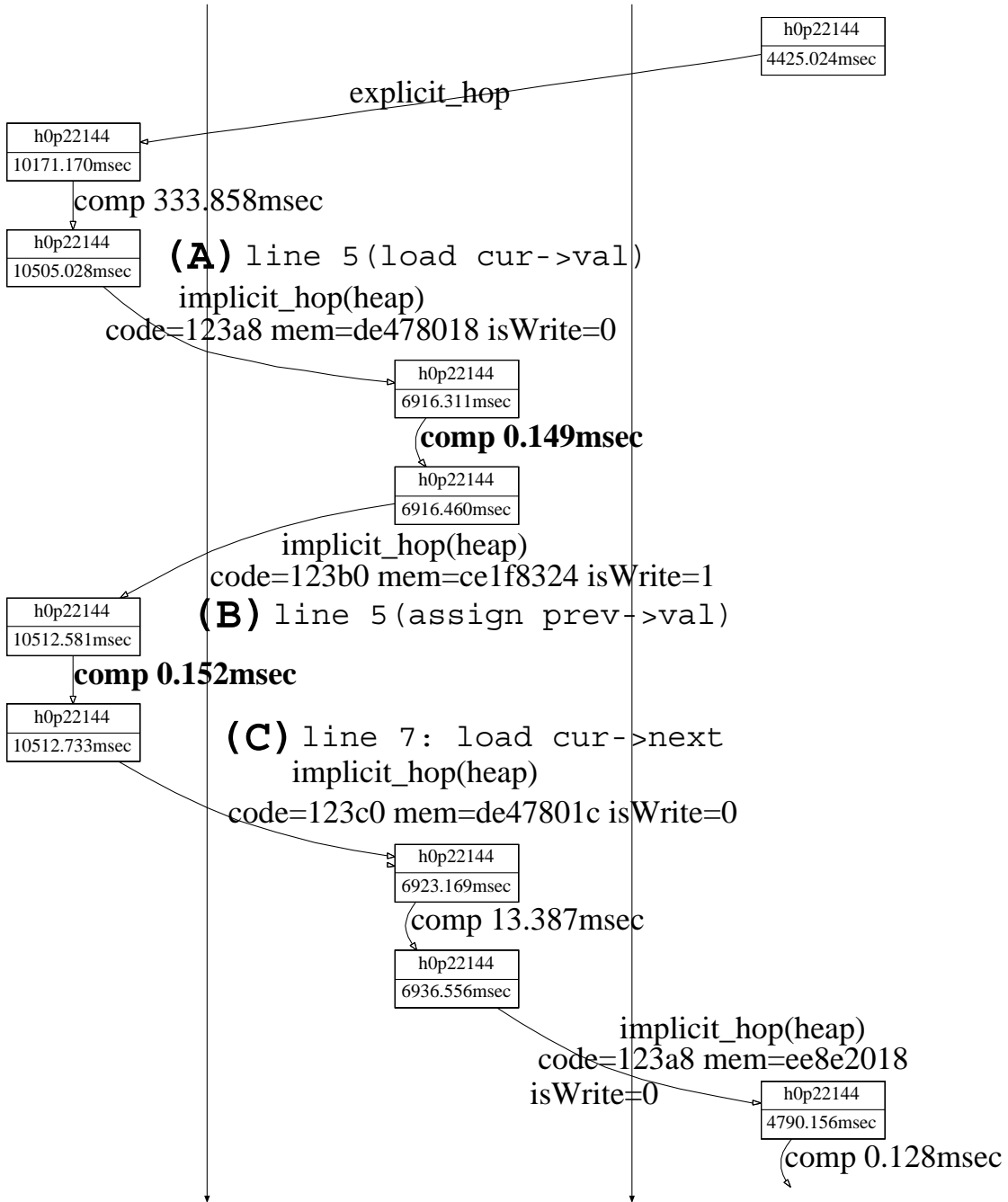


Figure 4.12: Opportunities for both Remote Read and Write (source code at Fig 4.11)

With careful observation, we see that both (A) and (B) fit our standards for remote read and write. They are implicit hops followed by a short computation and another hop back to the source daemon. Figure 4.13 shows when (A) is replaced with a remote read and Figure 4.14 when (B) is replaced with a remote write. Remote read/write could be at either (A) or (B) but not both. Once (A) is replaced with remote read, the implicit hop (B) will not occur since the process stays at the original daemon. When (B) is replaced with remote write, implicit hop (A) still exists but there is no hopping back to the original daemon which does not satisfy the requirement for remote read heuristic.

Now, what we want our heuristic to pick is remote write (Fig 4.14). However, heuristic in Fig 4.9 would most likely choose remote read since the implicit hop for remote read happens before that of a remote write. To remedy the situation, we give different thresholds for remote read and remote write. This simple modification would give higher preference to remote write and avoids losing some parallel opportunities. Drawback of this change is that potential remote read code will take a longer learning time until it starts replacing the hops.

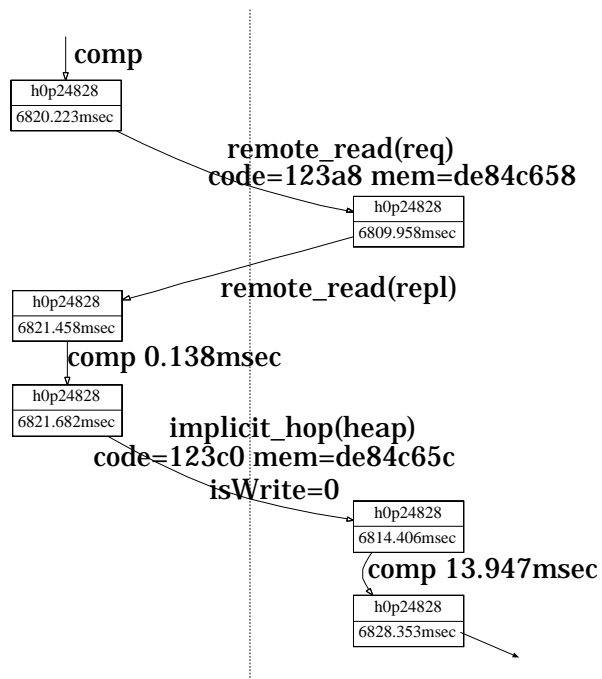


Figure 4.13: When Remote Read is used

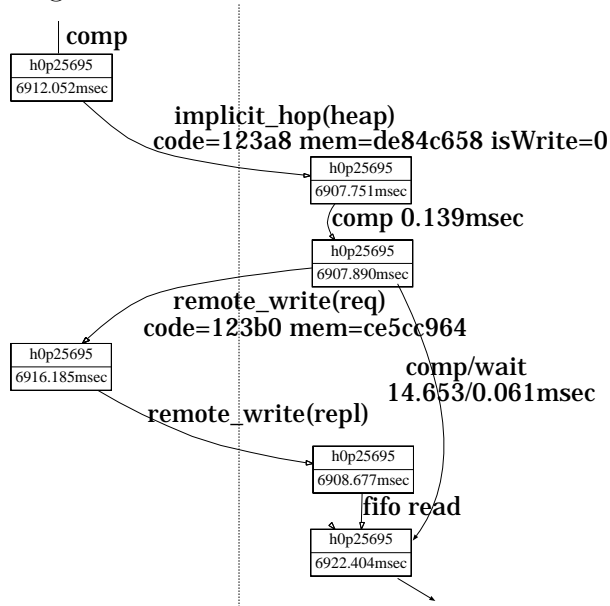


Figure 4.14: When Remote Write is used

Chapter 5

Page Migration

So far we have covered how an agent process can migrate itself or request certain bytes from a remote daemon. A heuristic on how each agent process decides has also been discussed. In this chapter, we will add one more migration choice where the entire virtual page can be moved to other daemons which have a high data affinity.

First, we will go over the problem of thrashing which neither process migration nor remote read/write can resolve. Next, details on how and when to migrate a page are covered. Last, we will show the modified version of *spm_malloc* and *spm_free* that work with our page migration feature.

Note that even when combining the three methods (process migration, remote read/write, and page migration), there is only so much the system can accomplish. If the data is ill-distributed or the access pattern is excessively complex, our system cannot magically sort out the problem. It is still a programmer's job to place the data close to each other as much as possible according to the access pattern they expect. Now, our system's job is to

improve the balance by using the information obtained at runtime which the programmer has no control over.

5.1 Thrashing

Thrashing is often used to express a state when many page faults and loads are happening in a short period of time. In our system, agent process can access local memory and remote memory alternately resulting in process thrashing or remote read/write thrashing. This could happen when two separate arrays or data structures are intermixed in a computation (Figs 5.1 and 5.2). In the extreme cases, the signal handler is activated for each pointer dereference followed by a remote read/write. Even the overheads of signal handler cannot be ignored (section 6.1.2).

By allowing the daemon to move the page(s), we try to hold pages with high affinity together to avoid high communication overheads. This flexibility at runtime is especially useful when affinity of data changes dynamically at runtime (section 6.4 Bitonic Sort).

5.2 System Implementation

Agent process migrations are guided by the location of the referenced memory. Moving the virtual address pages means it not only affects the agent processes on the two involved daemons but also the ones halfway through the migration process to access these pages. In addition, each daemon will not be able to tell the owner of the page by simple division of the memory address anymore (section 3.3.5). Therefore, unlike remote read/write, it requires more careful

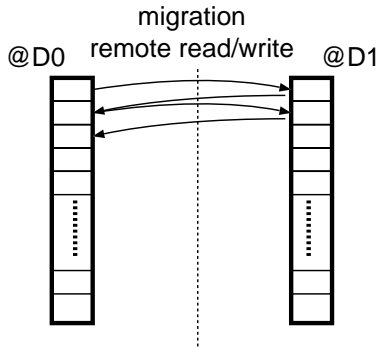


Figure 5.1: Accessing two separate arrays

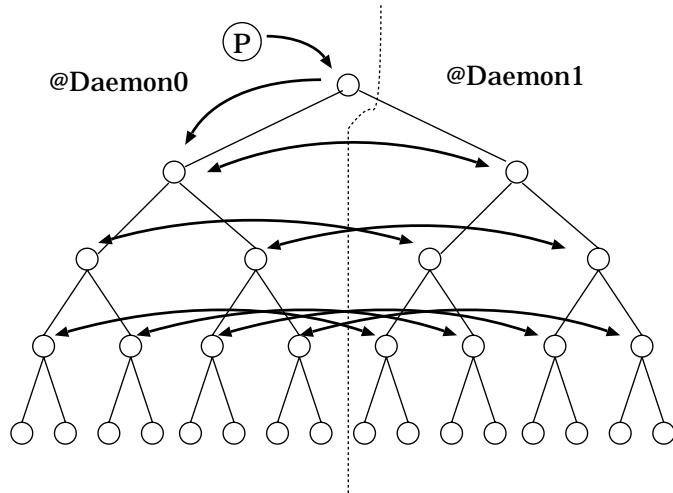


Figure 5.2: Accessing left and right subtrees

care to preserve the correctness of the application. We definitely do not want agent processes or remote read/write requests disappearing on the way.

Up to this chapter, the daemon played a minor role in the entire system, simply forwarding the migration requests from the agent processes and occasionally replying to remote read/write requests. Daemon was able to stay as a stateless server because most of the necessary information were included in the request/reply packet(*pmi*). This simple daemon design is drastically changed in the process of adding the page migration feature. However, we still try to keep the daemons' work low to avoid the daemon becoming the bottleneck of the whole system.

In this section, the following roles of the daemon are covered.

- Designating the ownership of the page
- Making sure no agent processes can access the old page contents
- Guaranteeing all the migration and remote read/write requests eventu-

ally arrive at the destinations

- Avoiding deadlock
- Reducing page mapping overhead

5.2.1 Basic Algorithm

Before digging into too much detail, figure 5.3 shows a very rough overview of how the page migration is handled by the daemons.

It is basically a negotiation process between the two daemons. While the daemon handles migration and remote read/write requests, it keeps a log of how certain pages are accessed. When the runtime heuristic *do_page_request* function meets the certain condition, the daemon will encapsulate the request packet and send the PAGE_REQUEST packet. On the receiver's side, the daemon would then examine the request by the *examine_page_req* function and decides on whether to accept the request or to reject it. If accepted, the daemon sends the page(s) and also forwards the encapsulated request back (Fig 5.4.) When the request is denied, the daemon simply handles the encapsulated request and continues (Fig 5.5) as if nothing special happened. Remote write request is analogous to the remote read request case. For process migration request, the only difference is that when page migration is accepted, the daemon tell the agent process that process migration is no longer necessary. This method has the following benefits.

- Overheads for denying the page migration request is low. (No extra packet exchange)
- It does not require any modification to the agent process.

```

1  daemon_main()
2  {
3      while(1)
4      {
5          wait for request and read
6          if( from local agents )
7              handle_local_pmi(&pmi);
8          else
9              handle_remote_pmi(&pmi);
10     }
11 }

1  handle_local_pmi(pm_info *pmi)
2  {
3      if( pmi->type != EXPLICIT_HOP
4          && do_page_request(&pmi) )
5      {
6          pmi->orig_type = pmi->type;
7          pmi->type = PAGE_REQUEST;
8          send_to_remote(&pmi);
9          return;
10     }
11     .
12     . handle regular migration
13     . and remote read/write
14     .
15 }

1  handle_remote_pmi(pm_info *pmi)
2  {
3      if( pmi->type = PAGE_REQUEST )
4      {
5          if( examine_page_req(pmi) )
6          {
7              add(pending_pages, pmi);
8              handle_page_requests();
9              return;
10         }
11         pmi->type = pmi->orig_type;
12     }
13     .
14     . handle regular migration
15     . and remote read/write
16     .
17 }

```

Figure 5.3: Pseudo Code of the daemon

- Process migration will not happen unless the page migration request is denied.

Details of *do_page_request* and *examine_page_req* functions are covered in heuristic section 5.3.

5.2.2 Sharing Page Mapping

When moving the pages, it is important that all the local agent processes and the daemon share the same page mapping. Each of them is a separate UNIX process and sets the protection of pages by *mmap* system calls. To prevent any agent processes accessing a stale data (pages that have migrated away), we introduce a protocol between the daemon and the local agent processes.

Since the daemon makes the decision on when to send the pages, most of the communication only needs to be one way, from daemon to agent processes. It is more like a broadcast, except that before proceeding to the sending phase, the daemon has to make sure all the agent processes have received the info and updated their page mappings. Using an individual FIFO to broadcast the updates is an overkill. To efficiently broadcast the update information, the daemon writes the page mapping information to the first few pages of the global shared heap and asks all the active agent processes to read from these pages. It utilizes the UNIX signal to let all the child processes, thus agent processes, know about the updates.

Page bitmaps

The daemon saves the page mapping of the entire global shared heap in a reserved space. For example, let say we reserved 800 Mbytes for the entire

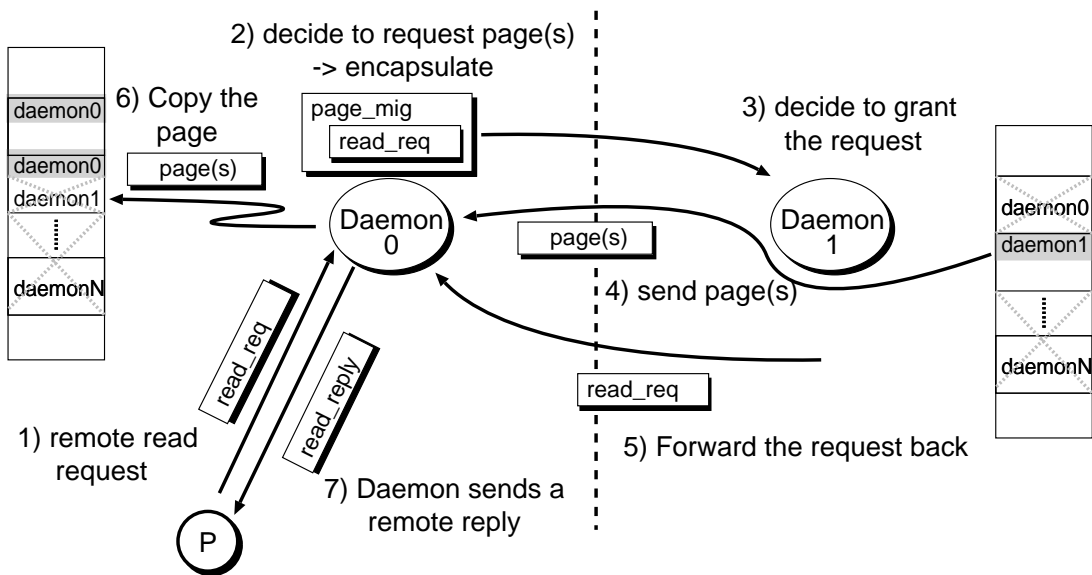


Figure 5.4: Granting a Page Migration Request

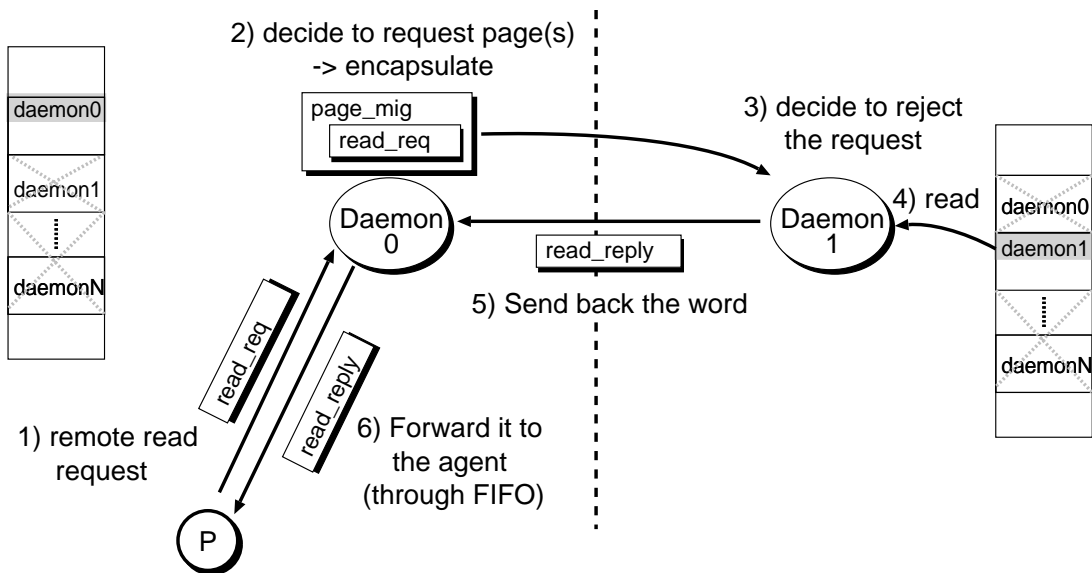


Figure 5.5: Rejecting a Page Migration Request

global shared heap. On the Solaris machines, this is equivalent of 100,000 pages (8192 Kbytes/page). Using 2bits for each page information, it totals to $100,000 * 2(bits) / 8(bits/byte) = 25Kbytes$ or about 3 pages. The information stored is as follows.

L Local page is available locally and has a page header.

M Multi page is available locally but no page header. Skipped in this section.

Explained in section 5.4.2.

R Remote page is on a remote daemon.

W Wait page is in the course of migration

Initially, the entire global shared heap is split evenly to each daemon. We refer to these daemons as *original owners* of the pages. Figure 5.6 shows a simple example when global shared heap is distributed among the daemons.

Note that in some cases, it is totally fine for the agent process not to have the latest page mapping. It is only critical when an agent process mistakenly thinks the page is available locally and accesses some stale data (Table 5.1). For other cases, agent process catches the SIGSEGV signal and looks at the page bitmaps to determine if the page is available locally or not. After the last page mapping update, some pages could have migrated from remote daemon or other agent process could have increased the size of the local shared space through *spm_malloc* call.

Broadcasting for page mapping updates

Before Figure 5.4 “4) send page” or inside Fig 5.3 *start_send_page_process* method, the daemon first needs to make sure that all the local agent processes

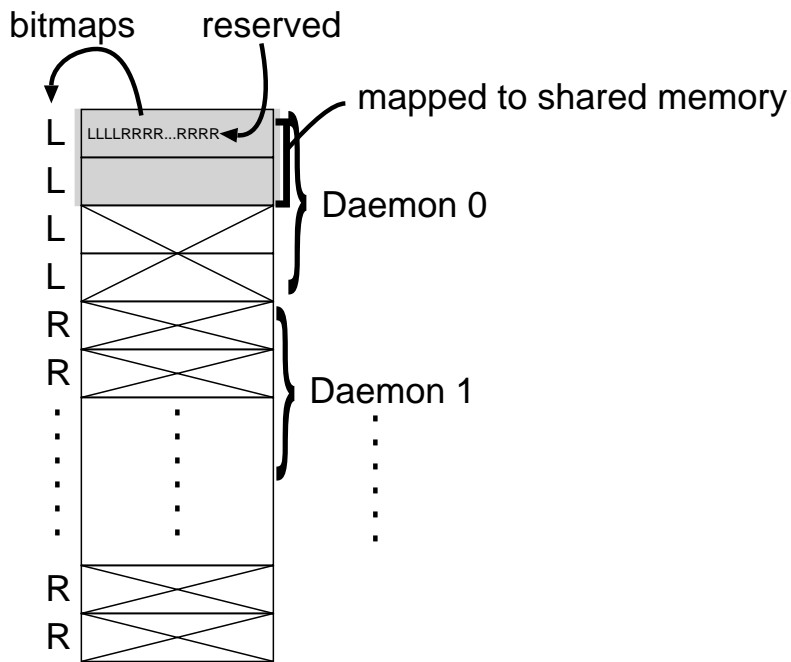


Figure 5.6: Initial Page Bitmaps

agent perspective ↓↓↓	actual page mapping	
	LOCAL	REMOTE
LOCAL	OK. It runs just like a sequential code. No signaling, thus no overheads.	NOT OK Agent thinks it is local so it will read from the stale page. No signal detection, thus program resume without knowing its mistake.
REMOTE	OK. Agent believes the page is remote but it is actually local. Signal handler can figure this out and resume execution.	OK Signal handler requests process migration or remote read/write to the daemon

Table 5.1: Potential Incorrect Page Mappings

have unmapped the page(s) it wants to send. Since the daemon is the hub for all the requests, we want to avoid busy waiting.

If there are no running agent processes, the daemon simply proceeds to the actual page sending process. Otherwise, it will set the shared counter to the current number of active agent processes and signal the child processes. Note that it is a single system call which will signal all the active agent processes. When the agent process receives the signal (at *signal_handler_SIGUSR2*), it simply updates the page mapping using the shared page bitmaps and decrements the shared counter. Last agent process to update would acknowledge the daemon. When the daemon receives an ack locally (at *receive_PMI_DONE_ACK*), it finally start sending all the pages.

One signal-ack iteration can handle multiple page migration requests at the same time. But once the signal is sent out, the next page migration iteration cannot start until all the waiting pages have been forwarded. This is necessary to guarantee that the agent processes have received the signals and updated the page mapping properly. Figure 5.7 shows how the status of the page changes during the page migration process.

used Page used locally.

pending Page chosen for page migration. Waiting for the next page migration iteration. (Still available to local agent processes)

waiting Page started page migration process. Waiting for all the local agent process to update their page mappings. (Unavailable for agent processes to access)

sending Page is being sent.

unmapped Sending done. Page is located at a remote daemon.

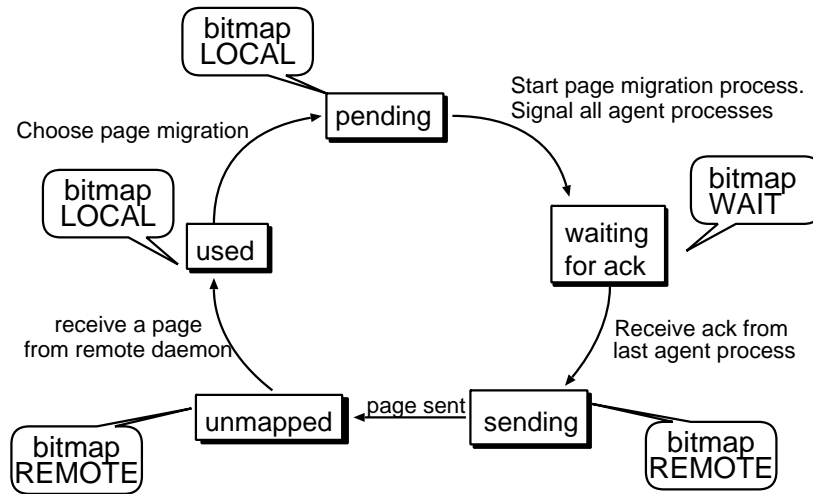


Figure 5.7: Page Mapping State Diagram

Figure 5.8 and 5.9 show pseudo codes on how the daemon and the local agent processes interacts during the page migration process. Before *handle_page_requests* is called, the daemon always adds the current page request to the *pending_requests* list. If the daemon is still waiting for an ack from last page migration iteration, it will simply continue with other work. Now, if the daemon is ready for a page migration, it changes the bitmap of all the pending pages to “WAIT” From agent process perspective, it is equivalent to “REMOTE” The only difference comes when daemon receives other requests coming for these migrating pages (section 5.2.3).

It is important for the daemons to correctly track the number of active agent processes running locally. Whenever agent process forks, injects a new program, or a ghost process wakes up, it coordinates with the local daemon through a shared semaphore to keep the value updated.

```

1  handle_page_requests()
2  {
3      if(other page migration in process)
4      {
5          return;
6      }
7      for all pending_pages
8      {
9          update the page bitmap to ‘‘WAIT’’
10     }
11     waiting_ack_pages = pending_pages;
12     empty pending_pages;
13     if( active_local_agents == 0 )
14         send_page(waiting_ack_pages);
15     else
16     {
17         set shared counter = #active process
18         signal_local_agents(SIGUSR2);
19     }
20     return;
21 }

22
23 receive_PMI_DONE_ACK()
24 {
25     send_pages(waiting_ack_pages);
26     empty waiting_ack_pages;
27     if( pending_pages not empty )
28         handle_page_requests();
29 }

30
31 send_page(lists)
32 {
33     for each entry in lists
34     {
35         send the corresponding page
36         unmap the page
37         update the bitmap to ‘‘REMOTE’’
38     }
39 }

```

Figure 5.9: Agent Process handling page mapping update

Figure 5.8: Daemon handling page mapping update

5.2.3 Pending Request Queue

While the daemon is handling the page migration, it can still receive various other requests. If the request has nothing to do with the pages in migration, the daemon can handle it without any problems. Now, the question comes when the requests are targeting a page that is in the *pending_pages* or *waiting_ack_pages* lists. Figure 5.10 shows when a remote read request asks for a page in a *pending_pages* list. In this case, the daemon can safely reply just like a regular procedure since the page migration iteration has not started for this page. Also local agent process can access memory in page 7 without any intervention.

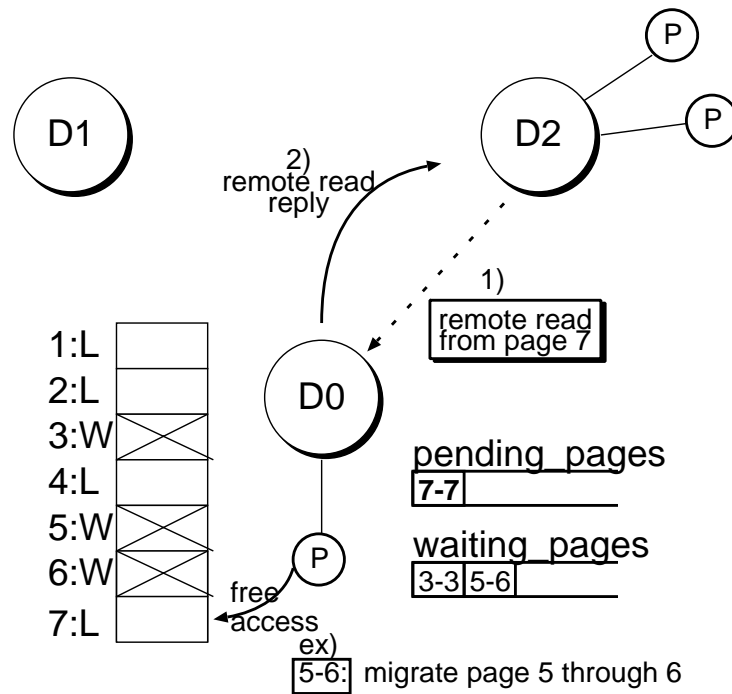


Figure 5.10: Request on a page that is in a pending list

Now, if the requests are targeting the pages in the *waiting_pages* list, it will attach those requests to corresponding page migration request info so

that once the page migrations are done, they can be forwarded to the correct destination. Figure 5.11 shows the example of such cases. At (a), two requests arrive from a remote daemon and also a local agent process tries to access a page with “WAIT” status and initiates a process migration request. At (b), since all the requests cannot be handled immediately, they are linked with corresponding page migration info to be handled later. At (c), the daemon receives an ack and sends the actual pages followed by all the pending requests. Finally at (d), the receiving daemon will handle the pending requests just as if they were regular requests.

One of the advantage of our approach is that only the *pmi* requests are pushed around and not the actual processes which could easily size over thousands of bytes. Another benefit is that the agent process does not even need to be aware of the page migration. The agent simply sends the request and waits for either a remote read/write reply or a TCP/IP connection for process migration. It does not care where they come from. The only changes we made for the agent processes were skipping local process migration for optimization purposes. (This happens when a target page happens to migrate to a local daemon.)

5.2.4 Page Update History

When mapping or unmapping the pages, our system relies on a system call *mmap*. Even if there are thousands of pages to map, as long as the protection type stays the same, a single system call can cover all the consecutive pages. Initially, without the page migration, each agent process only had to call *mmap* at most three times for the global shared heap space (Local-Remote or Remote-

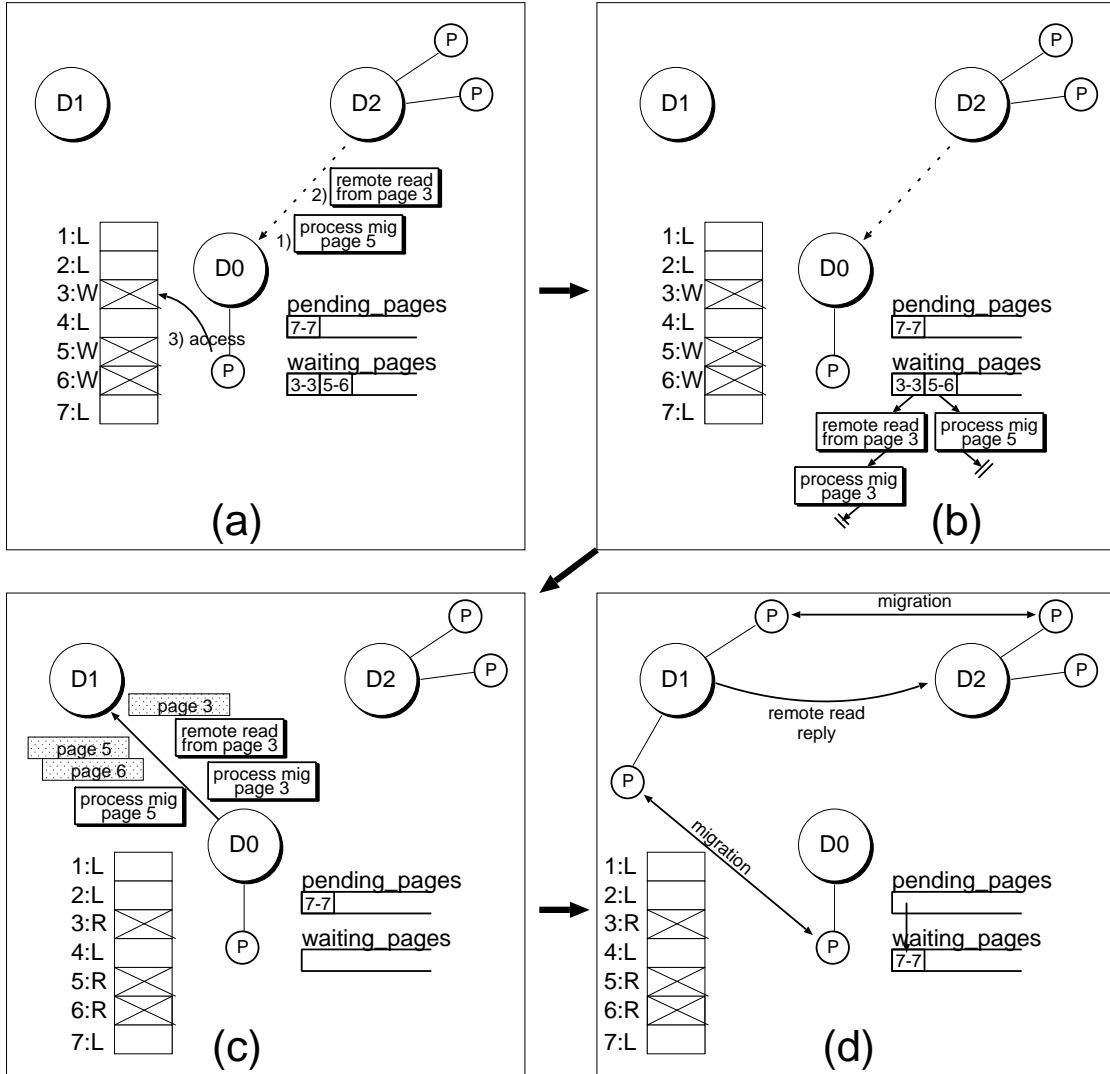


Figure 5.11: Request on pages in the *waiting_page* list

Local-Remote).

Now with the page migration introduced, granularity of the protection types could be much finer. In the worst case, *mmap* has to be called for each page, which will obviously result in a significant overheads. Moreover, mapping not only needs to be called at start up, but also after each page migration iteration that the daemon performs. To prevent this overheads becoming a bottleneck, we have made some minor changes to the broadcasting and shared page bitmaps.

Range of pages to update When broadcasting a signal for page mapping updates, there is no need for remapping the entire global shared heap. Instead, the daemon can inform the agent processes about the pages that have been modified. Since there could be more than one page migration for a single page migration iteration, the daemon simply tells the range of pages to be updated through the reserved shared space.

For example, in figure 5.11-a, the daemon would set the range of pages from 3 to 6 before signaling the agent processes.

History of changes When a complete new process is created by a daemon, entire global shared heap definitely needs to be mapped. However, in our system, we often rely on the ghost processes (section 3.5.2) to expedite the process migration procedure. Depending on how many page migrations occurred since the agent process became a ghost process, this ghost process could already have most of the current page mappings stored. To take advantage of this attribute, a daemon keeps track of the past

mapping updates from page migrations.¹ Each agent process will store a *mapping_version* that will be compared against the daemon's page mapping version. By looking at the difference of these two versions, the agent process can tell how many updates to look back at to bring itself up-to-date. If the difference is bigger than the available past updates, it simply remaps the entire global shared heap space.

5.2.5 Locating the Page

Before page migration, the location of the page was able to be determined by a simple arithmetic (section 3.3.5). Now, with pages moving from one daemon to another, we take a straightforward approach where the original owner of the pages keeps track of their latest locations. After each page migration, the daemon sends an update of the page's new location to the original owner.

When a daemon receives a request packet targeting a page it does not have, it simply forwards the request to the original owner of the page. If the original owner does not have the page, it will use the table to determine where the latest location of the page is and forward the request.

Before the latest update arrives, the daemon can forward the requests to incorrect daemons. In this case, the requests can be forwarded back-and-forth. However, we know that eventually the latest update packet will arrive to the original daemon and fix the problem.

The critical problem comes when the old update packet arrives after the latest update. Figure 5.12 shows the case when a single page is migrated from one daemon to another in a rather short time. Although rare, in these cases

¹So far, it simply stores the last 10 updates.

update packets can arrive in any order. If the old update (5.12-3) arrives after the latest update (5.12-5), the original owner can overwrite the correct location with an incorrect one. In order to avoid such cases, each update packet is attached with the logical clock [42] and compared against the last update of the page.

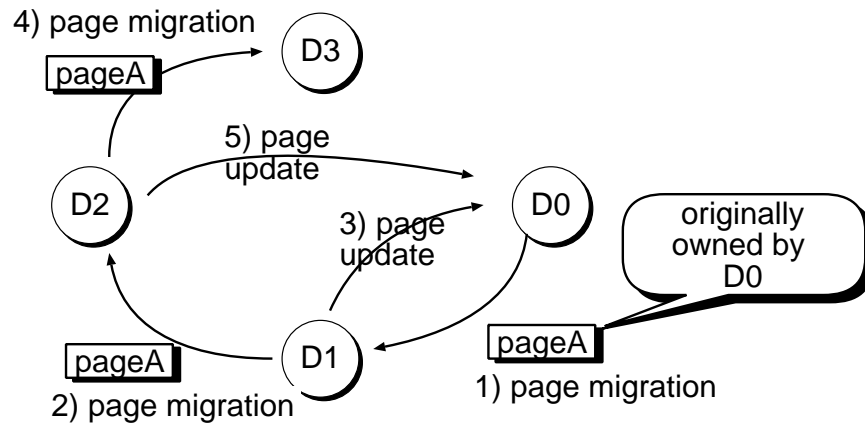


Figure 5.12: Page migrating one after another

5.2.6 Page Table Look aside Buffer

Just like regular TLB (Translation Lookaside Buffer) is used to improve the speed of finding the corresponding physical address, the daemon can prepare a table similar to this so that the extra step of always forwarding through the original owner can be skipped.

The idea is, whenever original owner forwards the request packet, it lets the sender know where it should have sent to. The sender daemon can use this information for future requests. The difference from TLB is that the location of the page can change all the time making the information stored in the table incorrect. In this case, a mechanism of later invalidating the entry is required.

This feature is left as part of our future work.

5.2.7 Avoiding Deadlock

Communications exchanged among daemons are all less than 100 bytes except when pages migrate. Multiple pages could be sent at once. Each physical page size is 8Kbytes in our Solaris machines. This increase in data size handled by the daemon requires us to rethink the iterative daemon design (section 3.1) of our system.

- While sending or receiving pages, the daemon cannot handle other incoming requests
- If the two daemons decide to migrate pages at the same time, send buffer could get full, blocking both side of the daemons resulting in a classic deadlock case.

When we look at the time spent for sending and receiving pages, the time spent on each function is quite different. System call *write* returns immediately after it is done copying the sending data to the local send buffer at kernel level. It does not guarantee anything about the receiving side. On the other hand, receiver knows how much data to expect, thus it will repeat the *read* system calls until it has received all the data sent from the sender (Fig 5.13).

From this observation, we introduce an extra thread dedicated to receiving pages. The main thread handles everything else. With this new design, the overheads of synchronizing the threads are minimized since each thread can run in parallel. And also the classic deadlock case is no longer a problem

since the receiving thread will keep on running even when the main thread is blocked by the full send buffer.

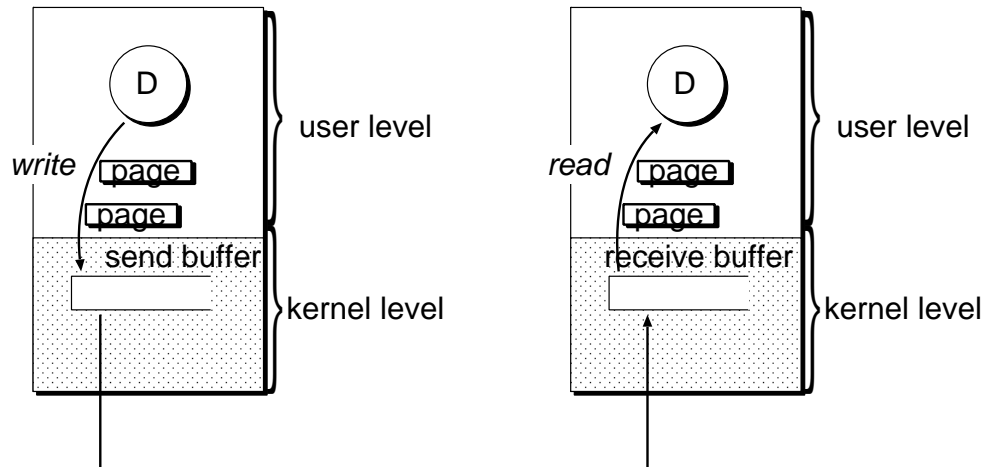


Figure 5.13: System calls *write* and *read*

5.2.8 Non-Migratable Page

Our migration library is written entirely at the user level and has no control over anything done at the kernel level (section 3.6.2). This raises an interesting question: What happens when a page containing a shared semaphore migrates to a remote daemon? It depends on the OS, but the result is undefined. To be safe, we provide a separate malloc, *spm_static_malloc*, which has the exact same format as *spm_malloc* except that whatever space allocated through this call will not be migrated.

We rely on the users to correctly call this function when combined with a system call (that saves some states at the kernel level).

5.3 Runtime Heuristic

Whenever an agent process tries to access some remote data, it makes a decision on whether to migrate itself or to remote read/write using the past migration pattern (section 4.3). Basically it is a greedy algorithm in the sense that the decision is made independent from the performance of local daemon or the behavior of the other agent processes.

With page migration, this is a completely different story as the page migration involves the daemon and any agent processes accessing the page(s) on both sides. Thus, a single daemon cannot make the decision by itself, and the heuristic involves a negotiation process between the two daemons.

Whenever a daemon receives a process migration or a remote read/write request, it will determine if the page migration would be a better choice (*do_page_request*). If this daemon decides to request a page migration, the receiving daemon would then examine the request and make the final decision on whether to migrate the page or deny the page migration request and simply handle the original agent process' request (*examine_page_req*). Figure 5.5 shows the basic steps on how the page migration request is handled.

Just like the heuristic by the agent process (section 4.3), making an incorrect decision on page migrations would only harm the overall speed of the application but not the correctness. This would give the daemons a way to aggressively migrate the pages and learn from errors.

5.3.1 do_page_request

Basically, we want our heuristic to detect when the process thrashing is happening. However, if it requires certain amount of sample for each page migration, the overhead will be too high. Instead, the system tries to find a pattern when process thrashing occurs and apply page migration the next time it sees that the same pattern is going to happen.

Each agent process keeps track of “several” past code addresses and memory addresses that caused process migration or remote read/write. Whenever it sends a request to a daemon, it also attaches the difference from the last memory address accessed by the same code address.

When daemon receives a request from the local agent process, it checks how frequently the requests are made from the same code address to addresses within “a small” distance from each other. (It does not have to be from the same agent process.) Once it hits a “certain” threshold, it will request a page migration instead of just forwarding the process request.

We use fixed value for the above quoted values, the number of past code addresses to record, range of memory access to check, and the threshold for starting page migration². It is left for a future research for varying these values at runtime based on the application behavior (chapter 8)

5.3.2 examine_page_req

One of our goals for the system design is that if the agent process stays on a single daemon, the speed of the agent process should be competitive with what the equivalent sequential C would have. We rely on the UNIX hardware

²Currently, 20 past code addresses information, range of 100 bytes, and threshold of 4

virtual address locking mechanism (section 3.2.3) to detect the remote pointer dereference and avoid any explicit pointer checking.

The drawback of our approach is that there are no means for knowing how often a certain virtual page is being accessed locally. Thus the daemon sending out the page has to make the initial decision without knowing how it would affect the local agent processes. We do not want to change our standard and add an overheads to each pointer dereference. Instead, we look at the time interval between synchronizing the page mapping with local agent processes and actually sending the page to the initiator daemon (section 5.2.2). Within this time, some local agent processes might access the page and be moved to the *waiting_pages* queue (Fig 5.11). This is an indirect measure of how much the page is being used locally. Now, the responder daemon can use this information, number of local agents moved to the queue, to finalize the page migration or to change its mind and keep the page after all.

Of course, we want to avoid changing its decision as much as possible since it involves all the local agent processes for synchronizing the page mappings. Thus, the daemon tries to filter out the request when it is first received before going on to the synchronizing page mappings phase. Although we haven't had this situation, we can filter out the requests when

- the same type of request has been denied before
- initiator's average cpu load is much higher than the responder's

5.4 Modified Malloc and Free

In *spm_malloc* (section 3.2.2), the library traverses the doubly-linked-list of holes to allocate a space. Obviously, when pages migrate, these linked lists can easily get broken and many dangling pointers can be left behind (Fig 5.14). (“Pointers” used in this section refers to the internal data structure employed in *spm_malloc* and not to the pointers used by the application programmers.) What makes the problem complicated is that we do not want any external data structures except for the page bitmaps.

In this section, we will go over the modified version of *spm_malloc* and *spm_free* that supports page migration with reasonable overheads (*cut and relink*). Then we extend our method for handling blocks bigger than virtual page size (*multi-page blocks*). Last, alternative way for *spm_free* (*lazy free*), is introduced and compared with our current method.

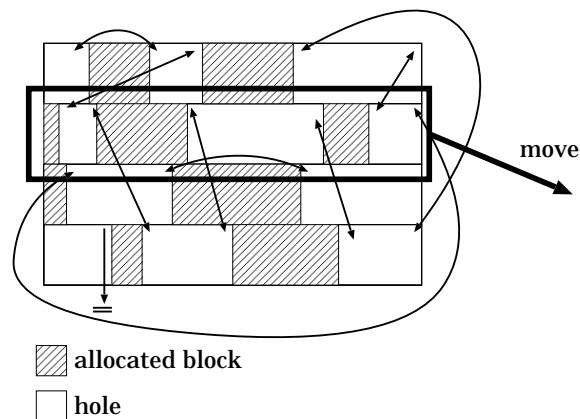


Figure 5.14: Moving a page inside the Global Shared Heap

5.4.1 Cut and Re-link

Our strategy restructures the linked list of holes on both sides of the daemons. On the sender's side, it will take out all the holes belonging to the pages being moved. On the receiver's side, all the holes are added to its linked list. In this way, the free block will be immediately available for reuse at the receiver's side.

When a page is moved, any pointers pointing out from the page or pointing into the page have to be updated. In our original implementation, the holes are connected in the order *spm_free* is called. In the extreme case, there could be $\#holes \times 4$ pointers (2 incoming and 2 outgoing) to update (Fig 5.14). It is unrealistic to update all these pointers, instead, the structuring of the linked list is modified so that the number of pointers to update is always fixed.

Figure 5.15 shows the case when the list of holes are connected sequentially. Note that it takes some extra work to find neighboring holes in *spm_free* [7]. Advantage of this ordering is that page migration only requires 2 pointers to be updated on the sender's side. But to find the head and tail of these holes still requires to traverse the linked list of holes.

Figure 5.16 shows modified version where holes are connected as a linked list of linked lists. Each page has a header which connects the pages that contain at least one hole. Any pages without a hole are skipped. To save the space for the page header, an allocated block smaller than the page size cannot straddle the page border. Unlike the previous case, list of holes can be randomly ordered within the page and pages can also be randomly ordered. When the pages migrate, it simply uses the header to detach itself.

Table 5.2 summarizes the methods. Basically, by ordering the holes, the

number of pointers to update is reduced. And by having a header for each page, the time to find the pointers is saved.

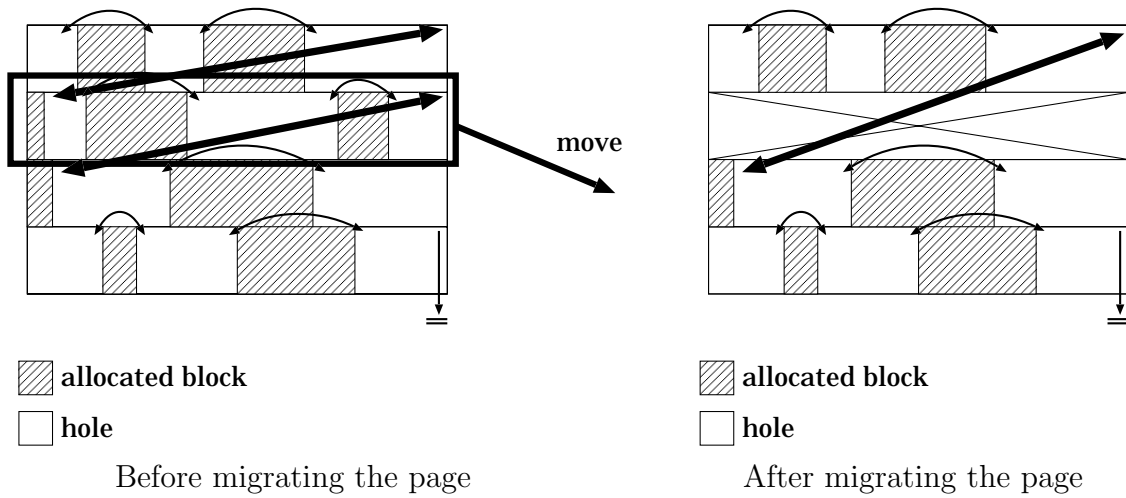


Figure 5.15: Sequentially sorted linked list

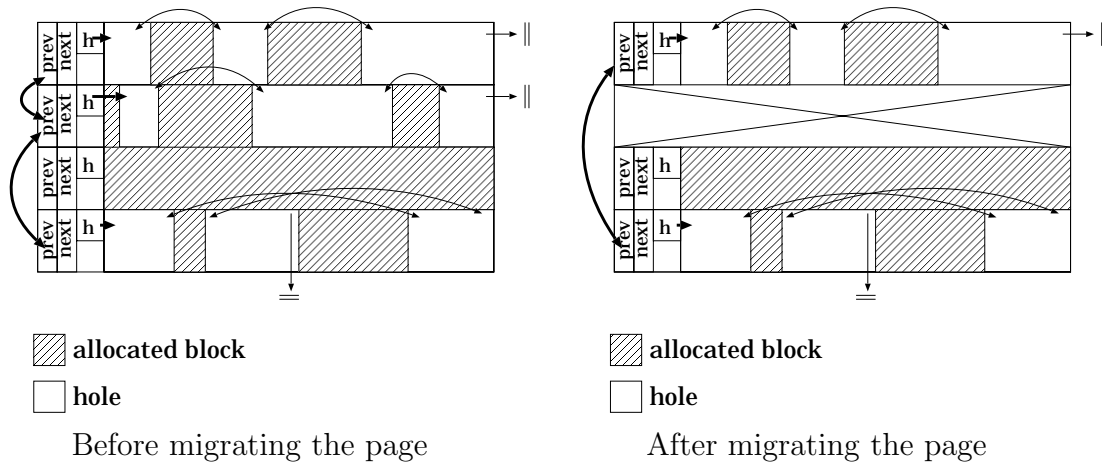


Figure 5.16: Linked List of Linked List

		Random Order Linked List	Sequential Order Linked List	Linked List of Linked List
spm_malloc		O(#holes)		O(#holes + #pages)
spm_free		O(1)	O(#adjacent used blocks)	O(1)
page	#pointers	O(#holes in page)	O(1)	
mig	finding pointers	O(#holes)	O(#holes in page) or O(#used blocks at the edges)	O(1)

Table 5.2: Overheads of *spm_malloc* and *spm_free*

5.4.2 Multi-page Block

In the previous section, we skipped the case when an allocated block is larger than a page size (we call it multi-page block) When the multi-page block is marked free by *spm_free*, we want that block to be immediately reusable. This requires the system to migrate the entire multi-page block, thus multiple pages, for the page migration.

Figure 5.17 shows how our system handles such cases. Initially, instead of having a page header for each empty page, a 5 page size hole is represented by a single page header (I). After the first allocation, this hole is split into two holes (II) to avoid small block straddling the page boundary. Now after the next allocation of multi-page block, a single page header represents three used pages(2-4) ³ When page migration request arrives for page 4, the daemon looks up the page bitmap and learns that it is part of the multi-page block and its header is at page 2 ⁴. For the immediate reusability for *spm_free*, the daemon

³User may choose not to use the remainder hole from the multi-page block.

⁴first occurrence of “L” from page 4 going up

changes the bitmap, acknowledges the processes, and migrates the entire three pages to remote host (IV).

Note that in our system, a multi-page block always starts at the head of the pages. Without it, we could have the end of one multi-page map and the start of another multi-page co-exist in a single page (Fig 5.18). If the bitmap is set to “L” daemon would think that there is a page header on that page. If set to “M”, the daemon could not be able to know that there are two multi-page blocks and would decide to migrate the entire four pages.

5.4.3 Lazy free

The advantage of migrating the entire multi-page block is that when the *spm_free* is called, the block becomes immediately available. On the other hand, the disadvantage is the overhead of sending the entire block when only part of the pages is being used.

In this section, we show the alternative solution by allowing a portion of the multi-page block to migrate but give up on the immediate reusability of *spm_free*. Basically, all the rules are followed from the previous section 5.4.2 except for migrating the entire multi-page block. Instead we allow migrating the individual pages separately. By not using the remainder hole from multi-page block, we guarantee that there are no other holes contained in any of the migrated pages, thus no pointer updates are needed.

Figure 5.19 shows an example of a single multi-page block distributed among three daemons. The pages can be accessed separately by the agent processes. The difference comes when *spm_free* is called at daemon 1 on the

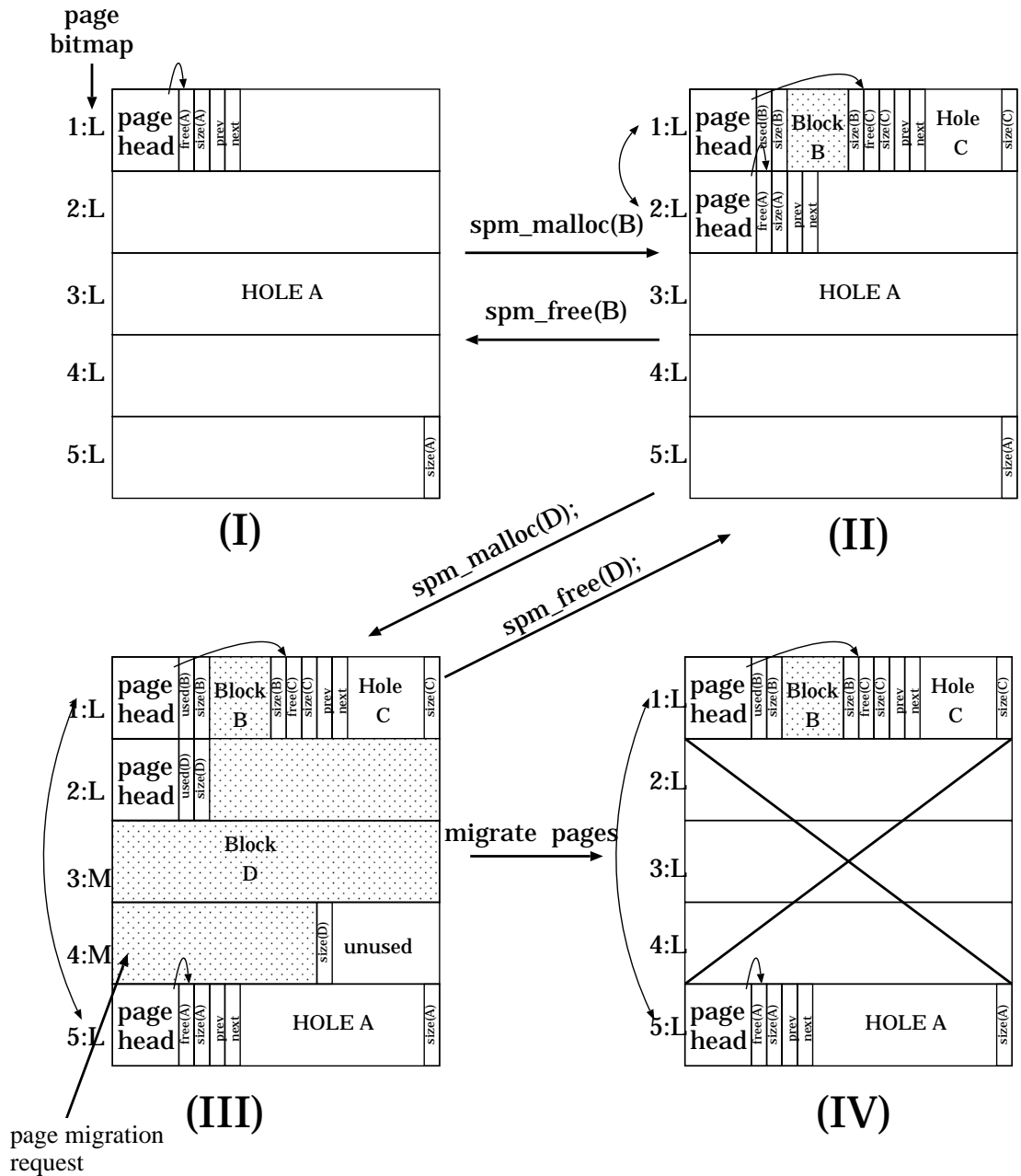


Figure 5.17: Allocating a block larger than a page

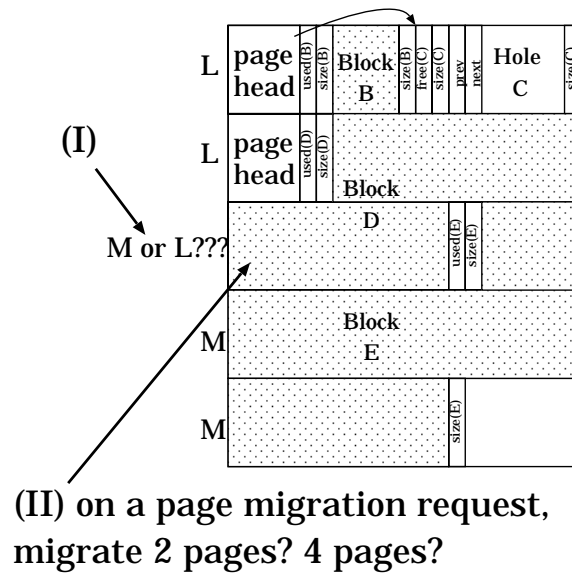


Figure 5.18: Problem of Multi-page blocks

multi-page block⁵. Daemon 1 sends a request to the original owner of the pages, in this case daemon 0, and asks to forward the request to the current owners. When the other daemons receive the requests, they simply disable the corresponding page(s) and acknowledge daemon 1⁶. Note that since the pages are marked free, the daemons can simply throw away the pages. What we are accomplishing through the acknowledgment process is to ensure that the future access to this previous multi-page block space would go through daemon 1. When daemon 1 receives all the acknowledgments, it can mark the block reusable and link the pages back for future *spm_malloc*.

⁵Just like UNIX *free* call, *spm_free* is always called with address pointing to the top of the block.

⁶It involves signaling the local agent processes to synchronize the page mapping.

5.4.4 Immediate *free* vs. Lazy *free*

As mentioned, there are pros and cons on both of the *free* methods. We cannot simply compare these without specifying which application will be used. It depends on how the multi-page block is accessed and how often the *spm_free* is called. Lazy free method becomes useful for applications involving intensive array-based computation. So far, our target application works well with immediate free methods. But the better strategy would be to let the runtime heuristic decide on which methods to use by looking at the past agent process behavior. This is left for our future work.

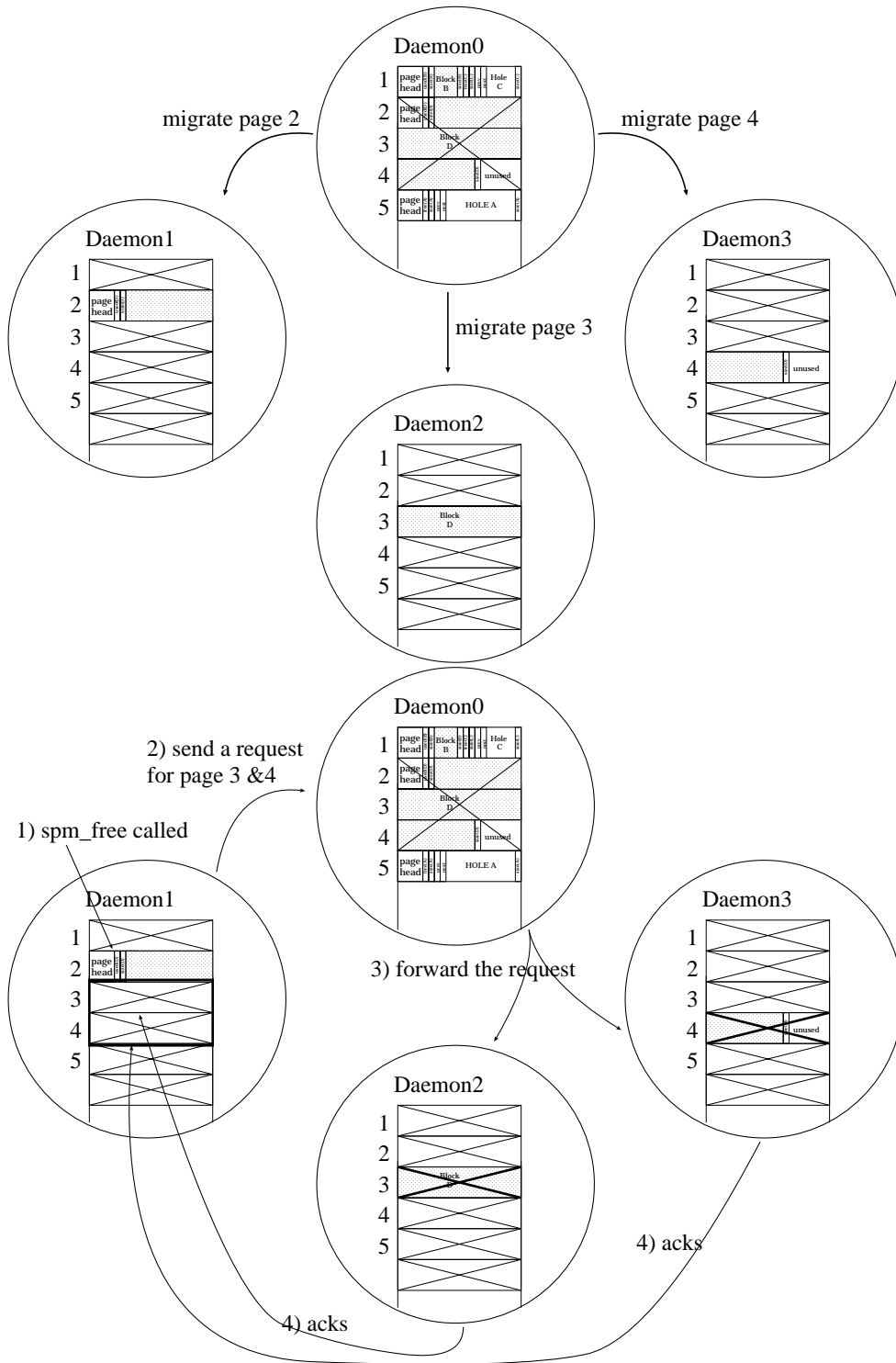


Figure 5.19: Example of Lazy Free

Chapter 6

Performance Evaluation

In this chapter, performance results of our system are collected. All the experiments were run on Sun-Blade-100 502 MHz 256 MB memory machines.

6.1 Overhead of Pointer Checking

One of our goals was to keep the high uni-processor performance as long as the process stays on the same daemon. We were willing to pay the cost when the actual migration is necessary. In this section, we evaluate the method for pointer checking.

6.1.1 Explicit Pointer Checking

As mentioned in section 2.3, some systems distinguish between local and global pointers. We pointed out that it is not always possible to determine if the pointer would be local or not at compile time. Here, we would like to observe the impact on using global pointers. Note that our system would not suffer

from this extra checking since it relies on hardware virtual address locking mechanism.

Table 6.1 shows the time it took to traverse a link list of 600,000 nodes for 2000 times. We intentionally made the linked list short in order to avoid paging. First column shows the pure sequential C code speed. Second column inserts a range check before each pointer dereference. Third, pointer is changed to a structure so that it hold the memory address and the `daemon_id`. Again, before before each pointer dereference, it checks the `daemon_id` of the pointer.

We can observe that simple range checking can slow down the process up to 30%. Furthermore, having extra information associated with the pointer costs even more, we see the slow down of 78%. This comes from the extra overhead for each assignment.

	Original	Pointer Range Checking	Pointer id checking
Time [sec]	82.90	108.61	147.84

Table 6.1: Overhead of Explicit pointer checking. Traversing 600,000 nodes for 2000 iterations.

Using the hardware trap does not have any overhead unless there is a fault. So the runtime is same as the original: 82.90 seconds.

6.1.2 Signal Handler

Relying on the hardware virtual address locking does not come for free. When the remote pointer is dereferenced, the signal handler has to be activated and the snapshot of the register has to be taken (so that it can return to where the signal has occurred). In order to measure the time spent on signal handler, we intentionally caused segmentation fault, activated the signal handler,

incremented the program counter by 1 (to skip the statement that caused the segmentation fault) and repeated this process 1000 times (Fig 6.1). As shown in table 6.2, the overhead for SIGSEGV is significantly larger but much less than the cost of a hop¹. Thus, the overhead of using hardware virtual address locking for detecting remote pointer dereference is negligible if it is followed by any of the agent remote access methods (hop, remote read/write, and page migration). However, we can not use this hardware trap to keep track of local accesses.

	Accessing 1000 integers	1000 SEG FAULTS
Time [sec]	0.000017	0.043

Table 6.2: Overhead of Signal Handler: 1000 Segmentation Faults

```

1 void signal_handler(int sig, siginfo_t * p_siginfo, ucontext_t * ucp)
2 {
3     INCREMENT_PC(ucp); // skip one statement
4     return;
5 }
6 int main()
7 {
8     <set signal_handler for SIGSEGV>
9     get_timer(&st_timer);
10    for( i = 0; i < 1000; i++)
11    {
12        *((char*)i) = i; //segmentation fault
13    }
14    get_timer(&en_timer);
15 }

```

Figure 6.1: Code for causing Segmentation Fault

¹It depends on an application, but it usually takes about 1-2 ms.

6.2 TreeAlloc & TreeAdd

One of the typical dynamic structure is a tree structure. Fig 6.2 shows the balanced tree distributed into 4 hosts. Fig 6.3 shows how the tree is allocated (`treeAlloc`) and how it is traversed (`treeAdd`) in our system. Same as the previous linked list example, allocation code is modified but the traversal code is untouched from the pure C code.

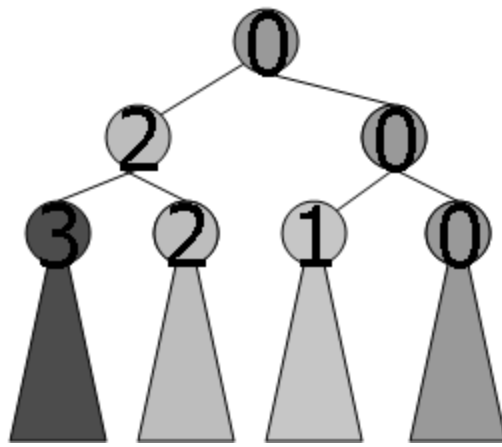


Figure 6.2: Tree Structure

Table 6.3 shows the result of the execution. First row is the result from a process sequentially traversing the entire tree. The result shows the overhead of paging and how our system can be used to avoid and achieve super-linear speedup. Once the paging is gone, there is no speedup by increasing number of CPUs for sequential code (4 to 8 hosts for 1 active process).

Second to fourth rows show the results obtained by task parallelism, using UNIX *fork* call. Each forked process traverses part of the tree and is synchronized at the original forked node. Fig 6.4 shows the source code. It create the semaphore inside the global heap so that both *sem_post* and *sem_wait* would

```

1 tree * treeAlloc(int level, int loc, int num_proc )
2 {
3     tree * t;
4     if( level == 0 ) return;
5     else
6     {
7         t = pm_malloc(lo, size);
8         t->val = assignValue();
9         t->left = treeAlloc(level-1, lo + num_proc/2, num_proc/2);
10        t->right = treeAlloc(level-1, lo, num_proc/2);
11        return t;
12    }
13 }

```

Tree Allocation

```

1 main() {
2     tree * head; int result;
3     head = treeAlloc(24,0,4);
4     result = treeAdd(head);
5 }
6
7 int treeAdd(tree *t) {
8     if( t == NULL ) return 0;
9     else
10        return t->val + treeAdd(t->left) + treeAdd(t->right);
11 }

```

Tree traversal

Figure 6.3: TreeAlloc and TreeAdd

```

1 int Parallel_treeAdd(tree *t, height) {
2     my_sem *ptr_sem;
3     if( t == NULL ) return 0; if (height <= 0) return treeAdd(t);
4     ptr_sem = pm_malloc(local_id, sizeof(my_sem));
5     if( fork() ) { //parent
6         tleft = t->val + Parallel_treeAdd(t->left,height-1);
7         sem_wait(ptr_sem->semaphore);
8         tmp_ptr->val += tleft;
9     }
10    else { //child
11        ptr_sem->val = Parallel_treeAdd(t->right, height-1);
12        sem_post(ptr_sem->semaphore);
13        exit();
14    }
15    return ptr_sem->val;
16 }

```

Figure 6.4: ParallelTreeAdd

migrate the process to its origin and then resume the semaphore procedure. As long as we have enough CPUs to utilize the processes, there is a speedup.

24 levels, $2^{25} - 1$ nodes (400Mbytes)	Number of Daemons (Hosts)				
		1	2	4	8
Number of	1	584.17	245.08	24.00	24.15
Active Processes	2	670.51	131.67	13.94	12.69
	4	710.14	114.64	7.75	7.71
	8	2326.95	118.31	9.07	5.93

Table 6.3: TreeAdd time with TCP/IP process migration and ghost processes

Table 6.4 shows the performance when process migration was done through NFS (section 3.3.2) and there were no ghost processes (section 3.5.2). It was performed on a similar environment: Solaris 2.8 UltraSparc 502 MHz 256 MB Memory. You can observe that because of the huge migration overhead,

it actually slowed down the performance when increasing from 4 hosts to 8 hosts. This shows how the two enhancements helped speed up the system.

24 levels, $2^{25} - 1$ nodes (400Mbytes)		Number of Daemons (Hosts)			
		1	2	4	8
Number of Active Processes	1	459.5	330.8	44.0	61.54
	2	642.0	184.7	23.68	43.3
	4	718.5	114.8	23.3	37.84

Table 6.4: TreeAdd time with NFS process migration and without ghost processes

6.3 Remote Read/Write

To test the performance of remote read/write, we used a simple linked list (Fig 6.5) to measure its performance. For every k nodes, it crosses the daemon boundary. The problem size is $k \times 50$ nodes, thus varies as the k varies. Figures 6.6 and 6.7 show the expression we tested. Our system will automatically select the remote read/write over migrating back and forth.

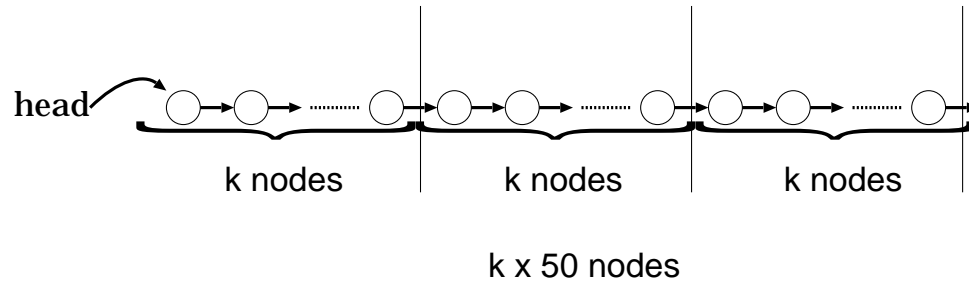


Figure 6.5: Testing remote read/write using linked list

Figure 6.8 and 6.9 show the performance benefit while varying the problem size with fixed number of hops. When k is small, most of the time is spent on communication thus the speed up you gain from the remote read/write is

```

1 test_remotewrite()
2 {
3   get_timer(&st_timer);
4   prev = head;
5   cur = head->next;
6   while( cur != NULL )
7   {
8     prev->val = cur->val + 10;
9     prev = cur;
10    cur = cur->next;
11  }
12
13  get_timer(&en_timer);
14 }

```

Figure 6.6: Program that can benefit from remote write

```

1 test_remoteread()
2 {
3   get_timer(&st_timer);
4   prev = head;
5   cur = head->next;
6   while( cur != NULL )
7   {
8     cur->val = cur->val + prev->val
9               + cur->val;
10    prev = cur;
11    cur = cur->next;
12  }
13  get_timer(&en_timer);
14 }

```

Figure 6.7: Program that can benefit from remote read

significant. Obviously, as the k grows, more time is spent on computation and the speed up decreases. But in any cases, we can observe a significant gain by the system automatically selecting remote read/write over process migration.

Even though the expression we tested for remote read and write are different, actual computation times were almost equivalent. Figure 6.10 shows the performance when two results are combined in the same plot for comparison. We can observe that remote write gives a slightly better performance over remote read. Actual gain increases when the problem size increases. This is because remote write can overlap the communication with computation, whereas remote reads always have to wait for the acknowledgment. The difference could become bigger as the daemon's load or the network load gets heavier.

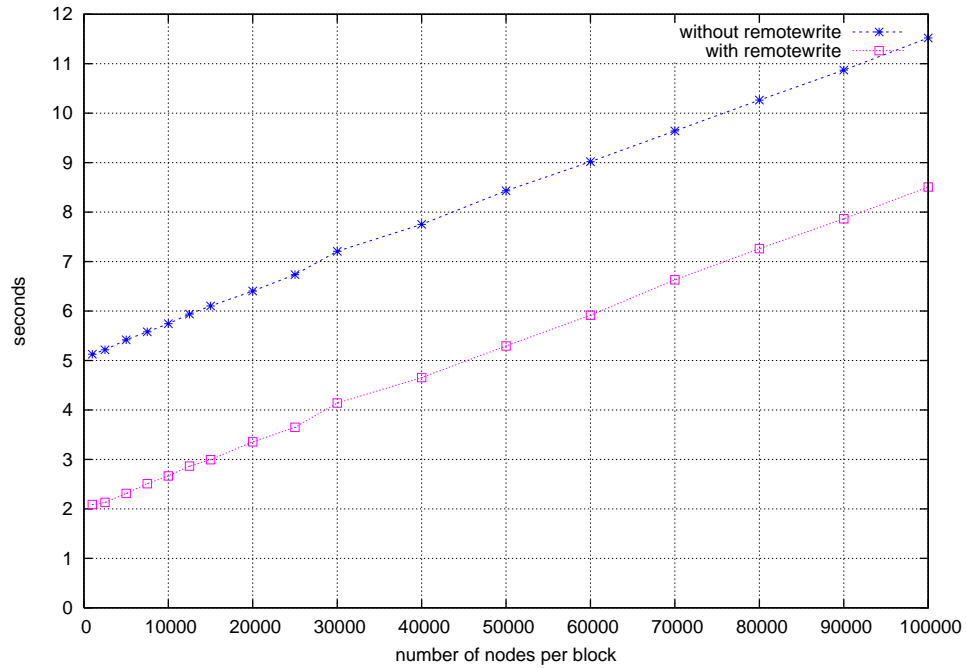


Figure 6.8: Result from remote write

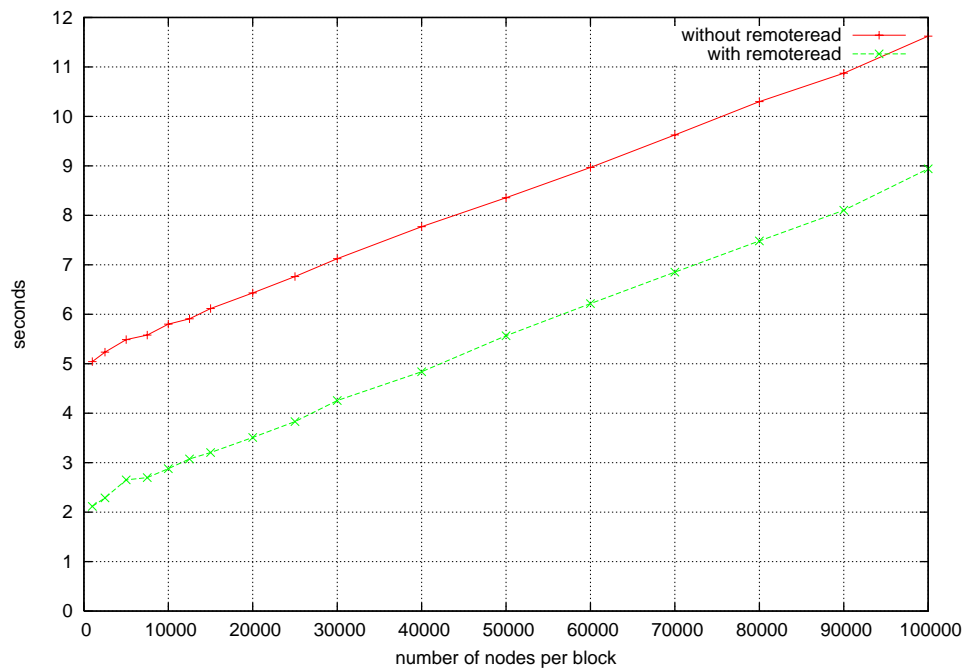


Figure 6.9: Result from remote read

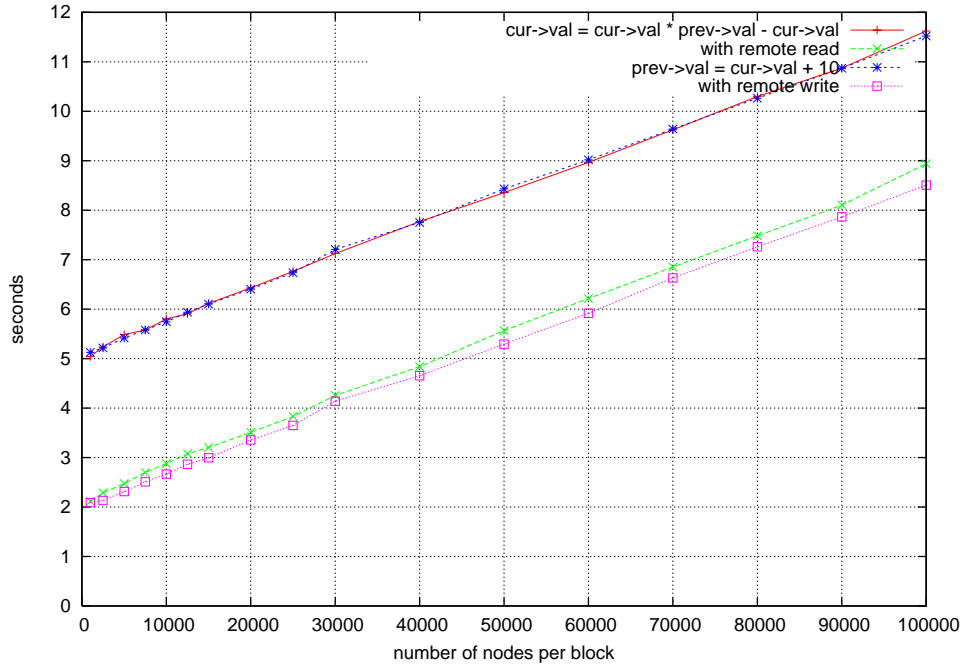


Figure 6.10: Result from remote read/write

6.4 Bitonic Sort

Bitonic Sort [5] is a sorting algorithm initially developed for a network of comparators. We tested the adaptive bitonic algorithm [8] which was targeted for general purpose shared-memory machine. It takes advantage of a tree structure to reduce the total number of comparisons and copying.

We modified the code so that once the sub-tree size become less than a certain threshold, we simply call a quicksort instead (Fig 6.11). Initially, the data is distributed equally to each host. Once the sorting starts, pointer swapping would occur at the intermediate nodes, and page migrations on the leaf array nodes. Thus, the distribution of the data would be quite random.

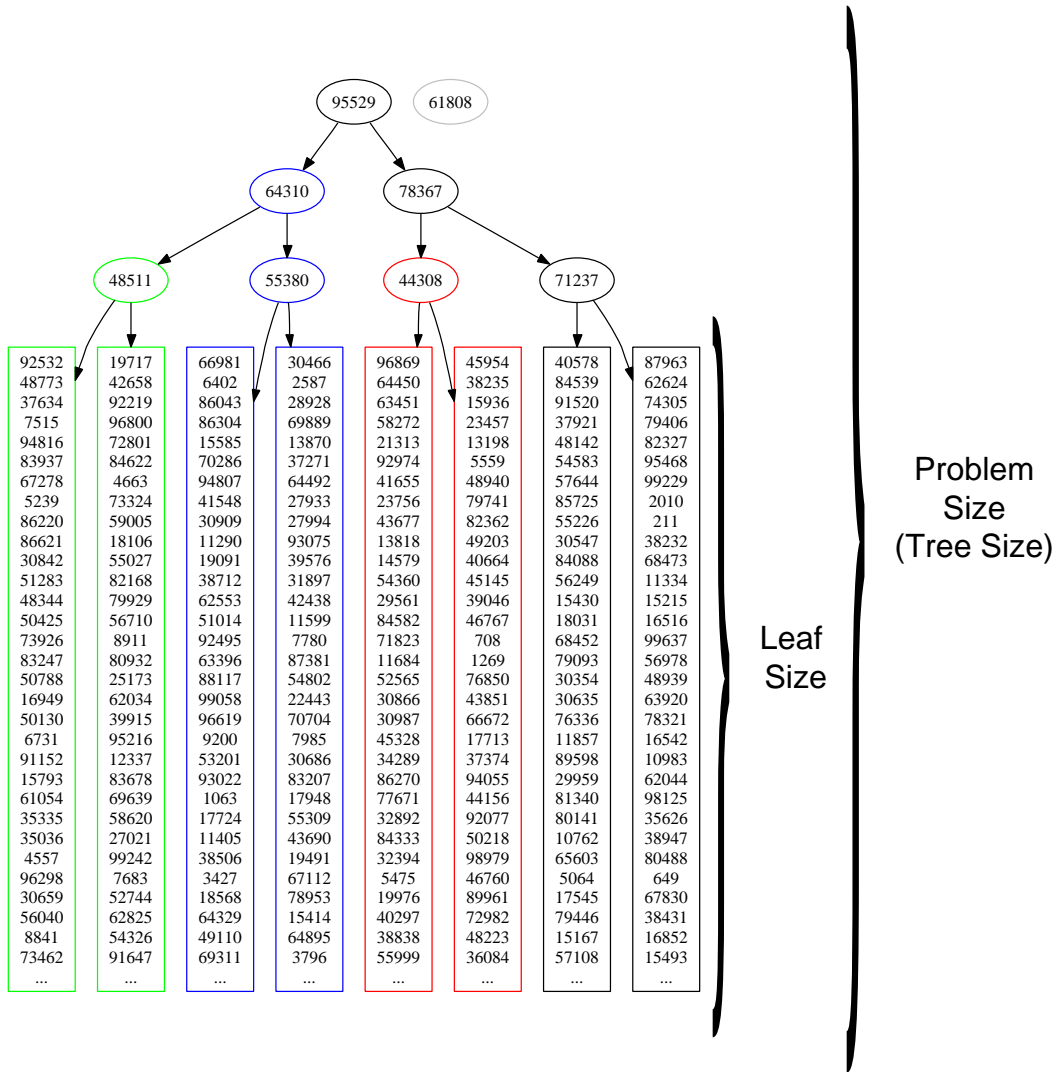


Figure 6.11: Bitonic Sort Initial Data Mapping

6.4.1 Advantage of Page Migration

In this section, we first ran a small problem on Sun-Fire-V240 2G RAM machines² to see the advantage of page migration (without paging slowing down at runtime). Table 6.5 shows bitonic sort on 2 hosts with tree size 2^{23} and leaf size 2^{17} . Sequential Code ran in 9.27 sec. We can observe that without remote read/write and page migration, the program takes extremely long time, over 6 hours. This is because, bitonic sort involves many leaf-to-leaf(array) comparisons and each element access can correspond to one process migration. Remote read/write gives ten-fold speedup from only-process-migration-result, but still shows over 2 orders of magnitudes slower than the sequential code. Now, with page migration, we see that they run only 13-16% slower than the sequential code. We see the slowdown because the problem size was small enough to avoid the paging. In the next section, we show that when the sequential code experiences paging, DSC would give an actual speedup.

Note that this experiment was performed by passing different options to the runtime systems and sequential code was only modified for the allocation, *spm_malloc*.

	w/o Remote Read/Write	w/ Remote Read/Write
w/o Page Migration	23453.10(sec)	2342.12(sec)
w/ Page Migration	10.75(sec)	10.46(sec)

Table 6.5: Bitonic Sort Tree Size 2^{23} Leaf Size 2^{17} running on 2 Hosts

²Other bitonic sort experiments were done in Sun-Blade-100 502 MHz 256 MB memory machines.

6.4.2 Distributed Sequential Code

Figure 6.12 shows the performance when only one agent process traverses the entire tree. It is tested on 1 and 4 hosts. Figure 6.13 shows the case when tree size is fixed to 2^{27} . We can see that distributed sequential code gives up to 100% speedup. But unlike the treeAdd result (table 6.3), the gain of DSC is much smaller. This is because treeAdd only has $n - 1$ migration points in the tree, whereas bitonic sort involves migration at any non-leaf nodes in the tree, thus the communication granularity becomes much finer. By pointer swapping used in the algorithm, non-leaf nodes are almost distributed randomly.

6.4.3 Parallel Code

By fixing the problem size and leaf size to 2^{27} and 2^{15} respectively, we now experiment with parallelizing the bitonic sort. Table 6.6 shows that we were able to obtain a speedup of 4.9 using 8 hosts (346.61 over 1710.45). It is quite encouraging considering that bitonic sort involves a complicate communication pattern.

Problem Size 2^{27} nodes Leaf Size 2^{15}		Number of Daemons (Hosts)			
		1	2	4	8
Number of Active Processes	1	1710.45	1118.61	1068.09	1182.18
	2		608.71	611.65	641.02
	4		613.00	402.03	422.67
	8		686.62	432.53	346.61

Table 6.6: Bitonic Sort: Parallel code

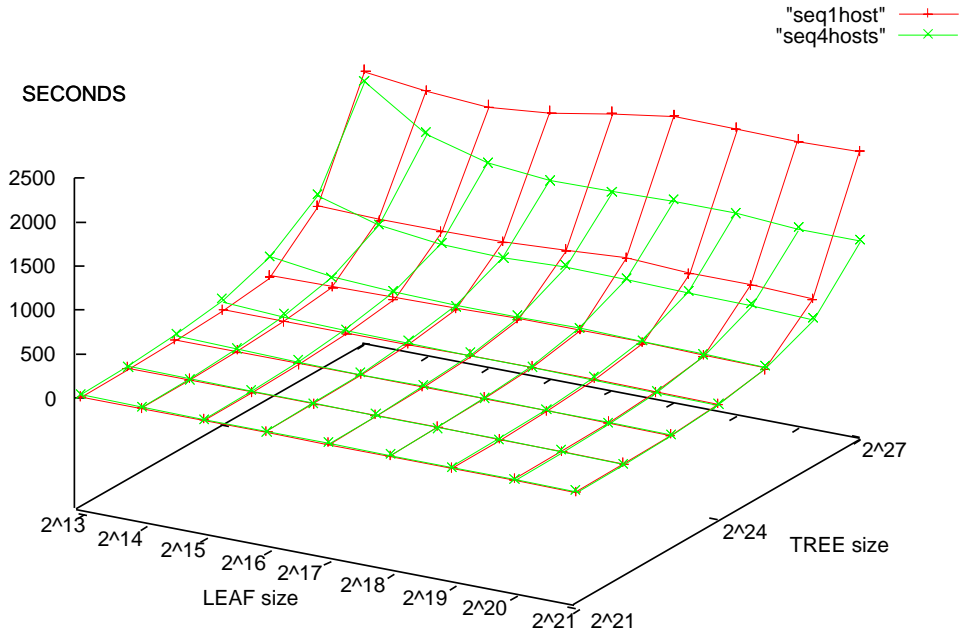


Figure 6.12: Bitonic Sort: Sequential Code and Distributed Sequential Code (4hosts)

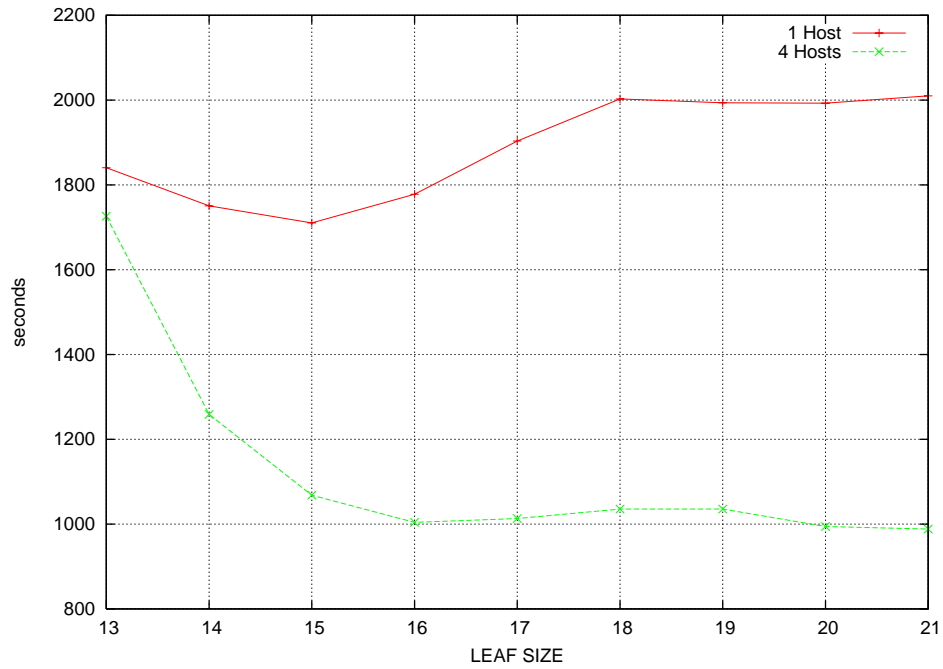


Figure 6.13: Bitonic Sort: Sequential Code and Distributed Sequential Code (4hosts) Tree Size = 2^{27} nodes

Chapter 7

Related Research & Future Work

Distributed computing has evolved over decades of research. If we focus on regular array-based computing, there have been automatic parallelizing compiler (Polaris [10], SUIF [29]), which take a pure sequential code and produce parallel codes. Initially the target platform was Uniform Memory Access machines, and later it was extended to run on distributed memory machines. Although limited, they perform well on certain applications but never became the main-stream of distributed computing.

Two main programming models people often use are message passing [10] and parallel programming language, which let the users specify how they want to parallelize their codes (HPF [32], Titanium [69], UPC [16], Fortran Coarray [55], NavP [57]). Users are involved in decisions on both the location of the data and which blocks to parallelize. Compilers often take care of the low level message passing and also optimize the amount of data transferred based on

the knowledge at compile-time.

Another approach is using the runtime library for supporting global data structures (global array [52], and KELP [21]). Depending on the level of complexity, users can choose which approach to use.

On computing based on dynamic data structures, the story is a little different. For over two decades, pointer analysis has been developed and used for automatic parallelizing compilers ([27], [31], [18]). However, because of the indeterministic nature of dynamic data structures, it is harder for the compiler to determine independent instructions. As a compromise, some suggest to require specific kind of programming styles [33], or introduce a pointer type qualifier ADDS [36], APT [37]. In all the cases, they assume and run on shared memory systems. It is still unclear on how to express the dynamic data structure distribution in a parallel language so that compiler has enough information to optimize at compile-time.

Thus the interest for runtime library that handles dynamic data structures on the fly come to play. These include Olden [59], MCRL [35], and our Spontaneous Process Migration system. The systems could directly be used by the users or they could be a target language for the automatic parallelizing compiler and parallel languages.

In this chapter, we compare our Spontaneous Process Migration system with three areas of research: process migration, distributed dynamic data structure, and automatic selection on data and process migration.

7.1 Process Migration for Locality of Access

The idea of process migration appeared in the early 1980s in the LOCUS Distributed Operating System [66]. Back then, most of the process migration systems’ goal was to load balance by utilizing the idle hosts (MOSIX [4], SPRITE [20], Condor [46], UPVM [14]). Migrations were mainly passive, initiated by high machine loads or by system administration purpose where processes are moved in order to reboot or reinstall certain hosts.

Later, as alternative to DSM (distributed shared memory), process migration was extended by hopping to remote data for locality of access (MESSENGERS [23], Nomadic Thread [40] [39], Orca [3], Ariadne [47], Cid [54], Earth [30], Olden [59], MCRL [35], PM2 [2]). Migrations are decided autonomously by each computing entity. Our SPM system also belongs to this category. Depending on the target applications, these systems provide different form of migrations. In Cid and Orca, threads can fork to different nodes, but threads will not migrate during computation. MESSENGERS allows full migration at user-inserted hop statement. Earth provides RPC-like function call, where user can specify where to execute the function. In MCRL systems, the process migrates when it enters a remote *region*, a programmer-defined memory area. Olden allows the program to migrate in the middle of the statement. Table 7.1 shows when the migration point is decided and also the migration types.

As described in section 2.1, our system implements “Autonomous Implicit Strong Migration” to handle shared dynamic data structures. This migration policy is similar to PM2 implementation with *MIGRATE_THREAD* protocol. We performs full migration rather than a partial migration to provide better transparency to the user. The system allows the program to pass in any

references of local, static, or shared variables and avoids the need to migrate back when returning from the current stack. Difference between PM2 is that our system allows dynamic allocation of shared data and provides the protocol which let both data and processes migrate by the runtime heuristic.

Alternative policy, that fixes the migration points at compile time, has a definite benefit from optimization point of view, but as discussed in section 1.2, it is an ill fit for handling distributed dynamic data structures.

For general background of process migration, we refer readers to Milojevic’s survey [48].

Migration Point	Remote Execution	Full Migration
Compile-time	Cid, Orca, Earth, MCRL	MESSENGERS, Ariadne
Compile and Run-time	Olden, Nomadic	SPM, PM2

Table 7.1: Migration Types

7.2 Distributed Dynamic Data Structure

Gupta first introduced a language and compiler support for SPMD execution of program with dynamic data structures [28]. It gives a distinct name for each allocated block based on the position in the structure. Users are asked to provide the naming strategy as well as the mapping to the physical hosts.

MCRL [35] run on CRL [41](C Region Library), software distributed shared memory system that creates region, a special memory block created by the user. It has a flexibility of creating dynamic data structures, but relies on the programmer to specify the code that might access the remote regions, by `rgn_start_read`, `rgn_end_read`, `rgn_start_write`, `rgn_end_write`.

Earth-C (Efficient Architecture for Running THreads), a parallel dialect of C, is designed to allocate dynamic memory local to each thread [30]. Unless the program explicitly moves the locus of computation, all the implicit references to the remote memory are pulled just like DSM.

Olden, which strongly influenced our work, provides an API that let the programmer specify where to allocate a new block. The program can migrate at any point.

What differentiate our SPM from these systems is the fact that we use flat memory model, and rely on hardware virtual address page checking to detect the remote memory access. In other systems, the compiler inserts explicit checks for any potential remote memory references to detect at runtime. Difference in speed is significant as shown in section 6.1.1. To avoid this overhead, other systems rely on the users and compilers to find as many local accesses as possible to avoid the extra checking. Although this sounds reasonable, this local-remote pointer distinction prevents the system from changing the owner of the data (pages) at runtime. We have shown an example when this feature is necessary even when data is placed carefully (section 6.4).

7.3 Selecting Data and Process Migration

Although only a few systems support distributed dynamic data structures, there are couple of systems that provide thread/process migrations on top of distributed shared memory.

The question is how the system makes the decision between migrating the process and pulling/pushing the data. In Earth and Cid, locus of computation

does not move unless explicitly stated at a function/fork call. In PM2, user can choose from different types of consistency model, but only has a single policy for migration; always migrate for remote pointer dereference.

Olden [59], MCRL [35], and Millipede [61] have a protocol for making a choice between process migration and data migration. In Olden, all the pointer dereferences will be analyzed and determined at compile time whether to migrate or simply cache the data. The compiler uses the quantified hints provided from the programmer which tells how likely the pointer is going to be remote. In MCRL, the programmer has to explicitly initiate a *rgn_start_read* or *rgn_start_write* to access the shared region. Then at runtime, it uses a simple heuristic from the history of remote read and write to determine whether to migrate or to cache.

Ideal as it sounds, both automatic selection methods have some weaknesses. In Olden, quantified hints and the actual data mappings have strong connections and both values always have to be changed together. Also, it is hard even for the programmer to know how the dynamic data structures will be distributed when it depends on the input. In MCRL, heuristic of always pulling at consecutive remote read means it will pull the entire dynamic data structure if one process traverses the pointers.

Millipede implements RAHM (remote access histories mechanism) that keeps track of a memory access pattern and tries to redistribute the threads accordingly. Besides from unbalanced loads or rebooting hosts, thread migration only happens when two pages are accessed repeatedly. Unfortunately, this approach is unfit for our target application which traverses the large data structures. Moreover, by only collecting page access histories, it would require

the system to collect samples for each page before making the correct decision.

Our system splits the responsibility and assign remote read/write to the agent process and page migration to the daemons. This separation let the agent process make its decision locally and let the daemon handle the global balancing. Also by collecting the PC (program counter) information, our system can benefit from the application's repetitive pattern. It automatically selects the migration type when accessing a new remote data. This is important when the agent processes traverse thousands of pages.

Chapter 8

Conclusions

For decades, process and thread migrations have been used as a new method for distributed computing. Often these systems are very flexible and they can be applied to different kinds of applications. In this thesis, we took a completely opposite approach. Our Spontaneous Process Migration system is tuned specially for running applications with heavy use of dynamic data structures. We believe the strength of process migration, ability to move locus of computation following the data at runtime, is especially useful for this type of application.

Our system gives the programmer the tools necessary to create pointer-based distributed data structures. Any reference pointing to the memory of a remote host causes the process to automatically and fully transparently migrate to the remote host.

We have demonstrated that distributing pointer-based data structure can lead to dramatic improvements in performance for both sequential and parallel programs. At the same time, we have shown some of the short-comings with

the migrate-only design:

- process migrating for the sole purpose of updating/reading one value
- process repeatedly migrating back and forth

and introduced two major runtime enhancements: remote read/write and page migration. Remote read/write let the process update a remote single data without migrating. Page migration moves the virtual page(s) instead of migrating the process. These enhancements are transparent to the users such that the runtime system tries to detect the pattern of the application and apply these enhancements using some heuristics.

Following are the future research directions that we believe will further improve our SPM system.

Saving address space We rely on the hardware trap to detect remote pointer dereference. One of the limitations of this approach is that whole problem size is bounded by the virtual address space size. Which is 2^{32} bytes for 32 bits OS. To overcome this limit, one option is to use pointer swizzling (at fault time) [68]. It is an address translation mechanism that can store large (linked) data structures, bigger than whole virtual address space, into a disk. When the data on disk is accessed, the entire page is mapped to the virtual address space, and future access comes for free.

The only problem for using this technique is that when a page is mapped to a virtual address space, all the pointers within that page have to be updated. However, since C is not a strong typed language, no compiler

or application can reliably keep track of all the pointers used in the data structures.

64 bits addressing Another obvious option is to run the system on a larger virtual address space, which supports 64 bits addressing. Note that our implementation reserves the virtual address spaces but does not map to any physical memory unless they are being used. This will let us write application in significantly larger scale.

Data mapping As mentioned in chapter 5, even though our runtime system re-distributes the data based on the application's access pattern, we still rely on the users for the initial data mappings. Once allocated, our system can only move the data by pages (or single data update for remote read/write). If the data is placed randomly, our runtime system will not be able to adapt to the access pattern, and this will easily become the bottleneck.

This problem is very similar to a caching problem. Chilimbi [17] introduced *ccmalloc* that takes an additional pointer parameter that points to an element that is likely to be accessed. In *spm_malloc*, we ask for host number, but this pointer information will be useful for deciding which page to allocate the space within that host. Also, Chilimbi introduces *ccmorph* that traverses the dynamic data structures at runtime and re-distributed the data elements. Users have to provide the traversal function. This is interesting in theory, but this would only work if there is no pointer pointing to the internal data elements.

Another interesting area is obtaining finer granularity but keeping the

hardware trap mechanism. In Multiview [38], it basically split the physical page into couple of sub-pages, and map multiple virtual pages into one physical page. Each virtual page is responsible for portion of the page. This does waste more virtual address space, but physical memory is not being wasted. With finer granularity of data to move around, runtime system will be able to adapt better.

Load balancing heuristic As described in section 4.3 and 5.3, our system has many conditional variables, sizes, and thresholds that are fixed at this time. It works well in our test applications, but most likely, these values need to be tweaked for some applications. It does not make much sense for the users to be involved in this process. (No transparency.)

Instead, it will be interesting to change these values over the runtime. For example, if we see too many page migrations, then we can change the condition stricter to have less. Also, runtime system may measure the network latency and bandwidth at start up in order to have a rough estimate on how long it would take for each migration.

Debugger With implicit process migration in place, it could be very hard for the users to track where the processes are. Tracing (section 3.5.1) would help, but when users need control at runtime, we need to provide a debugger with single UI that can trace the process hop among multiple nodes.

Bibliography

- [1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.
- [2] Gabriel Antoniu and Luc Bougé. DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols. *Lecture Notes in Computer Science*, 2026:55, 2001.
- [3] Heri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: a language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992.
- [4] Amnon Barak and Oren La’adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4–5):361–372, 1998.
- [5] K. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computing Conference*, 1968.
- [6] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory mul-

- tiprocessors. Technical Report CMU-CS-91-170, Pittsburgh, PA (USA), 1991.
- [7] Lubomir F. Bic and Alan C. Shaw. *Operating Systems Principles*. Prentice Hall, 2003.
- [8] Gianfranco Bilardi and Alexandru Nicolau. Adaptive bitonic sorting: an optimal parallel algorithm for shared-memory machines. *SIAM J. Comput.*, 18(2):216–228, 1989.
- [9] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.
- [10] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel programming with Polaris. *Computer*, 29(12):78–82, 1996.
- [11] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995.
- [12] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. In *2nd Workshop on Hardware/Software Support for High Performance Scientific and Engineering Computing, SHPSEC-PACT03*, 2003.
- [13] John B. Carter. Design of the Munin distributed shared memory system. *Journal of Parallel and Distributed Computing*, 29(2):219–227, 1995.

- [14] Jeremy Casas, Ravi B. Konuru, Steve W. Otto, Robert Prouty, and Jonathan Walpole. Adaptive load migration systems for PVM. In *Supercomputing*, pages 390–399, 1994.
- [15] Kasidit Chanchio and Xian-He Sun. Data collection and restoration for heterogeneous process migration. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, page 51. IEEE Computer Society, 2001.
- [16] W. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the berkeley UPC compiler. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS'03)*, June 2003.
- [17] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Making pointer-based data structures cache conscious. *Computer*, 33(12):67–75, 2000.
- [18] Francisco Corbera, Rafael Asenjo, and Emilio L. Zapata. New shape analysis techniques for automatic parallelization of C codes. In *International Conference on Supercomputing*, pages 220–227, 1999.
- [19] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with “readers” and “writers”. *Commun. ACM*, 14(10):667–668, 1971.
- [20] Fred Douglass and John K. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

- [21] Stephen J. Fink and Scott B. Baden. Run-time support for multi-tier programming of block-structured applications on SMP clusters. In *ISCOPE*, pages 1–8, 1997.
- [22] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufman, San Francisco, 1999.
- [23] Munehiro Fukuda, Lubomir F. Bic, Michael B. Dillencourt, and Jason M. Cahill. Messages versus messengers in distributed programming. *Journal of Parallel and Distributed Computing*, 57(2):188–211, 1999.
- [24] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.
- [25] Kouros Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta, and John L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *25 Years ISCA: Retrospectives and Reprints*, pages 376–387, 1998.
- [26] Carlo Ghezzi and Giovanni Vigna. Mobile code paradigms and technologies: A case study. In *Proceedings of the First International Workshop on Mobile Agents*, Berlin, Germany, 1997.
- [27] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Symposium on Principles of Programming Languages*, pages 121–133, 1998.

- [28] Rajiv Gupta. SPMD execution in the presence of dynamic data structures. In *Compiler Optimizations for Scalable Parallel Systems Languages*, pages 683–708, 2001.
- [29] Mary W. Hall, Jennifer-Ann M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, 1996.
- [30] L. J. Hendren, Xinan Tang, Yingchun Zhu, G. R. Gao, Xun Xue, Haiying Cai, and P. Ouellet. Compiling C for the EARTH multithreaded architecture. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, page 12. IEEE Computer Society, 1996.
- [31] L.J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 01(1):35–47, 1990.
- [32] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Houston, Tex., 1993.
- [33] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, UT, 2001.

- [34] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, 1992.
- [35] Wilson Cheng-Yi Hsieh, M. Frans Kaashoek, and William E. Weihl. Dynamic computation migration in DSM systems. In *Proceedings of Supercomputing '96*, November 1996.
- [36] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. Abstract description of pointer data structures: an approach for improving the analysis and optimization of imperative programs. *ACM Lett. Program. Lang. Syst.*, 1(3):243–260, 1992.
- [37] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 218–229, 1994.
- [38] A. Itzkovitz and A. Schuster. Multiview and millipage — fine-grain sharing in page-based DSMs. In *3rd Symp. on Operating Systems Design and Implementation (OSDI)*, New Orleans, Feb 1999.
- [39] Stephen Jenks. Multithreading and thread migration using MPI and Myrinet. In *Proceedings of Parallel and Distributed Computing and Systems*, 2004.
- [40] Stephen Jenks and Jean-Luc Gaudiot. Exploiting locality and tolerating remote memory access latency using thread migration. *Int. J. Parallel Program.*, 25(4):281–304, 1997.

- [41] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, volume 29, pages 213–226, 1995.
- [42] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
- [43] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.
- [44] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 229–239, New York, NY, 1989. ACM Press.
- [45] Ben Liblit and Alexander Aiken. Type systems for distributed data structures. In *Symposium on Principles of Programming Languages*, pages 199–213, 2000.
- [46] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.
- [47] Edward Mascarenhas and Vernon Rego. Ariadne: Architecture of a portable threads system supporting thread migration. *Software - Practice and Experience*, 26(3):327–356, 1996.

- [48] Dejan S. Milojevic, Fred Douglass, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Comput. Surv.*, 32(3):241–299, 2000.
- [49] Message Passing Interface Forum MPIF. MPI: A Message-Passing Interface Standard. Technical Report, University of Tennessee, Knoxville, 1995.
- [50] Message Passing Interface Forum MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, 1996.
- [51] Sun Developer Network. Java remote method invocation - distributed computing for java. Technical report. <http://java.sun.com/products/jdk/rmi>.
- [52] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.
- [53] Rishiyur S. Nikhil. Cid : A parallel, shared-memory c for distributed-memory machines. In *7th International Workshop on Languages and Compilers for Parallel Computing*, pages 376–390, Ithaca, NY, August 1994. Springer-Verlag.
- [54] Rishiyur S. Nikhil. Parallel symbolic computing in Cid. In *Proceedings of the International Workshop on Parallel Symbolic Languages and Systems*, pages 217–242. Springer-Verlag, 1996.

- [55] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.
- [56] Lei Pan, Lubomir F. Bic, Michael B. Dillencourt, and Ming Kin Lai. NavP versus SPMD: Two views of distributed computation. In *Proceedings, Int’l Conf. on Parallel and Distributed Computing and Systems (PDCS 2003)*, November 2003.
- [57] Lei Pan, Wenhui Zhang, Arthur Asuncion, Ming Kin Lai, Michael B. Dillencourt, Lubomir F. Bic, and Laurence T. Yang. Toward incremental parallelization using navigational programming. *IEICE - Trans. Inf. Syst.*, E89-D(2):390–398, 2006.
- [58] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. *Distributed Shared Memory: Concepts and Systems*, volume Summer 1996. IEEE Computer Society Press, 1996.
- [59] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.
- [60] Yair Sade, Shmuel Sagiv, and Ran Shaham. Optimizing C multithreaded memory management using thread-local storage. In *Lecture Notes in Computer Science*, volume 3443, pages 137–155, 2005.
- [61] A. Schuster and L. Shalev. Using remote access histories for thread scheduling in distributed shared memory systems. In *In 12th Intl. Symp. on Distributed Computing*, pages 347–362, Andros, September 1998.

- [62] Shamik D. Sharma, Ravi Ponnusamy, Bongki Moon, Yuan-Shin Hwang, Raja Das, and Joel H. Saltz. Run-time and compile-time support for adaptive irregular problems. In *Supercomputing*, pages 97–106, 1994.
- [63] Peter Smith and Norman C. Hutchinson. Heterogeneous process migration: The Tui system. *Software Practice and Experience*, 28(6):611–639, 1998.
- [64] Rick Utter. *Diaktoros: Full State Migration with Mobile Agents*. PhD thesis, Dept. of Information and Computer Science, University of California, Irvine, 2006.
- [65] Steve Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), 1997.
- [66] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The locus distributed operating system. In *Proceedings of the ninth ACM symposium on Operating systems principles*, pages 49–70. ACM Press, 1983.
- [67] C. Wicke. Implementation of an autonomous agents system. Master’s thesis, University of Karlsruhe, Germany, 1998.
- [68] Paul R. Wilson and Sheetal V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *1992 Int. Workshop on Object Orientation and Operating Systems*, pages 364–377, Dourdan (France), 1992. IEEE Comp. Society Press.

- [69] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, 1998. ACM Press.