INFORMATION TO USERS

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

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

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

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®
Supporting Type-Safe Languages on DSM Systems

by

Weimin Yu

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

[Signatures]

Dr. Alan Cox, Associate Professor, Chair
Computer Science

Dr. Peter Druschel, Assistant Professor
Computer Science

Dr. John K. Bennett, Associate Professor
Electrical and Computer Engineering

Houston, Texas

May, 2000
Abstract

Supporting Type-Safe Languages on DSM Systems

Weimin Yu

In this thesis, we present a new approach to support transparent sharing of data in a distributed system using modern programming languages, such as Java and Modula-3. Unlike RPC-based techniques such as CORBA and RMI, we provide transparent data sharing using software distributed shared memory (DSM). We find that modern programming languages provide new opportunities for optimization while the garbage collection in such languages provides new implementation challenges.

This thesis centers around the following two claims. First, the run-time type information and safety features in modern programming languages provide new opportunities to support both coarse-grained and fine-grained sharing efficiently on DSM systems. Our new DSM system, DOSA, takes advantage of the language support to efficiently maintain coherency at the object granularity, making fine-grained sharing efficient. It aggregates the communication of objects to make coarse-grained sharing efficient. Second, the overall performance of garbage collected programs on DSM systems can approach that of manual memory management. Our approach achieves this goal by making the implementation of the garbage collector orthogonal to the DSM operations, and by identifying garbage earlier and more accurately.

To substantiate the claims above, we have evaluated our approach against state-of-the-art methods. We first compared the performance of DOSA with that of TreadMarks, a state-of-the-art coarse-grain DSM system. Our evaluation shows that the performance of coarse-grained programs is comparable (within 6%) with TreadMarks, while the performance of fine-grained programs is significantly better (up to 98%) than TreadMarks. We also compared our DSM garbage collector with two state-of-the-art collectors. Our evaluation shows that, with our collector, the overall performance of the programs is close (within 5%) to that of manual management, and far better (up to 50%) than with the existing DSM collectors.
Acknowledgments

I dedicate this work to my parents, as an expression of love and gratitude for providing me with the best education. Without their love and sacrifices, I would not have been able to attend graduate school in the United States. I would like also to thank my wife, Hongsui. Over the last two years, her constant love and care carried me through the most difficult time in my life.

I would like to thank my advisor, Alan Cox, for the inspiration and guidance during the development of this thesis. I am also grateful for his numerous comments on this manuscript. I would like also to thank Peter Druschel and John Bennett for serving on my committee and providing invaluable help on my thesis. My thanks also go to Charlie Hu, for his cooperation over the last year, and to Willy Zwaenepoel, for his advices and comments. I wish to thank Iva Jean Jorgensen for her excellent administrative support.

I thank all the past and current members of the Systems group at Rice University, for the great working environment they provide.
Contents

Abstract ........................................... ii
Acknowledgments ................................... iii
List of Tables ..................................... viii
List of Illustrations .............................. x

1 Introduction .................................... 1
   1.1 Detailed Problem Statement .................. 3
   1.1.1 DSM Systems and Sharing Granularity ..... 3
   1.1.2 Garbage Collection on DSM Systems ...... 5
   1.2 Solutions .................................... 6
   1.3 Thesis ....................................... 7
   1.4 Contributions ................................ 8
   1.5 Thesis Organization ......................... 9

2 Garbage Collection on Conventional DSM Systems 10
   2.1 Background .................................. 10
   2.1.1 Uniprocessor Garbage Collection .......... 11
   2.1.2 DSM Garbage Collection .................. 12
   2.2 Problem Statement ........................... 16
   2.2.1 Intra-Processor Garbage Collection ..... 16
   2.2.2 Inter-processor Garbage Collection ..... 19
   2.3 Intra-Processor Garbage Collectors for TreadMarks 21
   2.3.1 The Mark-sweep Garbage Collector ....... 21
   2.3.2 The Copying Garbage Collector .......... 22
   2.4 Evaluation .................................. 25
   2.4.1 Methodology ............................. 25
   2.4.2 Applications ............................. 26
   2.4.3 Measurement .............................. 27
   2.4.4 Summary ................................ 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5</td>
<td>Solutions</td>
<td>35</td>
</tr>
<tr>
<td>2.6</td>
<td>Summary</td>
<td>36</td>
</tr>
<tr>
<td>3</td>
<td>Design and Implementation of DOSA</td>
<td>37</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>API and Memory Model</td>
<td>38</td>
</tr>
<tr>
<td>3.2.1</td>
<td>API</td>
<td>38</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Memory Model: Release Consistency</td>
<td>39</td>
</tr>
<tr>
<td>3.3</td>
<td>Design</td>
<td>39</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Handles</td>
<td>39</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Lazy Object Storage Allocation</td>
<td>41</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Data Aggregation</td>
<td>42</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Potential Overheads</td>
<td>43</td>
</tr>
<tr>
<td>3.4</td>
<td>Implementation</td>
<td>43</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Consistency Protocol</td>
<td>43</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Data Structures</td>
<td>44</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Object Storage Allocation</td>
<td>45</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Switching Protection</td>
<td>45</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Modification Detection and Write Aggregation</td>
<td>46</td>
</tr>
<tr>
<td>3.4.6</td>
<td>Access Miss Handling and Read Aggregation</td>
<td>46</td>
</tr>
<tr>
<td>3.4.7</td>
<td>Summary</td>
<td>47</td>
</tr>
<tr>
<td>3.5</td>
<td>Compiler Optimizations for Coarse-grained Applications</td>
<td>47</td>
</tr>
<tr>
<td>3.6</td>
<td>Language Support</td>
<td>50</td>
</tr>
<tr>
<td>3.7</td>
<td>Related Work</td>
<td>50</td>
</tr>
<tr>
<td>3.8</td>
<td>Summary</td>
<td>52</td>
</tr>
<tr>
<td>4</td>
<td>Garbage collection on DOSA</td>
<td>53</td>
</tr>
<tr>
<td>4.1</td>
<td>Intra-Processor Garbage Collection Algorithms</td>
<td>54</td>
</tr>
<tr>
<td>4.1.1</td>
<td>The Mark-sweep Garbage Collector</td>
<td>54</td>
</tr>
<tr>
<td>4.1.2</td>
<td>The Copying Garbage Collector</td>
<td>55</td>
</tr>
<tr>
<td>4.2</td>
<td>Inter-processor Garbage Collection</td>
<td>55</td>
</tr>
<tr>
<td>4.2.1</td>
<td>The Graph-Tracing Algorithm</td>
<td>56</td>
</tr>
<tr>
<td>4.2.2</td>
<td>The Cost of the Centralized Tracing</td>
<td>61</td>
</tr>
<tr>
<td>4.3</td>
<td>Related Work</td>
<td>62</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Distributed Garbage Collection</td>
<td>63</td>
</tr>
</tbody>
</table>
4.3.2 Garbage Collection on DSM Systems ........................................ 64
4.4 Summary ............................................................................. 65

5 Evaluation: Non-Garbage-Collected Programs ............................. 67
5.1 Methodology ...................................................................... 67
5.2 Applications ...................................................................... 68
5.3 Overall Results .................................................................. 68
  5.3.1 Fine-grained Applications .............................................. 68
  5.3.2 Coarse-grained Applications .......................................... 72
5.4 Effects of the Various Optimizations .................................... 74
  5.4.1 Lazy Object Allocation .................................................. 77
  5.4.2 Read Aggregation .......................................................... 77
  5.4.3 Write Aggregation .......................................................... 79
  5.4.4 Write Notice Reduction .................................................. 79
  5.4.5 Compile-time Optimization .............................................. 80
5.5 Discussion .......................................................................... 80
5.6 Summary ........................................................................... 81

6 Evaluation: Garbage Collection ................................................. 82
6.1 Applications ...................................................................... 83
6.2 Inter-Processor Garbage Collection ...................................... 84
  6.2.1 Measurement ................................................................ 85
  6.2.2 Discussion ................................................................... 96
  6.2.3 Summary ..................................................................... 97
6.3 Intra-Processor Collection ................................................... 98
  6.3.1 Measurements ............................................................. 98
  6.3.2 Discussion on Fine-grain DSM Systems ......................... 106
6.4 Conclusions ....................................................................... 107

7 Conclusions and Future Work .................................................. 109
7.1 Conclusions ..................................................................... 109
7.2 Future Work ..................................................................... 110
  7.2.1 Heterogeneous DSM System ........................................... 110
  7.2.2 Supporting Dynamic Sharing Patterns ............................. 111
Tables

2.1 Program description and sequential performance. ................... 27
2.2 Running time and garbage collection time (in seconds) for
  TreadMarks on 1 processor. For manual memory management, GC
time is the time spent in free. ......................... 28
2.3 Running time on 32 processors. ................................ 28
2.4 Detailed statistics for Game on 32 processors. ..................... 29
2.5 Detailed statistics for BH on 32 processors. ....................... 31
2.6 Detailed statistics for MIP on 32 processors. ...................... 33

5.1 Applications, input data sets, and sequential execution time. ..... 69
5.2 Detailed statistics for TreadMarks and DOSA on 8 processors for
  fine-grained applications, Barnes-Hut and Water-Spatial. .......... 71
5.3 Detailed statistics for TreadMarks and DOSA on 16 processors for
  fine-grained applications, Barnes-Hut and Water-Spatial. .......... 71
5.4 Detailed statistics for TreadMarks and DOSA on 32 processors for
  fine-grained applications, Barnes-Hut and Water-Spatial. .......... 71
5.5 Detailed statistics for TreadMarks and DOSA on 8 processors for
  coarse-grained applications SOR and Water-N-Squared. ............ 74
5.6 Detailed statistics for TreadMarks and DOSA on 16 processors for
  coarse-grained applications SOR and Water-N-Squared. ............ 75
5.7 Detailed statistics for TreadMarks and DOSA on 32 processors for
  coarse-grained applications SOR and Water-N-Squared. ............ 75
5.8 Statistics for DOSA and DOSA without each of the optimizations on
  32 processors for Barnes-Hut, Water-Spatial, and Water-N-Squared. 76
5.9 Statistics for DOSA with full and partial read aggregation on 32
  processors for Barnes-Hut and Water-Spatial with the large data set. 78
5.10 Running time (sec.) comparison between SOR with and without the
  compile-time optimization. ................................... 80
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Running time and garbage collection time (in seconds) for TreadMarks and DOSA on 1 processor. For manual memory management, GC time is the time spent by free.</td>
</tr>
<tr>
<td>6.2</td>
<td>Total execution time on 8, 16 and 32 processors (in seconds).</td>
</tr>
<tr>
<td>6.3</td>
<td>Detailed statistics for Game on 8 processors.</td>
</tr>
<tr>
<td>6.4</td>
<td>Detailed statistics for Game on 16 processors.</td>
</tr>
<tr>
<td>6.5</td>
<td>Detailed statistics for Game on 32 processors.</td>
</tr>
<tr>
<td>6.6</td>
<td>Detailed statistics for BH on 8 processors.</td>
</tr>
<tr>
<td>6.7</td>
<td>Detailed statistics for BH on 16 processors.</td>
</tr>
<tr>
<td>6.8</td>
<td>Detailed statistics for BH on 32 processors.</td>
</tr>
<tr>
<td>6.9</td>
<td>Detailed statistics for MIP on 8 processors.</td>
</tr>
<tr>
<td>6.10</td>
<td>Detailed statistics for MIP on 16 processors.</td>
</tr>
<tr>
<td>6.11</td>
<td>Detailed statistics for MIP on 32 processors.</td>
</tr>
<tr>
<td>6.12</td>
<td>Total running time (in seconds) on 32 processors.</td>
</tr>
<tr>
<td>6.13</td>
<td>Detailed statistics for Game on 8 processors.</td>
</tr>
<tr>
<td>6.14</td>
<td>Detailed statistics for Game on 16 processors.</td>
</tr>
<tr>
<td>6.15</td>
<td>Detailed statistics for Game on 32 processors.</td>
</tr>
<tr>
<td>6.16</td>
<td>Detailed statistics for BH on 8 processors.</td>
</tr>
<tr>
<td>6.17</td>
<td>Detailed statistics for BH on 16 processors.</td>
</tr>
<tr>
<td>6.18</td>
<td>Detailed statistics for BH on 32 processors.</td>
</tr>
<tr>
<td>6.19</td>
<td>Detailed statistics for MIP on 8 processors.</td>
</tr>
<tr>
<td>6.20</td>
<td>Detailed statistics for MIP on 16 processors.</td>
</tr>
<tr>
<td>6.21</td>
<td>Detailed statistics for MIP on 32 processors.</td>
</tr>
</tbody>
</table>
Illustrations

2.1 Chains of stub-scion pairs. .................................................. 20
2.2 Cycles of references. .......................................................... 20
2.3 Retained garbage in Game with the WRT collector. ...................... 30

3.1 Objects with handles. .......................................................... 40
3.2 Shared objects identified by unique OIDs. .................................. 41
3.3 Access detection using the handle pointers. .................................. 42
3.4 Out-of-bound access. .......................................................... 50
3.5 Multiview: one vpage for each object. ...................................... 52

4.1 A bipartite graph. .............................................................. 60

5.1 The speedup of Barnes-Hut for the small data set. ....................... 69
5.2 The speedup of Barnes-Hut for the large data set. ....................... 69
5.3 The speedup of Water-Spatial for the small data set. ................... 70
5.4 The speedup of Water-Spatial for the large data set. ................... 70
5.5 The speedup of SOR for the small data set. ............................... 73
5.6 The speedup of SOR for the large data set. ............................... 73
5.7 The speedup of Water-N-Squared for the small data set. ............... 73
5.8 The speedup of Water-N-Squared for the large data set. ............... 73
5.9 Speedup comparison between DOSA and DOSA without each of the optimizations on 32 processors. ................................. 75

6.1 The speedup of Game. ....................................................... 87
6.2 The speedup of BH. .......................................................... 87
6.3 The speedup of MIP. ......................................................... 87
6.4 Retained garbage in Game with the graph-tracing collector. .......... 87
6.5 Retained garbage in Game with the WRT collector. ..................... 88
6.6 Retained garbage in Game with the Stub-scion collector.  . . . . . . . . . . . . . . . . . 88
6.7 The speedup of Game with the mark-sweep collectors.  . . . . . . . . . . . . . . . . . . . . . . 99
6.8 The speedup of Game with the copying collector.  . . . . . . . . . . . . . . . . . . . . . . . . . . 99
6.9 The speedup of BH with the mark-sweep collector.  . . . . . . . . . . . . . . . . . . . . . . . . . 99
6.10 The speedup of BH with the copying collector.  . . . . . . . . . . . . . . . . . . . . . . . . . 99
6.11 The speedup of MIP with the mark-sweep collector.  . . . . . . . . . . . . . . . . . . . . . . 100
6.12 The speedup of MIP with the copying collector.  . . . . . . . . . . . . . . . . . . . . . . . . . 100
Chapter 1

Introduction

In this thesis, we present a new approach to support transparent sharing of data in a distributed system using modern programming languages, such as Java and Modula-3. We provide support for transparent sharing of data using a new form of software distributed shared memory (DSM) [LH89]. By transparent sharing we mean that the distributed nature of the system is transparent to the programmer, and the local and remote data can be accessed in the same way. A DSM system presents a shared memory abstraction over a network of workstations. It transparently handles the communication between the processors and provides an easy to use programming model. In contrast, the Remote Procedure Call (RPC) [BN84] based techniques such as CORBA and Java's Remote Method Invocation (RMI) mechanism do not support transparent sharing. In RMI, for example, the programmer must distinguish private objects from shared objects. Furthermore, the programmer must specify how the shared objects are to be communicated between the processors on a per class basis. This is inflexible and may not suit the needs of the applications well [YC97].

We aim to support modern programming languages such as Java and Modular-3, which make sufficient type information available at run-time to enforce type safety and to perform efficient garbage collection. Compared with historical languages, like C or C++, modern languages such as Java encourage the writing of clearer and safer programs. Until recently, the main obstacle to the wide acceptance of the modern languages has been the perception that they have poor performance. However, their underperformance is due to implementation problems rather than inherent flaws. With the popularity of Java and the great effort devoted to its implementation, it is expected to achieve decent performance and wider acceptance in the near future. Therefore, we deem it important to support such languages efficiently in a distributed environment.

We find that the run-time type information and the safety features in modern languages such as Java provide new opportunities for optimizing the DSM, while
garbage collection presents new implementation challenges on DSM. This thesis centers around the following two issues:

1. Taking advantage of the run-time type information and the safety features to efficiently support different sharing granularities on DSM, and,

2. Improving the performance of garbage-collected programs on DSM, and making their performance close to that of programmer memory management.

Different parallel programs often have different sharing granularities. In a coarse-grained program, the shared data items accessed by each processor have good spatial locality. In a fine-grained program, the shared data items accessed by each processor have poor spatial locality. Conventional DSM systems are not optimized for modern programming languages, and are limited to supporting only one granularity with good performance. Indeed, DSM systems have been divided into those offering support of coarse-grained sharing or fine-grained sharing. Each of these systems has merit, but none of them is ideal from the standpoint of transparently supporting a broad range of programs.

Garbage collection presents new implementation challenges to DSM in that the garbage collector often has side effects that seriously reduce the overall program performance. However, the issue of overall program performance has not been addressed by previous research. On a DSM system a garbage collector consists of two parts: an intra-processor collector and an inter-processor collector. The intra-processor collector allocates and reclaims objects in each processor's heap, while the inter-processor collector manages the references that cross processor boundaries. The implementation of the intra-processor collector can seriously affect the program performance on a DSM system. For example, a mark-sweep algorithm may reduce the spatial locality of a program, increasing the amount of communication on many conventional DSM systems [YC96]. Although a copying collector can improve the spatial locality, the object movement and reference updates generate extra modifications that must be propagated across the system, also increasing the amount of communication. Previous inter-processor garbage collectors for DSM systems focused on reducing the amount of communication between the processors. However, these algorithms are slow to reclaim shared objects, reducing the size of the available memory and increasing the cost of garbage collection. All these effects reduce the overall program performance on DSM systems.
This thesis develops new solutions to both of the problems above. We have designed a new DSM system that takes advantage of modern programming languages to efficiently support both coarse-grained and fine-grained sharing patterns, and to allow efficient garbage collection. The run-time type information and the safety features in such languages allow us to efficiently implement the abstraction of a shared object space. As a result, we are able to efficiently keep track of memory accesses and maintain coherency at the object granularity, making fine-grained sharing efficient. We also aggregate the communication of objects to make coarse-grained sharing efficient. In the case of garbage collection, the shared object space abstraction makes the implementation of the intra-processor garbage collector (either mark-sweep or copying) orthogonal to the DSM system, eliminating many sources of communication. We have also designed a new inter-processor algorithm that reclaims shared objects more quickly than state-of-the-art algorithms, improving the overall program performance.

We have conducted two studies to evaluate the new DSM design and the garbage collector. The evaluations were done on a network of 32 workstations. Our results show that the new DSM design can support both coarse-grained and fine-grained sharing efficiently; and that with our garbage collector, the overall performance of the garbage-collected programs is close to that with programmer memory management, and is far better than with previous DSM garbage collectors.

The remainder of this chapter is organized as follows. Section 1.1 discusses the main problems this thesis attempts to solve. Section 1.2 briefly describes the design of our new DSM system and garbage collection algorithms. Section 1.3 spells out the experiments we undertake to evaluate our designs, and Section 1.4 summarizes the contributions of this thesis. The outline of the rest of this thesis is in Section 1.5.

1.1 Detailed Problem Statement

1.1.1 DSM Systems and Sharing Granularity

A software distributed shared memory system [LH89] provides a shared memory abstraction on top of a network of workstations. It transparently handles the communication of data between machines, eliminating the need for the programmer to write message-passing code. It is widely accepted that it is easier to program with shared memory than message passing: Instead of sending and receiving messages explicitly, programs can use ordinary loads and stores to access shared data. This enables pro-
grammers to concentrate on algorithmic issues rather than on managing partitioned data sets and communicating values.

Conventional distributed shared memory (DSM) systems do not take advantage of modern programming languages, and only support the sharing of an untyped memory region *. These systems are limited to providing only one sharing granularity with good performance. Indeed, DSM systems have been divided into those offering support for coarse-grained sharing or for fine-grained sharing.

A program has a coarse-grained sharing pattern if the shared data items accessed by each processor have good spatial locality. Coarse-grain sharing systems are typically page-based, and use the virtual memory hardware for access and modification detection. The hardware detection has low overhead, and communicating all data in a page using one message takes advantage of the good spatial locality in coarse-grained programs. In fine-grained programs, however, the hardware access detection at the page granularity is inaccurate, increasing the chance of false sharing. Although relaxed memory models and multiple-writer protocols relieve the impact of the large page size, fine-grained sharing and false-sharing remain problematic [ACD+96].

A program has a fine-grained sharing pattern if the shared data items accessed by each processor have poor spatial locality. Fine-grain sharing systems typically augment the code with instructions to detect reads and writes, freeing them from the large size of the consistency unit in virtual memory-based systems, but introducing per-access overhead that reduces performance for coarse-grained applications. In addition, these systems do not benefit from the implicit aggregation effect present in the page-based systems. Fine-grain systems typically require a message per object, while page-based systems bring in all data in a page at once, avoiding additional messages if the application accesses other objects in the same page.

In short, conventional DSM systems cannot support both coarse-grained and fine-grained sharing efficiently, mainly because of the limitation in the memory access detection mechanisms they use.

*Orca [BKT92, BBH+98], which is a DSM system coupled with an object-oriented programming language, provides the abstraction of a shared object space to the programmer through the language API. However, the underlying DSM implementation still provides the abstraction of an untyped shared memory region to the language implementation.
1.1.2 Garbage Collection on DSM Systems

A garbage collection (GC) system automatically reclaims the unused memory, eliminating the need for the programmer to write code to track the status of allocated memory.

Garbage collection on DSM systems consists of two related issues: managing each processor's local memory (the *intra-processor* collection), and reclaiming the references that cross processor boundaries (the *inter-processor* collection).

On conventional DSM systems the intra-processor collector may have a negative impact on the system performance. Most garbage collection algorithms can be categorized as either *mark-sweep* [McC60] or *copying* [FY69, Che70] algorithms. Past research on uniprocessor garbage collection has shown that copying algorithms often offer superior performance over mark-sweep algorithms. However, on a conventional DSM system, the virtual address of each shared object must be the same across the system. Therefore, a copying algorithm incurs extra overheads in three ways. First, the move of an object must be synchronized. Second, whenever a processor moves an object, the new address of the object must be propagated to all other processors that access this object. Finally, a DSM system may not be able to distinguish the modifications due to object movement from the modifications made by normal program execution, and have to propagate both kinds of modifications across the system. This increases the amount of communication and affects the overall program performance.

A mark-sweep algorithm may also reduce the program performance on a conventional DSM system. A mark-sweep garbage collector often reduces the spatial locality of the program. The poor spatial locality increases the communication cost on conventional coarse-grain DSM systems in two ways: First, related objects are scattered in more pages. Since data is communicated at the page granularity in the coarse-grain DSM systems, a processor must send more messages if the spatial locality is poor. Second, irrelevant objects may be intermingled in the same page, increasing the chance of false sharing.

In the case of the inter-processor collection, state-of-the-art algorithms focused on the cost of garbage collection, ignoring their effects on overall program performance. To reduce the communication cost incurred by the garbage collector, these algorithms limit the scope of the information exchanged between the processors, and exchange this information asynchronously. Thus, intra-processor garbage collections are performed independently based on the information that is readily available locally.
However, garbage identification based only on local information is inaccurate and has a negative impact on the program performance. First, without complete information, the garbage collector has to be conservative and may delay the collection of many dead objects. Second, a processor cannot distinguish live objects that are temporarily unused by the local processor from garbage, and may flush such objects out of the local memory. If these objects are accessed later, they must be brought back again. This results in thrashing and increases the amount of communication.

1.2 Solutions

We have designed a new DSM system, DOSA, that takes advantage of modern programming languages to support efficient distributed sharing of objects, and to allow efficient garbage collection. DOSA requires that the programming language provide type information at run-time. The run-time information must be sufficient so that it allows an unambiguous determination of whether a memory location contains an object reference or not. In addition, in the case of a reference, the type and size of its referent must be known at run-time. DOSA also assumes that the programs written in such languages behave safely, e.g., an object access cannot go beyond the end of the object. Many modern languages satisfy this requirement, including Java and Modula-3.

Taking advantage of the run-time type information and the safety features, DOSA efficiently implements the abstraction of a shared object space. It uses the run-time type information, in particular, the ability to determine the object sizes at run time, to make fine-grained sharing efficient by supporting coherency at the object granularity. DOSA also aggregates the communication of objects to make coarse-grained sharing efficient.

The key implementation technique in DOSA is the use of "handles". A handle is a small object header holding a pointer to the object. All accesses to an object are indirect through the object's handle, and an object is named by a globally unique handle id. With the extra indirection, DOSA is able to transparently keep track of memory access at the object granularity using the virtual memory hardware: For each object we keep three VM mappings with different access permissions: invalid, read-only, and read-write. The state of an object can be changed by setting the object pointer in its handle to the appropriate mapping of the object. As we will explain in Section 3.6, the run-time type information is necessary to set up the VM mappings for
each object, while the safety features guarantee the correctness of the access detection mechanism. By naming objects using their handle ids instead of their addresses in the virtual memory, DOSA also allows each processor to allocate memory only for the objects it accesses, drastically improving the spatial locality of the programs.

On the garbage collection front, the shared object abstraction separates an object's naming from its address in memory, making the implementation of the intra-processor garbage collector orthogonal to the DSM system. In particular, the movement of an object on one processor does not appear as a write to other processors. Therefore, each processor is free to use any garbage collection algorithm for its intra-processor collector without imposing any communication cost on the processors it interacts with, allowing better program performance. To further improve the program performance, we have also implemented a new adaptive inter-processor garbage collector. We found that although the size of the complete object liveness information is very large in the worst case, it is actually small for many commonly used parallel data structures. Therefore, our inter-processor garbage collector dynamically adapts between two algorithms. It uses the complete liveness information gathered from the entire system to identify garbage more accurately and quickly in the common case, improving the overall program performance in such case. It also maintains good worst case performance by falling back to a state-of-the-art inter-processor collection algorithm in the worst case.

1.3 Thesis

This thesis argues that the combination of the DOSA system and our new garbage collection algorithm efficiently supports transparent sharing of data in a distributed system using modern programming languages. We make the following three claims in this thesis:

Our first claim is that DOSA supports both coarse-grained and fine-grained sharing efficiently.

We compare the performance of DOSA with that of the TreadMarks DSM system [ACD+96]. TreadMarks is a state-of-the-art DSM system that is efficient at handling coarse-grained sharing. We use TreadMarks as the yardstick with which to measure the success of DOSA.

TreadMarks detects accesses using virtual memory hardware and maintains coherency at the page granularity, benefiting from the good spatial locality in coarse-
grained programs. DOSA uses the same access detection technique, and aggregates the communication of objects to efficiently support coarse-grained programs. Although DOSA introduces extra dereferences due to the use of handles, classical compile-time optimization techniques can eliminate many of the extra dereferences in DOSA, making the dereference cost insignificant. With object level coherency management, DOSA is far more efficient than TreadMarks at handling fine-grained programs.

Our second claim is that the shared object space abstraction allows more efficient intra-processor garbage collection and better overall program performance than conventional DSM systems.

Conventional DSM systems support sharing over an untyped memory region. Tread-Marks is one example of such systems. We have implemented two intra-processor garbage collectors that are representative of those in common use on both DOSA and TreadMarks. One of the collectors is based on mark-sweep, the other on generational copying. On DOSA, the implementation of the intra-processor garbage collector is made orthogonal to the DSM system. In contrast, on TreadMarks the garbage collectors still have negative impact on the DSM performance. Our results show that the garbage-collected programs have much better performance on DOSA than on TreadMarks.

Our third claim is that our new inter-processor collector identifies shared garbage earlier and more accurately, allowing better overall program performance than previous collectors.

We have implemented our new collector as well as two state-of-the-art inter-processor collectors on DOSA and compared their performance. Our measurement shows that our collector reclaims shared objects earlier, reducing the cost of the intra-processor garbage collections and allowing better program performance. It also identifies garbage more accurately, eliminating the thrashing problem.

1.4 Contributions

The primary contributions of this thesis are the design, implementation, and evaluation of the DOSA system and a new garbage collector for DSM systems, and the consequent validation of the three claims made in Section 1.3.

Claim 1 says that DOSA supports both coarse-grained and fine-grained sharing efficiently. To validate this claim, we compared the performance of DOSA with that
of TreadMarks, a DSM system that is efficient at handling coarse-grained sharing. Our evaluation shows that: the performance of coarse-grained programs on DOSA is comparable (within 6%) with TreadMarks, and the performance of fine-grained programs is significantly better (up to 98%) than TreadMarks.

Claim 2 says that the shared object space abstraction allows more efficient intra-processor collection and better performance than conventional DSM systems. We have compared the performance of two intra-processor garbage collection algorithms that are representative of those in common use on both DOSA and TreadMarks. Our evaluation shows that, with the shared object abstraction, the performance of the garbage-collected programs on DOSA is much better (up to 41%) than on TreadMarks.

Claim 3 says that our new inter-processor collector identifies shared garbage earlier and more accurately, allowing better overall program performance than existing collectors. We have implemented two state-of-the-art inter-processor collectors as well as our new collector on DOSA and measured their performance. Our evaluation shows that the new garbage collector improves the overall program performance by up to 50% over the existing DSM collectors. Furthermore, with the combination of the shared object space abstraction and our new inter-processor garbage collector, the performance of the garbage-collected programs is close (within 5%) to that of manual memory management. We are also the first to evaluate the overall performance of the garbage-collected programs on DSM systems.

1.5 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 presents an evaluation of garbage collection on conventional DSM systems. It identifies the performance problems in state-of-the-art DSM garbage collectors, and motivates the design of DOSA and the adaptive inter-processor garbage collection algorithm. The design and implementation of DOSA are presented in Chapter 3, and the design of our adaptive algorithm for inter-processor garbage collection is presented in Chapter 4. The evaluation of DOSA using non-garbage-collected programs is presented in Chapter 5, and the evaluation of the garbage-collected programs is presented in Chapter 6. Finally, we offer our concluding remarks and a discussion of future work in Chapter 7.

Related work is examined in each chapter.
Chapter 2

Garbage Collection on Conventional DSM Systems

This chapter presents an evaluation of garbage collection on conventional DSM systems. We identify the performance problems in state-of-the-art DSM systems and garbage collection algorithms. These problems, which we have already outlined in Section 1.1.2, motivated our design of the DOSA system and the adaptive algorithm for inter-processor garbage collection. Our evaluation has been conducted on TreadMarks. However, as we will explain later in this chapter, most of the problems found in TreadMarks also exist in other conventional DSM systems.

The remainder of this chapter is organized as follows. Section 2.1 presents the background information on garbage collection. Section 2.2 states the problems in the design of both intra-processor and inter-processor garbage collection algorithms for DSM systems. Section 2.3 describes the implementation of two intra-processor garbage collectors for TreadMarks. We evaluate the performance of the two collectors in Section 2.4. We propose our solutions in Section 2.5, and summarize this chapter in Section 2.6.

2.1 Background

A garbage collector keeps track of the dynamically allocated objects and automatically reclaims those that are no longer used by the program, eliminating the need for the programmer to manually reclaim memory. Garbage collection eliminates the possibility of "dangling" references (references to live objects that are erroneously deallocated), which are a source of many hard to diagnosis errors in programs with manual memory management. Furthermore, it has been reported that garbage collection can exceed or match the performance of programmer memory management [App87, Zor93] on uniprocessor systems.

In this section we first briefly discuss uniprocessor garbage collection (GC) techniques. Then we review earlier work that represents the state-of-the-art in DSM garbage collection. We will later compare our new approach with these algorithms.
2.1.1 Uniprocessor Garbage Collection

Wilson [Wil92] has given a comprehensive survey of the uniprocessor garbage collectors. Most garbage collectors are tracing collectors. They work by starting from a root set of memory objects, and following references from these objects to other objects recursively, until all objects reachable from the roots have been found. Inaccessible objects are garbage and can be reclaimed. There are two classes of collectors: copying collectors [FY69, Che70], which copy accessible objects to another part of the address space and reclaim the entire old region; and mark-sweep collectors [McC60], which mark all accessible objects, then scan the heap and reclaim unmarked objects. In a copying garbage collector, the old memory region is called the fromspace, and the new region is called the tospace. They are collectively referred to as semispaces.

Garbage collectors allocate and reclaim memory cyclically. Because of the delay between the time an object becomes garbage and the time it is reclaimed and reused, garbage collectors may increase the footprint of the heap, decreasing the spatial locality of a program. A copying garbage collector can improve the spatial locality by moving the live objects together. A mark-sweep collector, however, does not improve the spatial locality.

Past research on uniprocessor garbage collection algorithms has shown that copying garbage collectors usually work better than mark-sweep garbage collectors if sufficient memory is available [App87]. There are two reasons for copying collectors' superiority. First, the cost of a copying collector is proportional to the size of the live data instead of the size of the heap. A copying collector only moves and examines the live objects, while a mark-sweep collector must examine every object in the heap in the sweep phase. Second, a copying collector moves the live data together, improving the spatial locality. This improves the cache and virtual memory performance. Copying garbage collection algorithms also effectively solve the problem of heap fragmentation.

The cost of the garbage collection can be further reduced by a generational approach [LH83]. It has been observed in many programs, in a variety of programming languages, that most objects live for a very short time, while a small percentage of them live a longer time [LH83, Ung84, Sha88, Zor90, DeT90, Hay91]. Generational collection takes advantage of this fact by segregating objects into multiple areas by age, and collecting the areas containing younger objects more frequently than the
older ones. Young objects that survive one or more garbage collections are advanced to older generations.

Although the generational approach has been applied to both mark-sweep and copying collectors [DWH+90], most generational collectors are copying collectors. This is because in a copying collector it is easy to separate the generations by moving objects belonging to different generations to different address ranges.

2.1.2 DSM Garbage Collection

In a DSM system, the registers, stacks, and the top-level variables form the local “roots” of that processor. The join of the local roots on all processors form the global roots of the DSM system. The objects reachable from the global roots form the global object graph.

The fundamental factor that distinguishes DSM garbage collection from uniprocessor garbage collection is that the global roots and the object graph are distributed among the processors. Each processor may only cache a portion of the graph. Therefore, an object may reside on one processor while all references to it are within objects residing on other processors. With the high cost of inter-processor communication in distributed systems, it is very expensive to traverse the entire object graph at once. To reduce the communication cost, state-of-the-art DSM collectors [FS94a, YC96] perform garbage collections independently. That is, each processor in the system performs garbage collections based on locally available information; and the processors exchange information asynchronously, usually by piggybacking such information on messages initiated by the normal program execution. In these algorithms, each processor maintains a set of cross-processor references to locally owned objects. The garbage collector on each processor traverses the cached portion of the object graph starting from both the local roots and the set of cross-processor references. A DSM garbage collector can therefore be divided into two parts: an intra-processor algorithm that performs the traversal and reclaims garbage; and an inter-processor algorithm that exchanges necessary data between the processors so that the dead references can be removed from the set of cross-processor references on each processor.

The implementation of an intra-processor collector closely interacts with the underlying DSM system. We will discuss this issue in detail in Section 2.2. There are only two inter-processor garbage collectors for DSM systems that have been im-
implemented and evaluated. Both collectors are adapted from garbage collectors for classical RPC based distributed systems [AMR92, PS95]. We briefly describe the two algorithms below.

The Stub-scion Algorithm

Ferreira and Shapiro [FS94a] have implemented a garbage collector based on the stub-scion algorithm [SGP90]. In this algorithm, each processor maintains a stub table and a scion table. When a message containing an object is sent or received, it is checked for references. For each reference in the message, the sender creates a scion (identified by the reference and the receiver's processor id) unless such a scion already exists, and the receiver creates a stub (identified by the reference and the sender's processor id) if such a stub does not already exist. In other words, if a reference has been passed from the sender to the receiver, there will be a stub-scion pair for this reference between the two processors. Repeated passing of the same reference between the same two processors will only result in one stub-scion pair. However, a single reference may have multiple scions (stubs) on a processor if it has been passed to (received from) more than one remote processor.

In each intra-processor collection, the references in the scion table are traced together with the processor's local "roots". After a collection, the stubs that are not reachable from either the local roots or the scions are removed from the stub table. A processor periodically sends its stubs to other processors. This can be done asynchronously by piggybacking the stubs in messages initiated by the normal program execution. When a processor receives a stub list, it checks its own scion list against the stubs. A scion is removed if its corresponding stub is not found in the stub list that has been received.

The Weighted Reference Counting Algorithm

In our previous work [YC96], we have implemented a DSM garbage collector based on a weighted reference counting (WRT) [Tho81, WW87, Bev87] algorithm.

In this algorithm, each processor maintains a table of imported and exported references. When a message containing an object is sent or received, it is checked for references. On the sending side, any references to locally created objects are entered into the export table; and on the receiving side, all references to remote objects are entered into the import table. In other words, if a message contains object A
and object $A$ references an object $B$ that was created by the sender, object $B$ is entered into the sender's export table and the receiver's import table. In each intraprocessor collection, the references in a processor's export table are traced together with the processor's local "roots"; and the imported references that are not reachable from either the local roots or the exported references are "dead" references to the processor. When a processor later determines that an imported reference is dead, the processor notifies the object's creator that the reference is no longer used and removes it from its import table.

The export and import tables are conceptually similar to the scion and stub tables in the stub-scion algorithm. However, there are subtle differences. The export table of a processor only contains references to the objects created by this processor, while the scion table contains both references to local objects and remote references that have been passed through this processor. A reference has only one entry in the export table (import table) no matter how many remote processors it has been passed to (received from). Also note that our algorithm sends the "dead" import references to their creators, while the stub-scion algorithm sends the "live" references to the processors from which they are received.

In our algorithm, the object creator's task of determining whether any references to the object still exist on some processor is complicated by the fact that a processor can send a reference to the object to another processor without the creator's knowledge. For example, consider a system consisting of three processors: processor $N_1$ creates an object and exports a reference to the object to processors $N_2$ and $N_3$. When processor $N_2$ no longer possesses a reference to the object, it notifies processor $N_1$. Then processor $N_3$ passes a reference to the object to processor $N_2$ and removes its own references to the object. Even though processor $N_1$ has received notifications from both processors $N_2$ and $N_3$, it must recognize that there is still a valid reference to the object. We use a technique called weighted reference counting [Tho81, WW87, Bev87] to solve this problem. It works as follows:

- A reference is assigned a predetermined weight when it is first exported by its creator.

- Whenever a reference is duplicated across a processor boundary, the weight of the reference is equally divided between the local reference and the new remote reference, so that the sum of the weights remains constant.
• When a reference is no longer used and is sent back to its creator, its weight is also returned. When the sum of the returned weights equals the original weight assigned at the reference's creation, the creator processor is sure that no one else needs this reference.

The primary problem of this technique is that the weight of a reference frequently passed between processors may underflow, that is, its weight may reach one and cannot be further divided. Watson and Watson [WW87] have proposed a solution to this problem. In their solution, whenever a weight of a reference underflows on a processor, the processor creates a new weight for the reference. If the reference is subsequently transferred to another processor, the processor divides this new weight between itself and the recipient. In addition to passing half of the weight to the recipient, the processor also passes a flag indicating that the weight should be returned to it instead of the creator of the reference. The processor does not return the reference to its creator until the sum of the returned weights equals the new weight.

Synchronous Global Collection

One important limitation of both algorithms described above is that they cannot reclaim cycles of garbage that span processors. However, the use of common data structures such as doubly-linked lists may result in such cycles. If the size of the cyclic garbage becomes too large, a cycle-reclaiming collector must be invoked. Here we describe a synchronous global collector used in our evaluation [TY97]. In a synchronous global collection, the system is suspended and the processors cooperate in a global tracing of the entire object graph. The global collection is expensive and is invoked only as the last resort.

In a synchronous global collection, each processor starts tracing from its local roots. For each reference found in the traversal, the referent is recursively traced if the local copy of the referent is up-to-date. Otherwise the reference is sent in a mark message to the processor that has the up to date copy of the referent. In practice, the mark messages to the same processor can be batched and sent together.

When a processor receives a mark message, it starts tracing from the named references. During the tracing, the receiver may send out several more mark messages. The receiver acknowledges a mark message to its sender if all of the mark messages derived from this message have been acknowledged. The global collection stops when none of the processors has any mark message waiting to be acknowledged.
2.2 Problem Statement

In this section we first state the problems facing the intra-processor garbage collectors on conventional DSM systems. Then we discuss the drawbacks of previous inter-processor collection algorithms for DSM systems.

2.2.1 Intra-Processor Garbage Collection

The primary challenge to implementing an intra-processor garbage collector on conventional DSM systems is that the garbage collector may reduce the performance of the subsequent program execution. For example, a mark-sweep collector reduces the spatial locality of the program, increasing the amount of communication [YC96]. In contrast, a copying collector can improve the spatial locality, but the object movement and reference updates generate extra modifications that must be propagated across the system, also increasing the amount of communication.

Mark-sweep

With a mark-sweep collector, the reduced spatial locality is especially detrimental to coarse-grain DSM systems which perform page-based data aggregation, e.g., TreadMarks. In such systems, the poor spatial locality increases the amount of communication in two ways. First, the shared objects are scattered in more pages. Since each page is communicated in a separate message, the reduced spatial locality increases the number of messages needed to fetch useful data. Secondly, the poor locality may introduce more false sharing because the garbage may be collected and reused for unrelated objects. This increases both the number of messages and the amount of data communicated.

Conventional fine-grain DSM systems do not perform data aggregation. Therefore they do not suffer from the two aforementioned problems when using a mark-sweep garbage collector. However, a mark-sweep collector may still be undesirable because it incurs higher sequential cost in terms of garbage collection time and memory hierarchy performance.

Copying

A Copying garbage collector can eliminate the problems associated with the mark-sweep collectors. However, it incurs extra overheads on conventional DSM systems
because the address of each object must be the same on all processors. The sources of these overheads include the need to synchronize the object movement, to propagate the *bogus writes*, to update remote references, and to delay the reuse of the fromspace. These issues are discussed below.

**Synchronization.** Synchronization is necessary because two processors conducting garbage collections simultaneously cannot be allowed to move the same object to different addresses. The straightforward solution is to require that a processor acquire a lock associated with an object before moving it. However, with the high interprocessor communication cost in distributed systems, the synchronization cost can become prohibitive. The alternative is to adopt a *single-writer, last-writer-move* policy [FS94a]. Under this policy, only one processor may write to a consistency unit at any particular time, and only the last writer of a consistency unit may move the objects in it.

The last-writer-move policy presents a problem to some coarse-grain DSM systems. It has been shown that for such systems a multiple-writer protocol may yield better performance [Kel96]. Therefore, to adopt the last-writer-move policy means that such systems must use an inferior protocol just for the sake of the copying garbage collector. For existing fine-grain DSM systems, a single-writer protocol presents no serious problem.

**The Bogus Writes.** Conventional DSM systems cannot distinguish the writes due to object movement from the writes made by the program execution. Therefore, if an unchanged object is moved by one processor, the object at the new address will be propagated to remote processors by the conventional DSM systems. This increases the amount of communication. We call the move of an unchanged object a *bogus write* in that the write is not made by the normal execution of the program.

**Remote Reference Update.** Updating references to moved objects on remote processors also incur communication costs on conventional DSM systems. It is also difficult if the type of the stack cells are not always available.

When an object is moved by one processor, its new address must be propagated to all other processors that access it. This increases the amount of communication. Furthermore, it may be difficult to identify the stale references that need update in languages such as Java. Java uses an operand stack whose entries can hold either
references or scalar values. The cost would be very high if the type of every stack entry must always be known. As a result, state-of-the-art copying collectors for Java [ADM98] restrict garbage collections to certain GC points, where the type of every entry in the stack is known. Therefore, if the new address of an object arrives at a processor between GC points, and there are potential references to this object from the stack, we will not know if we can safely update these potential references.

A possible solution is to update the stale references incrementally, using read-barriers or the virtual memory hardware. A read-barrier [Joh91] is a software check that verifies the validity of a reference before following it. By adding a read-barrier to every object access, all stale references will be updated as they are encountered. The downside of read-barriers is that their cost is present throughout the program execution, and is very expensive if implemented purely by software: with a read barrier we will have to compute the base address of the object being referenced (which may involve a hash table lookup), load the object header, and perform a compare to determine if this is an object that has been moved. We can also use the virtual memory hardware to detect memory accesses through stale references. All we need to do is to prohibit memory accesses to the fromspace. When a stale reference is followed, the access will trigger a protection fault. In the fault handler we can move the object being accessed to the tospace, and update the register holding the stale reference to the new address. The drawback of this scheme is that we may not be able to find out from which object this stale reference is loaded. Therefore the program may fault repeatedly at the old address until the object holding the stale reference is accessed and moved itself.

An alternative is to use immobile object handles, which are small object headers containing pointers to the objects. An object is referenced by the address of its handle, and all accesses to an object be indirect through the object’s handle. The use of handles eliminates the need to update references since handles do not move. In a conventional DSM system, the handles themselves should be allocated in the shared memory and used as a centralized location to find stale references. The drawback of this scheme is that a remote object takes two messages to fetch, one for the handle, and the other for the object. It is also an unsatisfactory solution for coarse-grain DSM systems in that the locality of the handles cannot be improved, therefore the programs will still suffer the problems caused by the poor spatial locality among the handles.
Compared with a non-generational copying collector, a generational copying collector may reduce the cost of write propagation and address update. This is because the objects in the older generations are collected less frequently than the young objects, therefore object movement occurs less than in a non-generational copying collector. However, as we will see later in this chapter, generational copying collectors still incur substantial costs.

**Delayed Memory Reuse.** A processor cannot reclaim the fromspace immediately after a copying garbage collection finishes, since the fromspace may still contain live objects to whom the processor is not the last writer. Even if the fromspace is empty, its reclamation should still be delayed until all remote processors have updated their stale references to the fromspace. Because of this delay, programs with high memory allocation rates may run out of memory because the reclamation rate cannot catch up with the allocation rate.

To justify the use of a copying garbage collector in a conventional DSM system, the gain in the sequential garbage collection time must outweigh the extra overhead it incurs.

### 2.2.2 Inter-processor Garbage Collection

In the case of the inter-processor garbage collector, state-of-the-art algorithms have focused on reducing the amount of data exchanged between the processors. To reduce the amount of communication, they limit the scope of the data exchanged, and propagate the data asynchronously. As a result, the set of cross-processor references on each processor is updated slowly, causing the retention of a large amount of garbage. This prevents the timely and accurate identification of garbage, and has a negative impact on the overall program performance.

The main drawback of the stub-scion method is that it may create a long chain of stub-scion pairs, causing a long delay between the time a object becomes garbage and the time it is reclaimed. In Figure 2.1, for example, object $O$ is allocated by processor $P_3$. A reference to $O$ is passed from $P_3$ to $P_2$, and $P_2$ passes the reference to $P_1$. This forms a chain of two stub-scion pairs. To reclaim object $O$ when it becomes garbage, $P_1$ must remove its stub to $O$ and communicate this fact to $P_2$ so that $P_2$ can remove its scion to $O$. The removal of the scion of $O$ on $P_2$ allows $P_2$ to remove its stub of $O$ in the subsequent intra-processor collection. When $P_3$ is told about $P_2$'s
removal of the stub to $O$, it will be able to remove its own scion for $O$ and reclaim $O$ in its subsequent intra-processor collection. This process takes two messages and two intra-processor collections on different processors to reclaim the object $O$. Generally speaking, in a system with $n$ processors, in the worst case there will be a chain of $n - 1$ stub-scion pairs for an object. For an object with a stub-scion chain of length $l$, it takes $l - 1$ messages before it can be reclaimed. In the rest of this thesis, we refer to the stub-scion chains as export chains.

Another drawback of the stub-scion method is that it does not identify garbage accurately. In particular, it cannot reclaim garbage cycles that span processors. Figure 2.2 illustrates a scenario where a cycle is formed. Assume we have a program with a doubly-linked queue. When two adjacent elements $t_1$ and $t_2$ in the queue are allocated by different processors, each of the elements will hold the address of the other. This puts them into their owners' scion tables, and the stub-scion algorithm will not be able to reclaim them. For example, when $N_1$ starts an intra-processor collection, $t_1$ will be found and traced since it is in $N_1$'s scion table. The reference

![Figure 2.2 Cycles of references.](image)
to $t_2$ will be found, so $N_1$ will keep $t_2$ in its stub table, and $N_2$ will not be able to remove $t_2$ from its scion table.

The weighted reference counting method allows earlier collection of garbage than the stub-scion method: instead of propagating the access information one step at a time along the export chain, it sends the information directly back to the creator. For a reference exported to $l$ processors, it still takes $l - 1$ messages to reclaim the object, but these messages can be in parallel. In contrast, in the stub-scion method, they are serialized (See Section 6.2 for the measurements that support this claim).

There are however three shortcomings of the weighted reference counting. First, the weights passed along with the references increase the amount of data communicated by the processors. Second, the solution proposed by Watson and Watson to deal with weight underflow (See Section 2.1.2) also results in export chains. Finally, like the stub-scion algorithm, it cannot reclaim cyclic garbage.

### 2.3 Intra-Processor Garbage Collectors for TreadMarks

We have implemented two intra-processor garbage collectors on TreadMarks that are representative of those in common use. The first collector is based on mark-sweep and the second one is based on generational copying. These collectors can be coupled with any inter-processor collection algorithm. In the following description, we use the weighted reference counting algorithm as an example to explain the interface between the intra-processor and inter-processor algorithms. Other algorithms will require only minor changes. The algorithms described in this section will also be used in the evaluation later in this chapter.

#### 2.3.1 The Mark-sweep Garbage Collector

In the mark-and-sweep collector for TreadMarks, the set of live objects is determined during the mark phase and the storage associated with dead objects is reclaimed during the sweep phase. The mark phase performs a depth-first traversal of the directed graph formed by the objects (vertices) and references (edges), marking each encountered object. The traversal starts at the set of "root" references and the exported references. During this traversal, memory coherence is disabled, eliminating a vast amount of potential communication. Roughly speaking, the only ill consequence of using an out of date copy of an object is that it may delay the reclamation of storage. (See Ferreira and Shapiro [FS94b] for a detailed explanation.) The sweep phase has
two parts. First, each locally created object is examined, and the storage used by
the unmarked ones is returned to the free pool. Second, each imported reference is
examined. If the object that it references was not marked, the object’s creator is
notified that the imported reference is no longer used.

2.3.2 The Copying Garbage Collector

Overview

The copying collector for TreadMarks has two generations. Following the suggestion
by Wilson and Moher [WM89], objects in the younger generation advance to the
older generation in every other garbage collection. Like Tarditi and Diwan’s collec-
tor [TD96], the old generation is included in a garbage collection if the size of the free
space is below 20% of the total heap size.

In the copying collector, each generation has several semispaces, and each semis-
pace consists of several memory pools, one for each processor. Each processor moves
objects only to its own tospace pool, and only reclaims its own pool in the fromspace
in the end. However, a processor may move an object from someone else’s fromspace
pool to its own tospace pool. At a barrier, each processor checks the amount of
the free memory left in its current memory pool. If the free memory drops below a
threshold, currently 20%, it signals other processors through the barrier. All proces-
sors start garbage collection immediately after the barrier. If a processor’s memory
pool is full before it reaches a barrier, it interrupts all other processors in the sys-
tem and forces them to suspend the computation, call a barrier, and start a garbage
collection. In short, in our algorithm for TreadMarks, the processors start garbage
collections synchronously but collect independently.

In a garbage collection, each processor starts a depth-first traversal from the “root”
references and the exported references, and scans every object found during the traversal
for references. A processor decides if it has the right to move each object according
to the last-writer-move policy, which is explained below. If an object is moved, a for-
warding pointer to its new location is written into its old address. References to the
moved object are updated only if the processor is also the last writer of the pages that
contain them. If the moved object is an imported object, its corresponding entry in
the import table is marked. Unlike in the mark-sweep collector for TreadMarks, the

\[^{\dagger}\text{The fact that an object is marked on some processor does not necessarily imply that a copy of the object exists on that processor, only that a reference to the object exists.}\]
memory coherency is not disabled during garbage collection. Therefore, the copying of the objects and the writing of the forwarding pointers may trigger page protection faults. Since each processor only writes to its tospace pool (new copies of the objects) or to the fromspase pages (forwarding pointers) to which it is the last writer, the writes do not cause communication.

After the traversal, every reference in the import table is examined. If an import reference is not marked, it is removed from the import table and sent back to the object's creator.

When the intra-processor collection is done, a processor returns to the normal program execution. There is no explicit synchronization between the processors at the end of the garbage collection. However, if a processor sends a data request to another processor which is still collecting, the request will be buffered, and served after that processor has completed the garbage collection.

**Determining the Last Writer**

In TreadMarks, we can only determine last writers on a per page basis. Therefore the last-writer-move policy means that only the last writer of a page is allowed to copy objects within the page. After the barrier, the last writer of every page is known to every processor. In the case of multiple concurrent writers to a page, an arbitration algorithm in the barrier will designate a single processor as the last writer and bring the page on the designated processor up to date. One ramification is that we do not allow small objects (smaller than a page) to cross page boundaries. The allocation routine also rounds up the size of large objects to a multiple of the page size. Since the last writer can only be determined at the page granularity, it is not accurate enough. As a result, a copying collector may not be able to improve the spatial locality. For example, consider a page p which contains 4 objects, $O_1$, $O_2$, $O_3$, and $O_4$. Processor $P_1$ is the last writer of $O_1$ and $O_2$, and processor $P_2$ is the last writer of $O_3$ and $O_4$. In a copying collection, we would like to have $P_1$ move $O_1$ and $O_2$ together in one page, and have $P_2$ move $O_3$ and $O_4$ together in another page. However, this is impossible since the copying collector on TreadMarks can only designate one processor as the last writer of all 4 objects. Therefore, all four objects will be moved together by one of the processors.

For example, suppose page p contains two objects $o_1$ and $o_2$. Processor $P_1$ writes $o_1$, then processor $P_2$ writes $o_2$. At the start of a garbage collection, $P_2$ may be
designated as the last writer of \( p \). However, \( P_2 \) may not have references to \( o_1 \), therefore it will not move \( o_1 \). \( P_1 \) will not move \( o_1 \) either, since it is not the last writer of page \( p \). In our current design we simply leave \( o_1 \) where it is until \( P_1 \) becomes the last writer of page \( p \) in a future collection. The arbitration mechanism guarantees that objects such as \( o_1 \) will eventually be moved by designating different processors as the last writer in different collections. As we have explained in Section 2.3.2, the reclamation of the fromspace will have to be delayed anyway, therefore we believe this solution is acceptable.

**Propagation of the Bogus Writes**

On TreadMarks, the bogus writes (i.e. movement of unchanged data) generate extra communication. For example, suppose an object \( o \) is copied to a new address by processor \( P_1 \). When processor \( P_2 \) later accesses \( o \) at its new address, it will fetch the entire object from \( P_1 \). Had the garbage collection not happened, however, \( P_2 \) would only fetch the recent diffs for \( o \), if any, from \( P_1 \). At first glance, it seems possible that we can manipulate the write-notices and the diffs so that \( P_2 \) can simply move its copy of \( o \) to its new address, fetch the diffs to update it, and go ahead using it. However, manipulating the diffs may not be cheap. For example, suppose object \( o \) was in page \( p_1 \), and has just been moved to page \( p_2 \). Also suppose that page \( p_1 \) has a diff that has not been fetched by all other processors in the system. In this case, we will have to examine \( p_1 \)’s diff to see if it covers any part of \( o \). If so, we must copy the relevant part to the corresponding diff created for page \( p_2 \). This has to be performed for every moved object on every diff in its old page that has not been propagated to all other processors. More importantly, to only fetch what it really needs in a tospace page, a remote processor must be able to construct the write-notices for that page. This means that the processor must know what objects that page contains. This will require that the address changes are eagerly propagated to all processors.

**Remote Reference Update**

After an intra-processor collection, the stale references on remote processors are updated incrementally: whenever a processor faults in the fromspace, it will fetch the page from its last writer, find the new address of the object being accessed, and update the register which holds the reference that has triggered the fault. The fault handler does not change the protection of the page so that other stale references to
the moved objects will be caught when they are followed. Note that if a fromspace page on a processor is clean, a read access to the page will not trigger a fault and the processor may read from the old copy of a moved object. This is not a problem, however, since the page being clean means that its content is consistent from the program's point of view. We can speed up the reference update process by invalidating the entire fromspace.

Reclaiming the Fromspace

The fromspace is reclaimed when all stale references have been updated on all processors. A processor resets the protection of the entire fromspace (including the memory pools owned by other processors) when it finds out that there are no export or import references to the fromspace. The protection for the pages in the locally owned memory pool is set to read-write, while the protection for other pages are set to invalid.

2.4 Evaluation

2.4.1 Methodology

A difficulty arises in selecting a programming language for our evaluation. For a variety of reasons, the most appealing programming language for our evaluation is Java. Unfortunately, the widely available implementations of Java are still much slower than C or C++. This would render our experiments largely meaningless, because inefficiencies in the Java implementation would dwarf the garbage collection cost. Perhaps more importantly, we expect efficient compiled versions of Java to become available soon, and we would expect that those be used in preference over the current implementations, quickly obsoleting our results.

We have therefore chosen to carry out the following experiments. We have taken existing C applications, and we have manually inserted code that registers the type of every pointer variable to the garbage collector. This approach represents the results that could be achieved by a Java language implementation. In other words, these experiments isolate the cost of the garbage collection from other aspects of the compilation and execution process.

We have implemented both collectors described in Section 2.3 on TreadMarks. Both collectors use the weighted reference counting algorithm for inter-processor garbage collection. Although we only used TreadMarks in our evaluation, we will
explain why the evaluation results also apply to other conventional DSM systems. As we will see in Chapter 6, our findings also apply to the stub-scion algorithm.

Our evaluation was conducted on a cluster of 32 300Mhz Pentium-II processors running the FreeBSD operating system. The processors are connected with a 100Mbps switched Ethernet. To avoid repetition, we only present the results on 32 processors here. More detailed measurement and analysis can be found in Chapter 6.

2.4.2 Applications

In the evaluation, we use three programs that are modified from real applications. The first program, Game, performs a game-tree search for a game called Othello. The second program, BH, is similar to Barnes-Hut, except that the tree cells are allocated anew in each iteration. The last program, MIP, solves a Mixed Integer Programming problem.

Game performs a game-tree search for a game called Othello. The program runs for several game steps. In each step, a master processor takes the current game board as the root of the game tree, and expands the tree for a predetermined number of levels. Each node in the tree has a back pointer to its parent node. After the master finishes expanding the tree, it puts the leaf nodes in a task queue. Then each processor repeatedly takes a task from the queue, computes the result, and writes the result into all ancestors of the task node, including the root node. The writes to the ancestor nodes are synchronized by a lock. At the end of each step, the master makes the best move, and the game tree is discarded.

We run the program for 20 steps. The size of the game tree generated in each step is around 256K bytes. Each processor also allocates a lot more private objects.

BH is similar to Barnes-Hut. It works on a number of particles. In each iteration, the particles are inserted into an oct-tree according to their positions by the master processor. Then every processor traverses the tree to find the particles assigned to it and performs the computation. There are 16K particles in the program. Unlike the original Barnes-Hut, the GC version of the program does not keep a list of tree cells that are reused each iteration. Instead, new tree cells are allocated by the master in the tree-building phases and discarded at the end of each iteration.

When running BH with the copying collector, we manually provide the collector with extra information so that the collector can identify the last writer at the object granularity. We provide this extra information because we want to identify the
copying collection cost that is common to all conventional DSM systems. Without this extra information, the copying collector will not be able to improve the spatial locality of the objects (See Section refsec:copying-design). This introduces false sharing in BH, and makes it difficult to isolate the cost of the bogus writes and remote reference updates. The performance reported later in this section is with this extra information.

**MIP** solves the Mixed Integer Programming problem [BCCL95], a form of linear programming in which many of the variables are restricted to have integer values. It uses branch and bound to find the optimal solution to the problem. Nodes in the search space are kept in a doubly-linked task queue. Each processor takes a node from this queue, performs some computation, perhaps generating new nodes, and puts these new nodes back into the queue. For each node, the computation involves "relaxing" the integer restrictions on the variables and solving the corresponding linear program to determine whether a better solution than the current best solution is possible below that node. This is repeated until the solution is found. This program allocates about 32K objects. All of them are shared.

Table 2.1 summarizes the applications. Table 2.2 presents the sequential running time and the garbage collection time under different garbage collection algorithms and manual memory management on TreadMarks. For the manual memory management, the GC time is the time spent by the `free()` call.

### 2.4.3 Measurement

Table 2.3 presents the total execution time for the three test programs on 32 processors. From the results we can see that garbage collection may reduce the overall program performance significantly. In Game and MIP, the overall program performance with the mark-sweep collector is up to 26% lower than that with manual

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Game</td>
<td>Game of Othello, 20 steps</td>
</tr>
<tr>
<td>BH</td>
<td>N-body simulation, 16K bodies.</td>
</tr>
<tr>
<td>MIP</td>
<td>Mixed Integer programming, 32K objects.</td>
</tr>
</tbody>
</table>

**Table 2.1** Program description and sequential performance.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Manual</th>
<th>Mark-Sweep</th>
<th>Copying</th>
</tr>
</thead>
<tbody>
<tr>
<td>Game</td>
<td>Time</td>
<td>282</td>
<td>295</td>
<td>286</td>
</tr>
<tr>
<td></td>
<td>GC (Free) time</td>
<td>17.5</td>
<td>31.4</td>
<td>21.7</td>
</tr>
<tr>
<td>BH</td>
<td>Time</td>
<td>27.4</td>
<td>28.0</td>
<td>27.6</td>
</tr>
<tr>
<td></td>
<td>GC (Free) time</td>
<td>0.10</td>
<td>0.70</td>
<td>0.30</td>
</tr>
<tr>
<td>MIP</td>
<td>Time</td>
<td>583</td>
<td>583</td>
<td>583</td>
</tr>
<tr>
<td></td>
<td>GC (Free) time</td>
<td>0.01</td>
<td>0.21</td>
<td>0.20</td>
</tr>
</tbody>
</table>

**Table 2.2** Running time and garbage collection time (in seconds) for TreadMarks on 1 processor. For manual memory management, GC time is the time spent in free.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Manual</th>
<th>Mark-Sweep</th>
<th>Copying</th>
</tr>
</thead>
<tbody>
<tr>
<td>Game</td>
<td></td>
<td>15.6</td>
<td>19.7</td>
<td>21.5</td>
</tr>
<tr>
<td>BH</td>
<td></td>
<td>7.58</td>
<td>8.14</td>
<td>6.22</td>
</tr>
<tr>
<td>MIP</td>
<td></td>
<td>50.1</td>
<td>57.1</td>
<td>70.3</td>
</tr>
</tbody>
</table>

**Table 2.3** Running time on 32 processors.

memory management. With the copying collector, the program performance is up to 40% lower.

The detailed statistics for Game on 32 processors is presented in Table 2.4. In Game, both collectors result in lower overall program performance than manual memory management. With the mark-sweep collector, the program performance is 26% lower (19.7 seconds versus 15.6 seconds) than with manual memory management. With the copying collector, the program performance is 38% lower (21.5 seconds versus 15.6 seconds). The inefficiency in the inter-processor garbage collection algorithm is the major factor that causes the lower performance of the garbage collected version of the program, while the extra overheads in the copying collector makes the overall program performance lower than that with the mark-sweep collector.

The inter-processor garbage collector, which uses the weighted-reference counting algorithm, fails to reclaim garbage quickly. As a result, the size of the available memory is reduced, and the cost of garbage collection increases.

In Game, the main data structure is the shared game tree. Every processor reads and modifies the root of the tree. The internal nodes of the tree are also shared by more than one processor. In the WRT collector, the original weight of the root
<table>
<thead>
<tr>
<th></th>
<th>Manual</th>
<th>Mark-Sweep</th>
<th>Copying</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec)</td>
<td>15.6</td>
<td>19.7</td>
<td>21.5</td>
</tr>
<tr>
<td>GC (Free) time (sec)</td>
<td>1.20</td>
<td>4.70</td>
<td>3.04</td>
</tr>
<tr>
<td>Total Data (MB)</td>
<td>31.0</td>
<td>36.8</td>
<td>60.5</td>
</tr>
<tr>
<td>Overlapped data requests</td>
<td>71.0</td>
<td>71.0</td>
<td>79.7</td>
</tr>
<tr>
<td>GC data (MB)</td>
<td>0</td>
<td>5.87</td>
<td>9.45</td>
</tr>
<tr>
<td>Suspend wait (sec)</td>
<td>0</td>
<td>0</td>
<td>0.44</td>
</tr>
<tr>
<td>Suspend data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Addr Update Msg</td>
<td>0</td>
<td>0</td>
<td>9.9K</td>
</tr>
<tr>
<td>Addr Update Data (MB)</td>
<td>0</td>
<td>0</td>
<td>1.2M</td>
</tr>
</tbody>
</table>

**Table 2.4** Detailed statistics for Game on 32 processors.

Reference quickly underflows, and the new weights created by the non-owner processors result in long export chains (See Section 2.2.2). We have observed that the export chains of the root reference has 7 links on average. When running on 32 processors, each processor on average performs less than one intra-processor collection in each iteration. Therefore it takes at least 7 iterations before the root can be collected on the master processor. This delay increases the size of the live data and the intra-processor garbage collection cost. The generational collector does not help because the live data size is large enough so that the older generation is included in almost every garbage collection.

Figure 2.3 shows the size of the retained garbage versus the size of the allocated data with manual memory management. With manual memory management, the allocated data refers to the shared objects that have been allocated but not yet freed. With garbage collection, the retained garbage refer to the objects that are already dead but are still treated as live by the inter-processor collector. In Game they are the objects allocated by the master processor that are reachable from the exported references, but unreachable from the master's local roots. We measured the size of the retained garbage at 10 points during the program execution. From the figure we can see that with garbage collection the size of the retained garbage is several times larger than with manual memory management. The size of the retained garbage is solely the result of the inter-processor collection algorithm. It stays the same with either the mark-sweep or the copying intra-processor collector. As we will see in Chapter 6, the problem of delayed reclamation is more serious with the stub-scion algorithm.
Figure 2.3  Retained garbage in Game with the WRT collector.

With the copying collector in Game, the program performance is lower than with the mark-sweep collector even though the copying collector spends less time in intra-processor garbage collection. This underperformance results from the extra cost to perform copying collections on TreadMarks.

To synchronize the move of the objects using the last-writer-move policy, TreadMarks must suspend the system and determine a last writer for each page at the beginning of each intra-processor collection. On 32 processors, the total suspension time is 0.44 seconds, accounting for 7.5% of the performance difference between the copying collector and manual memory management. This is the sum over 20 iterations. No data is exchanged during the suspension because in this program all writes to the shared objects are synchronized by a lock. Therefore there is a unique last writer for each page at the beginning of each copying collection, and the arbitration mechanism (See Section 2.3.2) is not invoked.

The impact of the bogus writes is reflected in the total amount of data communicated. From the tables we can see that with the copying collector, the program sends much more data than with the mark-sweep collector. On 32 processors, after subtracting the GC data from the total, the copying collector sends 49.8M bytes of program data, while the mark-sweep collector sends only 31.0M bytes of program data. This is a 61% difference. The address updates involve 9900 messages and 1.2M bytes of data. It is difficult to isolate the effect of each of these factors on the run-
ning time. Together they account for 38% of the performance difference between the copying collector and manual memory management, after excluding the suspension cost and the intra-processor garbage collection time.

The detailed statistics for BH on 32 processors is presented in Table 2.5. With the mark-sweep collector, the overall program performance is 7.4% lower than that of manual memory management. With the copying collector, the overall program performance is 22% higher than that of manual memory management. As we have said in Section 2.4.2, we manually supplied the copying collector with extra information so that the copying collector can identify the last writer of each object. Without this extra information, the performance of the copying collector on TreadMarks is about the same (within 2%) as that of the mark-sweep collector on TreadMarks.

Unlike in Game, the inter-processor garbage collection algorithm does not delay the reallocation of the shared objects in BH: In BH the master is the only processor that writes the tree cells. As a result, all other processors import the object references directly from the master, and there are no export chains.

The cost of copying collection in BH is smaller than in Game because it avoids the costs of suspension, the bogus writes, and the address updates.

In BH, the copying collector on TreadMarks does not pay extra cost to suspend the system: BH is a barrier based program, and the program only performs one garbage collection in each iteration. Therefore the copying collections on TreadMarks can start after one of the barriers in each iteration, avoiding the suspension cost.

The copying collector does not suffer from bogus writes in BH. This is because the program performs one garbage collection in each iteration (i.e., an object moves

<table>
<thead>
<tr>
<th></th>
<th>Manual</th>
<th>Mark-Sweep</th>
<th>Copying</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec)</td>
<td>7.58</td>
<td>8.14</td>
<td>6.22</td>
</tr>
<tr>
<td>GC (Free) time (sec)</td>
<td>0.10</td>
<td>0.70</td>
<td>0.30</td>
</tr>
<tr>
<td>Total Data (MB)</td>
<td>155M</td>
<td>162M</td>
<td>56.9M</td>
</tr>
<tr>
<td>Overlapped data requests</td>
<td>93.0K</td>
<td>93.0K</td>
<td>47.0K</td>
</tr>
<tr>
<td>GC data</td>
<td>0</td>
<td>3.40M</td>
<td>3.40M</td>
</tr>
<tr>
<td>Suspend wait (sec)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Suspend data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Addr Update Msg</td>
<td>0</td>
<td>0</td>
<td>3.02K</td>
</tr>
<tr>
<td>Addr Update Data (MB)</td>
<td>0</td>
<td>0</td>
<td>1.24</td>
</tr>
</tbody>
</table>

Table 2.5 Detailed statistics for BH on 32 processors.
at most once in each iteration), and all objects are completely overwritten in each iteration. Because of generational approach, the address updates do not seriously affect the program performance either. In BH, objects can be divided into two distinct categories. The particles are long-lived while the internal cells of the tree do not live beyond one iteration. With the generational collector, the particles will be advanced to the older generation very soon while the internal tree cells are limited to the younger generation. Once in the older generation, most of the particles do not move. The few particles that do move are those whose writers have changed and have been copied from one processor's memory range to another processor's memory range. But the number of such objects are small. Although the tree cells are moved by the master processor during every tree-building phase, the slave processors do not traverse the tree before the complete tree has been built. Therefore they do not see the old addresses of the tree cells. As a result, the address update cost on TreadMarks is low. It only involves 3000 messages totaling 1.2M bytes.

In our measurement, the cost of the address updates is reduced by the generational copying collector. With a non-generational garbage collector, the particles would move in every garbage collection, significantly increasing the address update cost. On 32 processors, a non-generational garbage collector would increase the address update cost to 24 thousand messages and 6.9M bytes of data, and increase the total execution time to 6.90 seconds. This is a 15% slowdown from using the generational collector in terms of total execution time.

With the manual optimization, the copying collector results in better overall program performance than manual memory management because it improves the spatial locality of the shared objects. In BH, the particles are randomly generated in the initialization phase, but are assigned to the processors according to their physical locations in a 3-dimensional space. After initialization, each particle is only written by the processor to which it is assigned. And the set of particles that are assigned to each processor changes little between iterations. With manual memory management, the particles are placed in memory according to the order of their creation in the initialization phase. As a result, particles in the same page are often assigned to many processors, resulting in significant false sharing. With the copying collector and the manual optimization, all particles assigned to the same processor are moved together, eliminating false sharing.

The detailed statistics for MIP is presented in Table 2.6. With the mark-sweep collector, the program performance is 14% lower than with manual memory manage-
<table>
<thead>
<tr>
<th></th>
<th>Manual</th>
<th>Mark-Sweep</th>
<th>Copying</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec)</td>
<td>50.1</td>
<td>57.1</td>
<td>70.3</td>
</tr>
<tr>
<td>GC (Free) time (sec)</td>
<td>0.00</td>
<td>0.21</td>
<td>0.20</td>
</tr>
<tr>
<td>Total Data (MB)</td>
<td>131M</td>
<td>135M</td>
<td>203M</td>
</tr>
<tr>
<td>Overlapped data requests</td>
<td>152K</td>
<td>175K</td>
<td>228K</td>
</tr>
<tr>
<td>GC data</td>
<td>0</td>
<td>2.26</td>
<td>2.26</td>
</tr>
<tr>
<td>Suspend wait (sec)</td>
<td>0</td>
<td>0</td>
<td>0.05</td>
</tr>
<tr>
<td>Suspend data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sync global GC time (sec)</td>
<td>0</td>
<td>6.60</td>
<td>6.60</td>
</tr>
<tr>
<td>Sync global GC data (MB)</td>
<td>0</td>
<td>0.79</td>
<td>0.79</td>
</tr>
<tr>
<td>Sync global GC msg</td>
<td>0</td>
<td>22.8K</td>
<td>22.8K</td>
</tr>
<tr>
<td>Addr Update Msg</td>
<td>0</td>
<td>0</td>
<td>51.2K</td>
</tr>
<tr>
<td>Addr Update Data (MB)</td>
<td>0</td>
<td>0</td>
<td>3.20M</td>
</tr>
</tbody>
</table>

Table 2.6  Detailed statistics for MIP on 32 processors.

ment. With the copying collector, the program performance is 40% lower than with manual memory management.

Garbage collection results in lower program performance in MIP because the default inter-processor collection algorithm cannot reclaim cyclic garbage. The main data structure in MIP, the task queue, is a doubly-linked list. With every processor creating new task nodes and inserting them into the queue, the task nodes form cross-processor cycles. The default inter-processor garbage collector, which uses the weighted reference counting algorithm, cannot reclaim cyclic garbage. As a result, the program has to invoke costly synchronous collections to reclaim cyclic garbage. From Table 2.6 we can see that the time spent in the synchronous collections almost equals the performance difference between the mark-sweep collector and manual memory management.

With the copying collector, the overall program performance is much lower than with the mark-sweep collector. This is because of the additional overhead in the copying collector.

The suspension cost for TreadMarks' copying collector is small in MIP because the program only performs a few collections during the entire program execution. For the same reason as in Game, there is no data exchange during the suspensions.

The major cause for the copying collector's underperformance is the bogus writes. From the table we can see that with the copying collector the total data communicated is 55% more than that with manual memory management. It is also 50% more than
that of the mark-sweep collector. The address updates also have non-trivial cost, sending 51.2K messages. This is because MIP only performs two garbage collections and the objects have not advanced to the older generation when the program ends. Therefore all live objects move during the garbage collections.

2.4.4 Summary

We have evaluated the performance of two DSM garbage collectors on TreadMarks using three programs that have been modified from real applications. For the intra-processor garbage collection, one of the collectors uses a mark-sweep algorithm, the other uses a generational copying algorithm. For the inter-processor garbage collection, both collectors use a weighted reference counting algorithm.

The problem in the intra-processor collection on conventional DSM systems is that the implementation of the garbage collector affects the DSM system performance. A mark-sweep collector reduces the spatial locality of the program, increasing the amount of communication. A copying collector improves the spatial locality, but the object movement and reference updates must be propagated across the system, also increasing the amount of communication and reducing the overall program performance.

The problem in the inter-processor collection is that state-of-the-art algorithms limit the scope of the garbage collector data exchanged between the processors. Therefore each processor does not have accurate knowledge of the set of cross-processor references and has to be conservative in identifying garbage. Because of the conservatism, the reclamation of the acyclic garbage is delayed, and the cyclic garbage often cannot be reclaimed at all.

The results of TreadMarks should apply to other coarse-grain DSM systems as well. Here we make an educated guess on how the fine-grain systems will be affected.

The poor spatial locality as a side effect of the mark-sweep collectors only increases the communication cost in coarse-grain systems. The fine-grain systems are immune to this problem since they do not perform page-based aggregation. However, as our measurement has shown, the performance of the mark-sweep collectors may be inferior to that of the copying collectors. If the fine-grain systems limit themselves to mark-sweep collectors, they are putting themselves at a disadvantage.

In the case of the copying garbage collector, our measurement has shown that the major cost in the copying collector on TreadMarks is the bogus writes and the address
updates. Both costs are not limited to TreadMarks, they exist in all conventional DSM systems, including the fine-grain DSM systems. Therefore, our results also apply to other conventional DSM systems.

In this chapter, we only evaluated the weighted reference counting algorithm. As we will see in Chapter 6, our findings also apply to the stub-scion algorithm.

2.5 Solutions

On conventional DSM systems that support sharing on untyped memory regions, the implementation of the intra-processor garbage collectors is not orthogonal to the DSM implementation. As a result, the intra-processor garbage collection can reduce the performance of the DSM system. To solve this problem, we propose to design a new DSM system that uses the shared object space abstraction. In the new DSM system, objects are identified by globally unique object ids instead of virtual addresses. The consistency information exchanged between the processors is also organized by objects rather than by virtual addresses. The shared object space abstraction allows the DSM system to distinguish bogus writes from the writes made by the program, and eliminates the need for address updates. Therefore, the memory layout and address changes on one processor are transparent to the remote processors. And the implementation of the intra-processor garbage collector is orthogonal to the DSM implementation.

The root cause of the problems in state-of-the-art inter-processor collection algorithms is that the processors do not exchange enough information. Therefore none of the processor has complete knowledge of the object graph, and has to maintain an inaccurate estimate of the set of live, cross-processor references, resulting in inaccurate garbage identification. The problem with gathering complete liveness information from the entire system is that the size of such information can be prohibitively large in the worst case. However, we believe that with many data structures commonly used in real applications, the size of such information is actually small. Therefore, we propose to design a new adaptive algorithm. In this new algorithm, the processors exchange more information in the normal mode of execution so that the entire object graph can be constructed on one processor. This will allow more accurate garbage identification and more timely garbage reclamation. When the size of the information to be exchanged becomes too large, our algorithm can fall back to one of the state-of-the-art algorithms.
2.6 Summary

A garbage collector for DSM systems can be divided into two parts: an intra-processor collector that performs the traversal and reclaims the garbage; and an inter-processor collector that maintains the set of cross-processor references on each processor. In this chapter we have identified the problems in state-of-the-art algorithms and presented our solutions.

The problem in the intra-processor collection on conventional DSM systems is that the implementation of the garbage collector affects the DSM system performance. For example, a mark-sweep collector reduces the spatial locality of the program, increasing the amount of communication. A copying collector improves the spatial locality, but the object movement and reference updates must be propagated across the system, also increasing the amount of communication and reducing the overall program performance. The root of the problem is that conventional DSM systems supporting sharing on untyped memory regions do not provide a clean interface to the garbage collector. We propose to use the shared object space abstraction to solve this problem. With the shared object space, the memory layout and address changes on one processor are transparent to the remote processors. Therefore, the implementation of the intra-processor garbage collector is orthogonal to the DSM operations.

The problem in the inter-processor collection is that state-of-the-art algorithms limit the scope of the garbage collector data exchanged between the processors. Therefore each processor does not have accurate knowledge of the set of cross-processor references and has to be conservative in identifying garbage. Because of the conservatism, the reclamation of the acyclic garbage is delayed, and the cyclic garbage often cannot be reclaimed at all. We propose that the processors exchange more information so that a complete object graph can be built on one processor, allowing more accurate garbage identification and more timely reclamation. We argue that in the common case the benefits of our algorithm outweighs the increased amount of communication, and in the worst case in which the amount of extra GC data is too large, our algorithm can seamlessly fall back to one of the state-of-the-art algorithms.
Chapter 3

Design and Implementation of DOSA

This chapter describes the design and implementation of DOSA, which supports distributed sharing of objects in modern programming languages, such as Java and Modula-3. DOSA is motivated by our desire to support more efficient intra-processor garbage collection. However, we find that the run-time type information and the safety features in modern programming languages also allow us to use implementation techniques that support both coarse-grained and fine-grained sharing patterns efficiently, benefiting non-garbage-collected programs as well.

Taking advantage of languages such as Java, DOSA efficiently implements the abstraction of a shared space of objects. The key implementation technique is the use of "handles", which allows the system to keep track of memory accesses at the object granularity, efficiently supporting fine-grained sharing. DOSA also performs various optimizations, including data aggregation to support coarse-grained sharing, and delayed storage allocation to improve the spatial locality of programs with either sharing patterns.

Section 3.1 presents an overview of DOSA, and Section 3.2 presents its programming model. We describe the design of DOSA in Section 3.3, and present the implementation details in Section 3.4. In Section 3.5 we explain how existing compiler optimization techniques can be used to reduce the cost of the handle dereferences. Section 3.6 describes the language support required by DOSA. The related work is examined in Section 3.7. Finally, Section 3.8 summarizes this chapter.

3.1 Introduction

DOSA provides support for distributed sharing of objects in modern programming languages. DOSA requires that the programming language provide sufficient type information at run-time, so that it allows an unambiguous determination of whether a location contains an object reference or not. In addition, in the case of a reference, the type and size of its referent must be known at run-time. DOSA also assumes
that the programs written in such languages behave safely, e.g., an object access cannot go beyond the end of the object. Many modern languages fall under this category, including Java and Modula-3. Unlike purely object-oriented languages, like e.g., Orca [BKT92, BBH+98], we do not restrict access to occur solely through method invocation. Direct access through a reference to object data is supported.

The key insight is that the run-time type information and the safety features allow efficient and transparent sharing of data with both fine-grained and coarse-grained access patterns. In contrast, conventional distributed shared memory (DSM) systems that do not take advantage of modern programming languages are limited to providing only one granularity with good performance. Indeed, DSM systems have been divided into those offering support for coarse-grained sharing or for fine-grained sharing. Coarse-grain sharing systems are typically page-based, and use the virtual memory hardware for access and modification detection. Although relaxed memory models and multiple-writer protocols relieve the impact of the large page size, fine-grained sharing and false-sharing remain problematic. Throughout this chapter, we will use TreadMarks [ACD+96] as the representative of such systems, but the results apply to similar systems. Fine-grain sharing systems typically augment the code with instructions to detect reads and writes, freeing them from the large size of the consistency unit in virtual memory-based systems, but introducing per-access overhead that reduces performance for coarse-grained applications. In addition, these systems do not benefit from the implicit aggregation effect present in the page-based systems. Fine-grain systems typically require a message per object, while page-based systems bring in all data in a page at once, avoiding additional messages if the application accesses other objects in the same page. Again, in this chapter we will use a single system, Shasta [SGT96], to represent this class of systems, but the discussion applies to similar systems.

3.2 API and Memory Model

3.2.1 API

The general model is a shared space of objects, in which each reference to an object is typed. The programmer is responsible for creating and destroying threads of control, and for the necessary synchronization to insure orderly access by these threads to the object space. Various synchronization mechanisms may be used, such as semaphores, locks, barriers, monitors, etc. No special API is required in languages with suitable
typing and multithreading support, such as Java or Modula-3. Unlike Orca, we
do allow references to be used for accessing objects. We do not require a method
invocation for each access.

Objects are considered the unit of sharing. In other words, an individual object
must not be concurrently written by different threads, even if those threads write
different data items in the object. If two threads write to the same object, they should
synchronize their writes. Arrays of objects may, however, have multiple concurrent
writers. This is in particular true for arrays of scalars. Of course, for correctness, the
different processes must write to disjoint elements in the array.

The single-writer nature of individual objects is not inherent to the design of our
system, but we have found that it corresponds to common usage, and is therefore not
restrictive. As will be seen in Section 3.4, it allows us to use an efficient single-writer
protocol for individual objects.

3.2.2 Memory Model: Release Consistency

The object space is release consistent. Release consistency (RC) [GLL+90] is a relaxed
memory consistency model. In RC, ordinary accesses to shared data are distinguished
from synchronization accesses, with the latter category divided into acquires and
releases. An acquire roughly corresponds to a request for access to data, such as a lock
acquire, a wait at a condition variable, or a barrier departure. A release corresponds to
the granting of such a request, such as a lock release, a signal on a condition variable,
or a barrier arrival. RC requires ordinary shared memory updates by a processor \( p \) to
become visible to another processor \( q \) only when a subsequent release by \( p \) becomes
visible to \( q \) via some chain of synchronization events. Parallel programs that are
properly synchronized (i.e., have a release-acquire pair between conflicting accesses
to shared data) behave as expected on the conventional sequentially consistent shared
memory model.

3.3 Design

3.3.1 Handles

The key technique behind the design of DOSA is the use of handles. A handle is an
object header that is separate from the object body, and contains a pointer to the
address of the object.
Consider a (single-processor) implementation of a programming language using a handle table (see Figure 3.1). Each object in the language is uniquely identified by an object identifier (OID) that also serves as an index into the handle table for that object. All references to an object refer in fact to its entry in the handle table, which then in turn points to the object proper. In such an implementation, it is easy to relocate objects in memory. It suffices to change the corresponding entry in the handle table. No other changes need to be made, since all references are indirect through the handle table.

Extending this simple observation allows an efficient distributed implementation of these languages. Specifically (see Figure 3.2), a handle table representing all shared objects is present on each processor. A globally unique OID identifies each object, and serves as an entry in the handle tables. As before, each handle table entry contains a pointer to the location in memory where the object resides on that processor. The consistency protocol can then be implemented solely in terms of OIDs, because these are the only references that appear in any of the objects. Furthermore, the same object may be allocated at different virtual memory addresses on different processors. It suffices for the handle table entry on each processor to point to the proper location. In other words, although the programmer retains the abstraction of a single object space, it is no longer the case that all of memory is virtually shared, and that all objects have to reside at the same virtual address at all processors, as is the case in both TreadMarks and Shasta. With the shared object abstraction, objects can also be locally re-arranged in memory, for instance to improve cache locality or during garbage collection, without affecting the other processors.
Logical Implementation

Handle Table

Heap

Processor 1

Processor 2

Figure 3.2 Shared objects identified by unique OIDs.

In order to provide good performance for coarse-grained applications, we continue to use the virtual memory system for access detection, thereby avoiding the overhead of instrumentation. Fine-grain access using VM techniques is then provided as follows. Although only a single physical copy of each object exists on a single processor, each object can be accessed through one of three VM mappings. All three point to the same physical location in memory, but with three different protection attributes: invalid, read-only, or read-write. A change in access mode is accomplished by switching between the different mappings for that object only. The mappings for the other objects in the same page remain unaffected. Consider the example in Figure 3.3. A page contains four objects, one of which is written on a different processor. This modification is communicated between processors through the consistency protocol, and results in the invalid mapping being set for this object. Access to other objects can continue, unperturbed by this change, thus eliminating false sharing between objects on the same page.

3.3.2 Lazy Object Storage Allocation

With the shared object space abstraction in DOSA, an object can be placed at different addresses by different processors. The benefit of this is that on a particular processor, memory needs to be allocated only for those objects that are accessed on that processor, resulting in a smaller memory footprint and better cache locality.
Figure 3.3 Access detection using the handle pointers.

N-body simulations illustrate this benefit. Each processor typically accesses its own bodies, and a small number of "close" bodies on other processors. With global allocation of memory, the remote bodies are scattered in memory, causing lots of misses, messages, and – in the case of TreadMarks – false sharing. Here, in contrast, only the local bodies and the locally accessed remote bodies are allocated in local memory. As a result, there are far fewer misses and messages.

3.3.3 Data Aggregation

To achieve good program performance, especially for coarse-grained applications, DOSA performs data aggregation. When a fault is detected on an object in a particular page, all invalidated objects in the same page as the faulted object are brought up-to-date.

Although data aggregation potentially re-introduces false sharing, its harmful effects are much smaller than in a conventional page-based system. There are three reasons for this. First, we are free to co-locate or not to co-locate certain objects in a page on a per-processor basis. Second, with lazy object storage allocation, the spatial locality of the data on each processor is much better than in a conventional page-based system. This means that the objects in the same page as the faulted object are likely to be accessed too. Returning to the N-body application, the location of bodies typically changes slowly over time, and a given processor accesses many of the same bodies from one iteration to the next. Thus, bringing in all bodies in the same page on the first access miss to any one of them is beneficial. Finally, the messages to fetch the different objects in the page are sent out in parallel. Therefore their latencies and the latencies of their replies are largely overlapped. In the rest of this thesis, we call the overlapped message round-trips for a page a message round.
3.3.4 Potential Overheads

While there are many apparent performance benefits from the handle-based design, there are some obvious questions about the performance of such a system as well. For instance, the extra indirection is not free, and consistency information now needs to be communicated per-object rather than per-page, potentially leading to a large increase in its size. To evaluate these tradeoffs, we have implemented the system outlined above, and compared its performance to that of TreadMarks.

3.4 Implementation

We focus on the consistency maintenance of individual objects. Synchronization is implemented as in TreadMarks.

3.4.1 Consistency Protocol

DOSA uses a single-writer, lazy invalidate protocol to maintain release consistency. The lazy implementation delays the propagation of consistency information until the time of an acquire. At that time, the releaser informs the acquiring processor which objects have been modified. This information is carried in the form of write notices.

The protocol maintains a vector timestamp on each processor, the i-th element of which records the highest interval number of processor i that has been seen locally. An interval is an epoch between two consecutive synchronization operations. The interval number is simply a count of the number of intervals on a processor. Each write notice has an associated processor identifier and vector timestamp, indicating where and when the modification of the object occurred. A processor sends its vector timestamp on an acquire, and the responding processor sends only those write notices with a vector timestamp between the received vector timestamp and its own current vector timestamp.

Arrival of a write notice for an object causes the acquiring processor to invalidate its local copy, and to set the last writer field in the handle table entry to the processor identifier in the write notice. A processor incurs a page fault on the first access to an invalidated object, and obtains an up-to-date version of that object from the processor indicated in the last writer field.

In DOSA, the write notices are in terms of objects. As a consequence, the number of write notices can potentially be much larger than in a page-based DSM for very
fine-grained applications. To this end, DOSA employs a novel compression technique
to reduce the number of write notices transmitted during synchronizations.

Each time a processor creates a new interval, it traverses in reverse order old
intervals that it has created before and looks for the one that consists of similar write
notices. If such a "match" is found, the difference between the new interval and the old
interval is presumably much smaller than write notices themselves. The processor can
then create and later transmit when requested only the write notices that are different
from those of the matched old interval, and thus reduce the consistency data. Since
intervals are always received and incorporated in the forward order, when a processor
receives such an interval containing difference of write notices, it is guaranteed to
have already received the old interval based on which the diff of the new interval is
made. It can then easily reconstruct the write notices of the new interval.

3.4.2 Data Structures

A handle table is present on each processor. The handle table is indexed by a globally
unique object identifier (OID). Each entry in the handle table contains the corres-
ponding object's address in local virtual memory. This address may be different from
processor to processor. The object's local state, i.e., invalid, read-only, or read-write,
is also reflected in the handle table entry through different mappings of the object's
local virtual address with the corresponding protection attributes (see Section 3.4.4).
The handle table entry contains a last writer field, indicating from which processor
to fetch an up-to-date copy of the object on an access miss. Finally, a handle table
entry contains a field linking it with other objects allocated in the same page.

A few auxiliary data structures are maintained as well. An inverse object table,
implemented as a hash table, is used by the page fault handler to translate a faulting
address to an OID. Each processor maintains a per page linked list of objects allo-
cated in that page. This list is used to implement communication aggregation (see
Section 3.4.6). Finally, each processor maintains its vector timestamp and an efficient
data structure for sending write notices when responding to an acquire.

As a practical matter, OIDs are currently assigned as the virtual addresses of the
entry in the handle table. Therefore, the handle table must reside at the same virtual
address on all processors. Should this ever become a restriction, it could easily be
removed.
Objects are instantiated by a new operation or the equivalent. An OID is generated, and memory is allocated on the local processor to hold the object. In order to minimize synchronization overhead for unique OID generation, each processor is allocated a large chunk of OIDs at once, and this chunk allocation is protected by a global lock. Each processor then independently generates OIDs from its chunk.

3.4.3 Object Storage Allocation

The ability to allocate objects at different addresses on different processors suggests that we can delay the storage allocation for an object on a processor until that object is first accessed by that processor. We call this optimization lazy object storage allocation.

Lazy object storage allocation is implemented as follows. Each processor reserves an address range called the shadow area. The access permission for all pages in the shadow area is set to invalid. Whenever a processor fetches an object from a remote processor, it checks the object for references. If the processor finds a reference that has not been seen before, storage is allocated for the referent in the shadow area. A fault will be triggered when the referent is accessed by the processor for the first time. At this time, DOSA will allocate storage for this object in the heap, and free its storage in the shadow area.

3.4.4 Switching Protection

DOSA relies on hardware page protection mechanism to detect accesses to invalid objects and write accesses to read-only objects. We create three non-overlapping virtual address regions that map to the same physical memory, from where shared objects are allocated. An object thus can be viewed through any of the three corresponding addresses from the three mappings. DOSA assigns the access permissions to the three mappings to be invalid, read-only, and read-write, respectively. During program execution, it regulates accesses to a shared object by adjusting the object’s handle to point to one of the three mappings. In addition to providing per-object access control, this approach has the substantial additional benefit that no kernel-based memory protection operations are necessary after the initialization of all mappings.

As a practical matter, the three mappings of shared memory region differ in two leading bits of their addresses. Therefore, changing protection is a simple bit masking operation.
3.4.5 Modification Detection and Write Aggregation

On a write fault, we make a copy (a twin) of the page on which the fault occurred, and we make all objects in the page read-write. At a (release) synchronization point, we compare the modified page with the twin to determine which objects were changed, and hence for which objects write notices need to be generated\(^1\). After the (release) synchronization, the twin is deleted and the unchanged objects in the page are made read-only again.

This approach has better performance than the more straightforward approach, where only one object at a time is made read-write. The latter method generates a substantially larger number of write faults. If there is locality to the write access pattern, the cost of these write faults exceeds the cost of making the twin and performing the comparison (see Section 5.4.3). We refer to this optimization as write-aggregation.

3.4.6 Access Miss Handling and Read Aggregation

When a processor faults on a particular object, if the object is smaller than a page, it uses the list of objects in the same page (see Section 3.4.2) to find all of the invalid objects residing in that page. It sends out concurrent object fetch messages for all these objects to the processor recorded in the write notice for each object.

By doing so, we aggregate the requests for all objects in the same page. This approach performs better than simply fetching one faulted object at a time. There are two fundamental reasons for this phenomenon.

1. If there is some locality in the objects accessed by a processor, then it is likely that the objects allocated in the same page are going to be accessed closely together in time. Here, again, the local object storage allocation works to our advantage. It is true that some unnecessary data may be fetched, but the effect of that is minimal for the following reason.

2. With read aggregation as described above, the messages to fetch the different objects go out in parallel, and therefore their latencies and the latencies of the replies are largely overlapped.

\(^1\)The twin is used here for a different purpose than the twin in TreadMarks. Here it is simply used to generate write notices. In the TreadMarks multiple-writer protocol it is used to generate a diff, an encoding of the changes to the page. Since we are using a single-writer protocol, there is no need for diffs.
If an object is larger than a page, we fall back to a page-based approach. In other words, only the page that is necessary to satisfy the fault is fetched.

3.4.7 Summary

We summarize with a discussion of the salient differences between DOSA on one hand, and TreadMarks and Shasta on the other hand.

DOSA shares with TreadMarks its use of invalidate-based lazy release consistency, its use of the VM system for access and write detection, and its page-based aggregation. It differs in that it allocates storage for shared data locally, rather than globally, it performs per-object rather than per-page access and write detection, and it uses a single-writer protocol per object rather than a multiple-writer protocol per page.

Shasta uses an invalidate-based eager release consistency. More importantly, it differs from DOSA in that it uses global rather than local memory allocation. It uses instrumentation rather than the VM system for access and write detection. It does access and write detection on a per “cache line” basis, where the cache line is implemented in software and can be varied from program to program. There is no attempt to aggregate data.

3.5 Compiler Optimizations for Coarse-grained Applications

The extra indirection is a source of concern for applications that access large arrays, because the extra indirection for each array access may cause significant overhead, without any gain in return from better support for fine-grained sharing. Here again, however, we can take advantage of the type information in the language, this time by exploiting the type information at compilation time.

Consider, for example, a C program with a two-dimensional array of scalars, such as float, that is implemented in the same fashion as a two-dimensional Java array of scalars. In other words, an array of pointers to an array of the scalar type, i.e.,

    scalar_type **a;

This program performs a regular traversal of the array with a nested for loop. First, consider the TreadMarks program. Here, the accesses to the two-dimensional array take the form of

for i
    for j
        ... = a[i][j];
In general, a C compiler cannot further optimize this loop nest, because it cannot prove that `a` and `a[i]` do not change during the loop execution\(^5\). In a language with sufficient type information, however, `a`, `a[i]` and `a[i][j]` are of different types, and therefore the compiler can easily determine that `a` and `a[i]` do not change, and transform the loop accordingly to

```c
for i
{
  p = a[i];
  for j
    ... = p[j];
}
```

resulting in a significant speedup. In the DOSA program the original program takes the form of

```c
for i
  for j
    ... = a->handle[i]->handle[j];
```

which, in a language such as Java can be similarly transformed to

```c
for i
{
  p = a->handle[i];
  for j
    ... = p->handle[j];
}
```

This transformation is valid if there are no synchronization points in the inner loop, because with lazy release consistency data is only invalidated at synchronization points. While offering much improvement, this transformation still leaves the DOSA program at a disadvantage compared to the optimized TreadMarks program, because of the remaining pointer dereferencing in the inner loop. Observe also that the following transformation of the DOSA program is legal but not profitable:

```c
for i
{
  p = a->handle[i]->handle;
```

\(^5\)Some C compilers support a `#pragma` or command line option that enables the programmer to make assertions about pointer aliasing that the compiler cannot determine automatically.
for j
    ... = p[j];
}

The problem with this transformation occurs when a->handle[i]->handle has been invalidated as a result of a previous synchronization. Before the j-loop, p contains an address in the invalid region, which causes a page fault on the first iteration of the j-loop. The DSM run-time system changes a->handle[i]->handle to its location in the read-write region, but this change is not reflected in p. As a result, the j-loop page faults on every iteration.

One alternative is to dynamically re-write the value of p during execution. Another alternative, which is the one we use, is to perform a slightly different transformation, by touching a->handle[i]->handle[0] before assigning it to p. In other words,

for i
{
    touch( a->handle[i]->handle[0] );
    p = a->handle[i]->handle;
    for j
        ... = p[j];
}

Touching a->handle[i]->handle outside the j-loop causes the fault to occur there, and a->handle[i]->handle to be changed to the read-write location. The same optimization can be applied to the outer loop as well, resulting in

touch( a->handle[0] );
q = a->handle;
for i
{
    touch( q->handle[0] );
    p = q->handle;
    for j
        ... = p[j];
}

This optimization is essentially loop invariant analysis, and can be carried out with well-understood compiler technology. The only twist is that a "cached" handle, such as p or q above, cannot be reused across a synchronization point, when new invalidation messages may arrive.
3.6 Language Support

DOSA requires that the language provide sufficient type information at run time. At the minimum, typing must be strong enough so that it allows an unambiguous determination of whether a memory location contains an object reference or not, and in the case of a reference, the type and size of its referent must be known. This information is needed to support lazy storage allocation (See Section 3.4.3). DOSA needs this information to identify references in objects fetched from remote processors, and to allocate storage for their referents.

DOSA also assumes that the programs behave safely. In particular, an access cannot go beyond the end of an object. Otherwise DOSA’s access detection mechanism will not function correctly. Take Figure 3.4 for example, objects $O_1$ and $O_2$ are placed adjacent to each other. $O_1$ is read-write, and $O_2$ is read-only. For the program to run correctly, accesses to $O_2$ should be to its read-only mapping. However, if out-of-bound accesses may happen, then a write access to $O_2$ may be dereferenced through $O_1$'s handle. As a result, the write will be made to the read-write mapping of the object, therefore not caught by DOSA’s access detection mechanism.

3.7 Related Work

Orca [BBH+98], Jade [RL98], COOL [CGH94], and SAM [SL94] are parallel or distributed object-oriented languages. All of these systems differ from ours in that they present a new language or API to the programmer to express distributed sharing, while DOSA does not. DOSA aims to provide transparent object sharing for exist-

![Figure 3.4 Out-of-bound access.](image)
ing languages, such as Java. Furthermore, none of Orca, Jade, COOL, or SAM use VM-based mechanisms for object sharing.

Free et al used handles to reduce false sharing in a page-based DSM system [FA96]. If false sharing occurs within a page, their system moves the objects causing the false sharing to another page. However, their system still maintains coherency at the page granularity. The handles are only used to facilitate the data movement, not to detect memory accesses at the object granularity. Unlike DOSA, their system supports an untyped shared region. Therefore the address changes on a processor must be propagated to other processors.

Two other systems have used VM mechanisms for fine-grain DSM: Millipede [IS99] and the Region Trapping Library [BS99]. The fundamental difference between DOSA and these systems is that *DOSA takes advantage of the run-time type information and the safety features in modern programming languages and these other systems do not*. This allows DOSA to implement a number of optimizations that are not possible in these other systems.

Specifically, in Millipede a physical page may be mapped at multiple addresses in the virtual address space, as in DOSA, but the similarity ends there. In Millipede, each object resides in its own vpage, which is the size of a VM page. Different vpages are mapped to the same physical memory page, but the objects are offset within the vpage such that they do not overlap in the underlying physical page (see Figure 3.5). Different protection attributes may be set on different vpages, thereby achieving the same effect as DOSA, namely per-object access and write detection. The Millipede method requires one virtual memory mapping per object, while the DOSA method requires only three mappings per page, resulting in considerably less address space consumption and pressure on the TLB. Also, DOSA does not require any costly OS system calls (e.g., mprotect) to change page protections after initialization, while Millipede does.

The Region Trapping Library is similar to DOSA in that it allocates three different regions of memory with different protection attributes. Unlike DOSA, it does not use the regions in a way that is transparent to the programmer. Instead, it provides a special API. Furthermore, in the implementation, the read memory region and the read-write memory region are backed by different physical memory regions. This decision has the unfortunate side effect of forcing modifications made in the read-write region to be copied to the read region, every time protection changes from read-write to read. Their implementation also does not use handles. Instead, they
require the program to register the location of every shared reference to the run-time system. A reference must be registered as either \textit{constant} (referring to the same object throughout the program execution) or \textit{variable} (referring to different objects). This eliminates the extra indirection in our system. However, changing the object states can be very expensive: whenever the state of an object is changed, all variable references must be examined.

3.8 Summary

In this chapter, we have presented a new run-time system, DOSA, for supporting distributed sharing of objects in modern programming languages. The key insight is that \textit{the run-time type information and the safety features in modern programming languages allow efficient and transparent sharing of data with both fine-grained and coarse-grained access patterns}. Taking advantage of the language support, DOSA efficiently implements a shared space of objects on distributed systems. It uses lazy release consistency and a single-writer protocol. Like previous fine-grain systems, DOSA improves performance of fine-grained applications by eliminating false sharing. Furthermore, it incorporates a number of novel techniques that further improve the performance. These include lazy object allocation to improve data locality, read aggregation to reduce the number of message rounds for updating shared objects, write aggregation to reduce the number of write faults, and finally, a novel write notice reduction technique which dramatically reduces consistency data. We expect DOSA to support both coarse-grained and fine-grained sharing efficiently. The performance evaluation of DOSA will be presented in Chapter 5.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure3.5.png}
\caption{Multiview: one vpage for each object.}
\end{figure}
Chapter 4

Garbage collection on DOSA

In this chapter, we first describe the design of two intra-processor garbage collectors for DOSA. One of the collectors is based on mark-sweep, the other is based on generational copying. Next we present a new inter-processor garbage collection algorithm that achieves better overall program performance than state-of-the-art algorithms.

As we have seen in Chapter 2, the primary challenge to implementing an intra-processor garbage collector on conventional DSM systems is that the garbage collector may reduce the performance of the subsequent program execution. In contrast, the shared object space abstraction in DOSA decouples an object's naming from its address in memory, making the intra-processor garbage collector orthogonal to the DSM operations. Therefore, a processor is free to use any garbage collection algorithm without it having any negative effect on the performance of other processors. The shared object space abstraction not only eliminates the negative effect of the intra-processor garbage collector and improves the overall program performance, it also simplifies the design of the intra-processor garbage collector.

State-of-the-art inter-processor garbage collection algorithms have focused on reducing the amount of data exchanged between the processors. To reduce the amount of communication, they limit the scope of the data exchanged, and propagate the data asynchronously. As a result, the export lists on the processors are updated slowly, causing the retention of a large amount of garbage. This prevents the timely and accurate identification of garbage, and has a negative impact on the overall program performance. We have designed a new adaptive inter-processor garbage collector. We found that although the size of the complete object liveness information is very large in the worst case, it is actually small for many commonly used parallel data structures. Therefore, our inter-processor garbage collector dynamically adapts between two algorithms. In its normal mode of operation, it uses the complete liveness information gathered from the entire system to identify garbage more accurately and quickly than state-of-the-art algorithms, improving the overall program performance.
It also maintains good worst-case performance by falling back to a state-of-the-art inter-processor collection algorithm in the worst case.

The remainder of this chapter is organized as follows. Section 2.3 describes the intra-processor garbage collection algorithms for DOSA. Section 4.2 presents our inter-processor garbage collection algorithm. The algorithms presented in these two sections will be evaluated in Chapter 6. We examine related work in Section 4.3 and summarize this chapter in Section 4.4.

4.1 Intra-Processor Garbage Collection Algorithms

We have implemented two intra-processor garbage collection algorithms on DOSA that are representative of those in common use. The first algorithm is based on mark-sweep and the second algorithm is based on generational copying. These algorithms can be coupled with any inter-processor collection algorithm. In the following description, we use the weighted reference counting algorithm as an example to explain the interface between the intra-processor and inter-processor algorithms. Other algorithms will require only minor changes. The difference between an algorithm described in this section and its counterpart in Section 2.3 demonstrates how the shared object abstraction in DOSA simplifies the garbage collector implementation and eliminates the overheads in the copying collector. These algorithms will also be used in the evaluation in Chapter 6.

On DOSA, both garbage collectors are responsible for reclaiming unused handles in addition to the underlying storage. Handles are managed in the same way by both garbage collectors: There is a list of free handles and a list of allocated handles. During garbage collection, live handles are moved from the old allocated list to a new allocated list. In the mark-and-sweep collector, this occurs when an object is marked; and in the copying collector, this occurs when an object is copied from the old space to the new space. At the end of a garbage collection, any handles remaining in the old allocated list are unused, so the old allocated list is appended to the free list.

4.1.1 The Mark-sweep Garbage Collector

Apart from the handle management required in DOSA, the mark-and-sweep collector for DOSA is identical to its TreadMarks counterpart described in Section 2.3. The set of live objects is determined during the mark phase and the storage associated with dead objects is reclaimed during the sweep phase. The mark phase performs a depth-
first traversal of the directed graph formed by the objects (vertices) and references (edges), marking each encountered object. The traversal starts at the set of "root" references and the exported references. During this traversal, memory coherence is disabled, eliminating a vast amount of potential communication. Roughly speaking, the only ill consequence of using an out of date copy of an object is that it may delay the reclamation of storage. (See Ferreira and Shapiro [FS94b] for a detailed explanation.) The sweep phase has two parts. First, each locally created object is examined, and the storage used by the unmarked ones is returned to the free pool. Second, each imported reference is examined. If the object that it references was not marked, the object's creator is notified that the imported reference is no longer used.

4.1.2 The Copying Garbage Collector

The copying collector for DOSA is much simpler than its TreadMarks counterpart in Section 2.3. Each processor can start an intra-processor garbage collection asynchronously because the last writer of an object is always known. A processor starts a garbage collection if its current memory pool is full or if the number of import references received since the end of the last intra-processor garbage collection has exceeded a threshold (currently set at 8192 entries). Like the copying collector for TreadMarks, the collector starts a depth-first traversal from the "root" references and the exported references, and copies and scans all reachable objects if it is the last writer of the object. Forwarding pointers are unnecessary because the only reference that needs update is in the object's handle. After the traversal, imported references still pointing to the old address ranges (the fromspace) are removed from the import table and sent back to the object's creator. The fromspace can be immediately reclaimed and reused.

4.2 Inter-processor Garbage Collection

We have designed an inter-processor garbage collector that dynamically adapts between a modified version of Liskov and Ladin's centralized tracing algorithm [LL86] and the stub-scion algorithm. Our collector uses the centralized tracing algorithm to identify garbage more quickly and accurately in the common case by exchanging more GC data between the processors, and falls back to the stub-scion algorithm when the amount of extra GC data becomes too large. In this section, we first describe the graph-tracing algorithm, then explain why we need an adaptive algorithm.
4.2.1 The Graph-Tracing Algorithm

In our algorithm, a processor designated as the leader gathers complete object connectivity information from the entire system, builds an internal representation of the object graph, and identifies the garbage and notifies the other processors. This algorithm is able to reclaim both acyclic and cyclic garbage. Both the gathering of the liveness information and the notification are done asynchronously to avoid extra messages and synchronization.

Like the algorithm described in the previous section, our algorithm also maintains an export table and an import table on each processor. Outgoing messages are scanned and references to locally created objects are put in the export table. Incoming messages are also scanned and any references to remotely created objects are put in the import table. For every export or import reference, our algorithm also remembers the processors it has been sent to or received from. This extra bookkeeping allows each processor to build the stub and scion tables independently when it has to fall back to the stub-scion algorithm. In an intra-processor collection, the garbage collector starts tracing from the local roots and the set of exported references. The intra-processor garbage collector only reclaims local objects that are not reached in the traversal. All objects that are shared by more than one processor are reclaimed by the inter-processor collections.

Each processor asynchronously performs intra-processor garbage collections. In addition to reclaiming garbage, a processor also compiles the following information during an intra-processor collection:

1. The reachable set. This is the set of import references that are reachable from the local roots.

2. The transit set. This is the set of references in transit, that is, the references that have appeared in some outgoing messages between the end of the last intra-processor collection and the start of the current intra-processor collection.

3. The connectivity information between the locally unreachable export references and the locally unreachable import references. This information is expressed in the form of an adjacency list. To avoid duplicated entries from many processors due to replication, a processor only includes an object and its outgoing edges if it is the last writer of the object.
After an intra-processor collection, the three items listed above, together with the current vector timestamp of the processor, are asynchronously transmitted to the processor designated as the leader.

When the leader processor receives the three items above from another processor, it handles them differently. For the reachable set and the connectivity information, it discards earlier versions from the same processor and only keeps the most recent version of the data. A transit set, however, is discarded only when its vector timestamp is lower than that of the reachable sets from all other processors. When a transit set is discarded, we are already sure that all references contained in it have reached their destinations.

The first global collection is performed by the leader after it has received at least one set of information from every other processor. Afterwards, it can start a new global collection whenever it receives some new information from any one of the processors. In a global collection, the leader first builds an internal representation of the object graph using the connectivity information from every processor. The join of all reachable sets and the transit sets forms the global roots of the object graph, and the connectivity information shows the relationship between the objects. The transit sets must be traced because they contain potentially live references that may not appear in any processor's reachable set. The global collector starts tracing from the global roots using any standard tracing algorithm. After the tracing completes, dead objects, including cycles, are identified. The list of dead objects is asynchronously transmitted to all other processors so that they can be reclaimed.

The problem with the graph-tracing algorithm is that it cannot guarantee progress on conventional DSM systems when coupled with an intra-processor copying collector. This is because conventional DSM systems cannot distinguish the bogus writes by the copying collector from the writes by the program, and must propagate those writes as well. As a result, references to dead objects may appear in the transit set, preventing their reclamation. Take TreadMarks for example, assume there are two objects, $O_1$ and $O_2$. Object $O_1$ is live, while $O_2$ is in a cross-processor cycle of dead objects, i.e., it is only reachable from an exported reference. The intra-processor collector cannot reclaim either of the objects, and may copy both objects to the same page. Later, when a remote processor fetches the live object, the content of both objects will be sent to that processor, and the references in the two objects will be added to the transit set of the local processor. Therefore, the objects referenced by $O_2$ will be treated as live, and all dead objects in the cycle that contains $O_2$ will not
be reclaimed. DOSA avoids this problem because it does not propagate the bogus writes. Even if $O_1$ and $O_2$ are moved to the same page, the remote processor will only fetch $O_1$ because its own copy of $O_2$ is still valid.

Our garbage collector is similar to Liskov and Ladin’s algorithm [LL86], with three differences. First, Liskov and Ladin’s algorithm only requires each processor to compile partial connectivity information. For example, assume a processor has two exported objects, $O_1$ and $O_2$, and an imported reference $I_1$. Also assume that there are two paths, one from $O_1$ to $I_1$, the other from $O_2$ to $I_1$. Liskov and Ladin’s algorithm may record only one of the two paths. In contrast, our algorithm records both paths in the adjacency list. It has been pointed out by Rudalics [Rud90] that by omitting some of the paths between the export and import references, Liskov and Ladin’s algorithm may erroneously reclaim some live objects. Second, our algorithm does not require an upper bound of the message delivery time. Liskov and Ladin use the bound of the message delivery time to know when a transit reference set can be discarded. In our algorithm, the vector timestamp serves the same purpose. Finally, our algorithm performs a little extra bookkeeping so that it can switch to the stub-scion algorithm at any time.

**Thrashing**

On DOSA, the graph-tracing algorithm solves a new problem facing the state-of-the-art inter-processor collection algorithms – *thrashing*.

State-of-the-art inter-processor collection algorithms, such as the stub-scion algorithm and the weighted reference algorithm, cannot identify garbage accurately. As a result, they may cause thrashing on DOSA. Specifically, an imported object is removed from a processor’s local memory if it is unreachable either from the local roots or the scions. However, the object may still be live because it may be reachable from some other processor’s local roots. If the processor later accesses this object again, the object will have to be brought back. On some systems, e.g., DOSA, thrashing is very harmful because the objects that are thrashing do not enjoy the benefit of aggregation. For example, assume objects $A$, $B$, and $C$ are thrashing, and $A$ contains references to $B$ and $C$. When a processor follows a reference to $A$ and accesses it, memory is allocated for $A$, say in page $p$. When the processor follows the reference in $A$ to access $B$, $B$ is very likely to be allocated in page $p$ too. The first access to $A$ will bring every object already in $p$, including $A$, up to date. When $B$ is accessed,
however, a message will be sent to fetch $B$ alone. The same occurs when the processor later accesses $C$. This example shows that if there are many thrashing objects, it often takes a separate message to fetch a single object. This increases the number of messages. The graph-tracing algorithm identifies garbage accurately. Therefore, it eliminates the problem of thrashing.

The problem of thrashing also exists in conventional DSM systems. However, its effect on the program performance is smaller than in DOSA. In coarse-grain systems such as TreadMarks, an object that is unreachable from the local roots of a remote processor will still be sent to that processor as long as it resides in the same page as an object requested by that processor. In other words, TreadMarks avoids the thrashing problem by sending objects that may not be used by the remote processors. In fine-grain systems such as Shasta, thrashing is also not a problem because these systems do not aggregate the communication of objects. In short, conventional DSM systems do not have the problem of thrashing because they do not optimize as aggressively as does DOSA.

The Adaptive Algorithm

The drawback of the graph-tracing algorithm is that the amount of garbage collector data exchanged between the processors becomes prohibitively large in the worst case.

The garbage collector data in our algorithm consists of three parts: the reachable set, the transit set, and the adjacency list. The sizes of the reachable set and the transit set are comparable to the existing algorithms. In fact, the total size of the two sets should be no larger than the set of live references broadcast by each processor in the stub-scion algorithm.

The adjacency list, however, can be very large. It is proportional to the number of edges in the object graph, which is $O(n^2)$ in the worst case, with $n$ as the number of objects. We argue that in the common case the size of the adjacency list is much smaller. With many commonly used data structures, such as lists and trees, the size of the adjacency list is $O(n)$. In more complex data structures, such as dags and graphs, the size of the adjacency list may still be $O(n)$. For example, if the locally unreachable objects form a complete graph, we only have to record the tree edges in the depth-first-tree of the graph, which is $O(n)$. In such cases, we expect the size of the adjacency list to be reasonably small.
However, there are cases in which the size of the adjacency list is definitely $O(n^2)$. The size of the adjacency list of a directed bipartite graph, for example, the one in Figure 4.1 with $n/2$ export references and $n/2$ import references, is $O(n^2)$. In such cases, the adjacency list can be too large. Our solution is to fall back to the stub-scion algorithm when the adjacency list becomes too large. Our algorithm sets a threshold on the size of the adjacency lists. If a processor finds out that the threshold has been reached, it signals the other processors. Then each processor builds the stub and scion tables based on its local information and falls back to the stub-scion algorithm.

Our algorithm falls back to the stub-scion algorithm instead of the weighted reference counting algorithm because switching to the former does not require communication between the processors. In contrast, to switch to the weighted reference counting algorithm, the creator of each shared reference must find out how many remote processor have seen the reference in question. This requires communication between the processors. Besides, our evaluation shows that switching to the weighted reference counting algorithm will not offer much additional benefit to program performance (See Section 6.2).

To determine when to switch from the graph-tracing algorithm to the stub-scion algorithm, each processor remembers the size of the adjacency lists it has sent out in past intra-processor collections. Whenever a new intra-processor collection finishes, the processor adds the size of the newly created adjacency list to the total, then compares it with the size of the program data that has been sent out. It signals a switch if the total size of the adjacency lists exceeds a threshold, currently set as 20% of the size of the program data that has been sent out.

After switching to the stub-scion algorithm, each processor still calculates how large the adjacency lists would be during the intra-processor collections, and adds
the calculated sizes to the total. If the accumulated sizes drop below the threshold mentioned above, the system can switch back to the graph-tracing algorithm.

Cyclic Garbage

After switching to the stub-scion algorithm, we periodically invoke the graph-tracing algorithm or the synchronous global collection algorithm presented in Section 2.1.2 to reclaim cyclic garbage. Here we briefly describe when we invoke a cyclic collection and how we choose the algorithm.

After switching to the stub-scion algorithm, each processor maintains a GC timestamp, which is similar to the vector timestamp used by DOSA (See Section 3.4). In a system with \( n \) processors, the GC timestamp consists of \( n \) entries. Whenever a processor performs an intra-processor collection, its corresponding entry in the GC timestamp is increased by one. Changes to the GC timestamp are propagated asynchronously. The distance between two GC timestamps is the sum of the differences between the corresponding entries in the two timestamps. Whenever a processor allocates a new object, it tags the object with the current GC timestamp. An exported reference is deemed old by a processor if it is not reachable from the local roots and the distance between its GC timestamp and the current GC timestamp is greater than \( (2 \ast n) \). A processor invokes a cyclic collection if the size of the old objects exceeds 50% of the heap.

At the beginning of a cyclic collection, the processors exchange the calculated sizes of the adjacency lists in their most recent intra-processor collections. If the sum of the sizes exceeds the amount of program data communicated since the start of the program execution, they invoke the synchronous global collection algorithm presented in Section 2.1.2. Otherwise they switch to the graph-tracing algorithm to reclaim the cyclic garbage.

4.2.2 The Cost of the Centralized Tracing

In the centralized graph-tracing algorithm, a single processor traverses the entire object graph. In contrast, in the existing algorithms such as the WRT or the stub-scion algorithm, each processor only traverses the partial graph cached locally. One question to ask is if the CPU cost in the centralized tracing will become too high. We argue that the centralized tracing will not become a serious performance bottleneck. We will back up our argument with the measurements in Section 6.2.
The cost of the centralized tracing is unlikely to be high for two reasons. First, the processor that performs the tracing has complete object liveness information. Therefore it only traces those objects that are live. In contrast, the existing algorithms such as the WRT and stub-scion algorithms reclaim shared garbage slowly. Therefore on each processor there are many objects that are actually dead but are still treated as live and must be traced. Second, the tracing cost is proportional to the number of edges in the object graph. Consequently, it is also proportional to the size of the adjacency lists exchanged between the processors. If the adjacency lists are small, there are not many edges in the object graph and the tracing cost will be very low. If the adjacency lists are large enough so that the tracing would have a serious impact on the program performance, our algorithm will have already fallen back to the stub-scion algorithm and avoided the tracing cost.

4.3 Related Work

Concurrent garbage collection for shared-memory multiprocessors [AEL88, BDS91] and distributed systems [AMR92, PS95] has been an active area of research. We are, however, aware of only four attempts to design garbage collectors for DSM systems.

The fundamental difference between garbage collection for shared-memory multiprocessors and DSM systems is the communication overhead. If we apply a GC algorithm designed for a shared-memory multiprocessor to a DSM system, the cost will be unacceptable due to the communication and synchronization overheads. These overheads are due to the fact that current multiprocessor GC algorithms implicitly assume the existence of strongly consistent objects. In fact, the communication and synchronization overheads arise because of the necessity of providing strongly consistent objects.

Garbage collection on DSM systems is somewhat different from the RPC-based distributed systems due to the existence of multiple copies of the same object on several processors and the consistency problem that comes with it.

In this section, we first discuss the garbage collection algorithms for the RPC-based distributed systems, than review the previous DSM garbage collectors.
4.3.1 Distributed Garbage Collection

Plainfosse and Shapiro [PS95] gave a comprehensive review of distributed garbage collection algorithms. Existing distributed garbage collection algorithms can be categorized as either acyclic or cyclic algorithms.

The acyclic algorithms use variations of the reference counting technique to keep track of the cross-processor references. The weighted reference counting algorithm and the stub-scion algorithm discussed earlier in this chapter are representative of these algorithms. The main drawback of such algorithms is that they cannot reclaim cyclic garbage.

The cyclic algorithms can reclaim cyclic garbage. They can be further divided into tracing algorithms and heuristics algorithms.

There are four tracing algorithms that we know of. The centralized tracing algorithm by Liskov and Ladin [LL86] has been discussed in Section 4.2. Here we describe the remaining three algorithms.

Hughes [Hug85] proposed a variation of a distributed mark-sweep algorithm, with mark bits replaced by timestamps. The key idea behind this algorithm is that a dead object's timestamp remains constant whereas a live object's timestamp increases monotonically. Each intra-processor collection repeatedly propagates the timestamps from the local roots and the export references to the import references. An import reference reachable from the local roots is marked with the time the intra-processor collection started; an imported reference only reachable from an export reference receives the timestamp of the export reference. A timestamp threshold is computed, and import references whose timestamps are less than the threshold can be safely reclaimed. The drawback of this algorithm is that it reclaims both acyclic and cyclic garbage slowly: all garbage cannot be reclaimed until a global collection completes.

Bennett has designed a garbage collector for Distributed Smalltalk [Ben87, Ben90], which includes a cyclic garbage collector that asynchronously performs global tracing in the system. His algorithm places no restrictions on the intra-processor collector, which may employ either reference counting or the tracing schemes. In the cyclic algorithm, each processor starts collections from the local roots. Whenever a remote reference is found, a mark message is sent to the owner of that object. When a processor receives a mark message, it marks the object named by the message and searches it for remote references. This process continues until all objects reachable from the root sets have been marked. Bennett's design also includes a reference counting based
acyclic algorithm, which can be run concurrently with the cyclic algorithm. Therefore, acyclic garbage can be reclaimed earlier than in Hughes' algorithm. The drawback of this algorithm is that cyclic garbage cannot be reclaimed quickly. The global tracing can take a long time to complete, since the garbage collection time is determined by the length of the longest cross-processor data chain. Lang et al [LQP92] have also designed a similar algorithm.

There are also some heuristic algorithms that attempt to reclaim garbage cycles without global tracing. Bishop [Bis77] proposed an object migration algorithm in which objects suspected to be in a garbage cycle are migrated to a single processor so that they can be reclaimed by an acyclic collector. In Vestal's trial deletion algorithm [Ves87], objects suspected to be in a garbage cycle are used as a seed for a cycle-detection protocol. This technique essentially consists of a trial deletion of the seed object in order to see if the descendent's counts drop to zero. The main difficulty of both algorithms is to come up with a good heuristics to select suspect object. If the heuristics is wrong, Bishop's algorithm will result in significant thrashing in the system, and Vestal's algorithm will send many extra messages.

All of the above algorithms require that each processor perform intra-processor garbage collections asynchronously to reduce the communication costs. Taura and Yonezawa [TY97] concluded that synchronous collections may be more efficient than asynchronous collections. However, their conclusion may not apply to other systems. In their implementation, the intra-processor collector on each processor does not respond to remote data requests during the garbage collections. This simplifies the synchronization between the mutator and the garbage collector, but penalizes the asynchronous collections because many processors may have to wait while one processor performs garbage collection. However, some intra-processor collectors, including ours, can serve the data requests while collecting. Therefore Taura and Yonezawa's conclusion may not apply in our case. This issue will be further explored in Chapter 6.

4.3.2 Garbage Collection on DSM Systems

Le Sergent [SB92, MS95] described an extension of an incremental copying collector originally designed for a multiprocessor to a DSM system. The garbage collector requires a consistent image of the entire object graph, therefore is very expensive. They have not reported any performance measurement for it.
Kordale's GC design [KAS93] for DSM is based on a mark-sweep technique. Like Le Sergent's collector, this algorithm also requires a consistent image of the entire object graph, and is therefore very expensive.

Ferreira and Shapiro [FS94a] designed a copying garbage collector for weakly consistent DSM systems. They were the first to point out that garbage collectors can be designed to tolerate memory inconsistency. Their algorithm allows the processors to collect independently. Their design depends on the entry consistency model, which presents a single-writer interface. Address changes are propagated asynchronously and piggybacked in messages sent out by the application program. The problem of delayed memory reclamation was not fully addressed in their paper. Also, their collector does not reclaim garbage cycles that span processors. They evaluated the scalability of their design, but did not study the impact of the garbage collector on overall program performance. It is not straightforward to adapt their algorithms to DSM systems using other relaxed consistency protocols.

In our previous work [YC96], we have designed a conservative mark-sweep garbage collector for DSM systems. We showed that the garbage collector can be very efficient, incurring low overheads during garbage collection. However, the poor spatial locality as a side effect of the mark-sweep collector could result in high communication cost, and was very detrimental to the overall performance of some programs.

4.4 Summary

In this chapter, we have first described the implementation of the intra-processor garbage collectors for DOSA. We have also presented the design of a new adaptive algorithm for inter-processor garbage collection.

The shared object space abstraction in DOSA decouples an object's naming from its address in memory, making the intra-processor garbage collector orthogonal to the DSM operations. Therefore, a processor is free to use any garbage collection algorithm without it having any negative effect on the performance of other processors. The share object space abstraction not only eliminates the negative effect of the intra-processor garbage collector and improves the overall program performance, it also simplifies the design of the intra-processor garbage collectors.

We have designed a new adaptive algorithm for inter-processor garbage collection. In its normal mode of execution, the processors exchange more information so that a complete object graph can be built on one processor, allowing more accurate garbage
identification and more timely reclamation. We argue that in the common case the benefits of our algorithm outweighs the increased amount of communication, allowing better overall program performance. In the worst case in which the amount of extra GC data is too large, our algorithm can seamlessly fall back to one of the state-of-the-art algorithms.
Chapter 5

Evaluation: Non-Garbage-Collected Programs

This chapter presents an evaluation of DOSA using non-garbage-collected programs. We compare its performance with TreadMarks, a DSM system that is efficient at handling coarse-grained sharing. Section 5.1 describes our evaluation methodology, and Section 5.2 describes the programs used in the evaluation. The overall performance of the programs are presented in Section 5.3, and the effects of the individual optimizations are presented in Section 5.4. Finally, we make a qualitative comparison of DOSA with other DSM systems in Section 5.5, and summarize the results in Section 5.6.

5.1 Methodology

Our performance evaluation seeks to substantiate the following claims:

1. The performance of coarse-grained applications in DOSA is nearly as good as in TreadMarks.

2. The performance of fine-grained applications in DOSA is considerably better than in TreadMarks.

For the same reason as we have explained in Section 2.4.1, we have chosen to carry out the following experiments using instrumented C programs. We have taken existing C applications, and we have re-written them to follow the model of a handle-based implementation. In other words, a handle table is introduced, and all pointers are indirected through the handle table. This approach represents the results that could be achieved by a language or compilation environment that is compatible with our approach for maintaining consistency, but otherwise exhibits no compilation or execution differences with the conventional TreadMarks execution environment. In other words, these experiments isolate the benefits and the drawbacks of our consistency maintenance methods from other aspects of the compilation and execution
process. It also allows us to assess the overhead of the extra indirection on single-processor execution times. The compiler optimizations discussed in Section 3.5 have been implemented by hand in both the TreadMarks and the DOSA programs. We report results with and without these optimizations present.

Our evaluation was conducted on a cluster of 32 Pentium-II processors connected with a 100Mbps Ethernet.

5.2 Applications

Our choice of applications follows immediately from the goals of our performance evaluation. First, we have chosen two coarse-grained applications to assess the potential performance loss in such applications, compared to a system that is geared towards such coarse-grained applications. These two applications are SOR and Water-N-Squared. SOR performs red-black successive over-relaxation on a 2-D grid, and Water-N-Squared is a molecular dynamics simulation from the SPLASH [SWG92] benchmark suite.

Second, we use two fine-grained applications for which we hope to see significant benefits over a page-based system. These applications are Barnes-Hut and Water-Spatial from the SPLASH-2 [WOT+95] benchmark suite. Barnes-Hut is an N-body simulation, and Water-Spatial is a molecular dynamics simulation optimized for spatial locality.

For each of these applications, Table 5.1 lists the problem size and the sequential execution times. The sequential execution times were obtained by removing all TreadMarks or DOSA calls from the applications and for DOSA using the compile-time optimizations described in Section 3.5. The optimizations were applied by hand. These timings show that the overhead of the extra level of dereferencing in the handle-based versions of the applications is never more than 5.2% on one processor for any of the four applications. The sequential execution times without handles were used as the basis for computing the speedups reported later in this chapter.

5.3 Overall Results

5.3.1 Fine-grained Applications

Figures 5.1 through 5.4 present the speedups for Barnes-Hut and Water-Spatial for a small and a large data set. Tables 5.2 through 5.4 detail the statistics from the
<table>
<thead>
<tr>
<th>Application</th>
<th>Problem Size</th>
<th>Time (sec.)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Orig.</td>
<td>Handle</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4094x2047, 20 steps</td>
<td>27.57</td>
<td>28.05</td>
<td></td>
</tr>
<tr>
<td>Water-N-Squared</td>
<td>1728 mols, 2 steps</td>
<td>71.59</td>
<td>73.83</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2744 mols, 2 steps</td>
<td>190.63</td>
<td>193.50</td>
<td></td>
</tr>
<tr>
<td>Barnes-Hut</td>
<td>32K bodies, 3 steps</td>
<td>58.68</td>
<td>60.84</td>
<td></td>
</tr>
<tr>
<td></td>
<td>131K bodies, 3 steps</td>
<td>270.34</td>
<td>284.43</td>
<td></td>
</tr>
<tr>
<td>Water-Spatial</td>
<td>4K mols, 9 steps</td>
<td>89.63</td>
<td>89.80</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32K mols, 2 steps</td>
<td>158.57</td>
<td>160.39</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1  Applications, input data sets, and sequential execution time.

Figure 5.1  The speedup of Barnes-Hut for the small data set.

Figure 5.2  The speedup of Barnes-Hut for the large data set.
execution of the two programs on a varying number of processors. In TreadMarks, a call to diff request which may involve parallel messages to different processors is counted as one message round. In DOSA, a call to object request which may involve parallel messages to different processors to update other objects in the same page is counted as one message round.

We derive the following conclusions from the results. First, the overhead of the extra indirection in the sequential code for these applications is less than 5.2% for Barnes-Hut and 1.1% for Water-Spatial. Second, even at a small number of processors and for a small data set, the benefits of the handle-based implementation are larger than the cost of the extra indirection. For example, for Barnes-Hut with 32K bodies, DOSA outperforms TreadMarks by 22% and 29% on 8 and 16 processors, respectively. For Water-Spatial with 4K molecules, DOSA outperforms TreadMarks by 25% and 62% on 8 and 16 processors, respectively. Third, for larger numbers of processors and larger data sets, the benefits of the handle-based implementation grow considerably larger. For example, for Barnes-Hut with 128K bodies, DOSA outperforms TreadMarks by 98% on 32 processors. For Water-Spatial with 32K molecules, DOSA outperforms TreadMarks by 51% on 32 processors. The reason that this gap between the two systems is smaller than with 4K molecules on 16 processors is from somewhat reduced false sharing in TreadMarks when scaling up the number of molecules in Water-Spatial.
<table>
<thead>
<tr>
<th>Application</th>
<th>Barnes-Hut</th>
<th>Water-Spatial</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Small</td>
<td>Large</td>
</tr>
<tr>
<td></td>
<td>Tmk</td>
<td>DOSA</td>
</tr>
<tr>
<td>Time</td>
<td>15.67</td>
<td>12.88</td>
</tr>
<tr>
<td>Data (MB)</td>
<td>82.35</td>
<td>32.90</td>
</tr>
<tr>
<td>Messages (1000)</td>
<td>248.5</td>
<td>75.57</td>
</tr>
<tr>
<td>Msg rounds (1000)</td>
<td>33.19</td>
<td>25.58</td>
</tr>
<tr>
<td>Memory (MB)</td>
<td>7.36</td>
<td>1.99</td>
</tr>
</tbody>
</table>

**Table 5.2** Detailed statistics for TreadMarks and DOSA on 8 processors for fine-grained applications, Barnes-Hut and Water-Spatial.

<table>
<thead>
<tr>
<th>Application</th>
<th>Barnes-Hut</th>
<th>Water-Spatial</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Small</td>
<td>Large</td>
</tr>
<tr>
<td></td>
<td>Tmk</td>
<td>DOSA</td>
</tr>
<tr>
<td>Time</td>
<td>14.17</td>
<td>11.02</td>
</tr>
<tr>
<td>Data (MB)</td>
<td>155.5</td>
<td>51.99</td>
</tr>
<tr>
<td>Messages (1000)</td>
<td>784.6</td>
<td>156.7</td>
</tr>
<tr>
<td>Msg rounds (1000)</td>
<td>58.2</td>
<td>51.31</td>
</tr>
<tr>
<td>Memory (MB)</td>
<td>7.36</td>
<td>1.42</td>
</tr>
</tbody>
</table>

**Table 5.3** Detailed statistics for TreadMarks and DOSA on 16 processors for fine-grained applications, Barnes-Hut and Water-Spatial.

<table>
<thead>
<tr>
<th>Application</th>
<th>Barnes-Hut</th>
<th>Water-Spatial</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Small</td>
<td>Large</td>
</tr>
<tr>
<td></td>
<td>Tmk</td>
<td>DOSA</td>
</tr>
<tr>
<td>Time</td>
<td>18.07</td>
<td>12.06</td>
</tr>
<tr>
<td>Data (MB)</td>
<td>315.3</td>
<td>82.6</td>
</tr>
<tr>
<td>Messages (1000)</td>
<td>2549</td>
<td>307.2</td>
</tr>
<tr>
<td>Msg rounds (1000)</td>
<td>108.2</td>
<td>98.90</td>
</tr>
<tr>
<td>Memory (MB)</td>
<td>7.36</td>
<td>1.05</td>
</tr>
</tbody>
</table>

**Table 5.4** Detailed statistics for TreadMarks and DOSA on 32 processors for fine-grained applications, Barnes-Hut and Water-Spatial.
The reasons behind these improvements can be seen in Table 5.2, 5.3, and 5.4. These tables show for both implementations, the number of messages exchanged, the number of message "rounds", the amount of data, and the average amount of shared data allocated on one processor. The benefits of lazy object allocation for these applications are quite clear: the memory footprint of DOSA is considerably smaller than that of TreadMarks, and, for a given data set, gets smaller as the number of processors is increased.

The tables above also show a substantial reduction in the amount of data sent for DOSA, as a result of the reduction in false sharing. There is also a substantial reduction in the number of messages, and, more importantly, the number of message rounds in the two programs. On 32 processors, DOSA reduces the amount of data communicated by up to 80% in Barnes-Hut, and by up to 50% in Water-Spatial. It also reduces the number of message rounds by up to 22% in Barnes-Hut, and by up to 79% in Water-Spatial.

5.3.2 Coarse-grained Applications

Figures 5.5 through 5.8 present the speedups of SOR and Water-N-Squared for a small and a large data set. Tables 5.5 through 5.7 detail the statistics from the execution of the two programs on a varying number of processors.

For these two coarse-grained applications, DOSA performs comparably to TreadMarks. In fact, DOSA performs almost identically to TreadMarks for SOR; both send the same amount of data and number of messages, and allocate the same amount of physical memory on each processor.

For Water-N-Squared, DOSA slightly underperforms TreadMarks on 8 and 16 processors, but slightly outperforms TreadMarks on 32 processors. This is because Water-N-Squared has migratory data, and diff accumulation occurs in TreadMarks. TreadMarks uses a multiple-writer protocol, and the modifications made to a page by a processor are represented by a diff. A diff is a runlength encoding of the modifications made to a page, generated by comparing the page to a copy saved prior to the modifications. A processor accessing migratory data may have to fetch diffs from all other processors that have previously modified the page that contains the data [LDCZ95]. The sum of the diffs may exceed the size of the data itself. In contrast, DOSA uses a single-writer protocol, and always fetch the whole object. On the other hand, in TreadMarks diffs are only one-fifth to one-fourth the size of a page,
Figure 5.5  The speedup of SOR for the small data set.

Figure 5.6  The speedup of SOR for the large data set.

Figure 5.7  The speedup of Water-N-Squared for the small data set.

Figure 5.8  The speedup of Water-N-Squared for the large data set.
thus it takes a long migration path for diff accumulation to lose to sending whole objects. On 16 processors, diff accumulation does not outweigh sending whole objects, and DOSA is up to 7% slower than TreadMarks from sending 14% more data. On 32 processors, DOSA sends 14% less data than TreadMarks and is up to 6% faster than TreadMarks.

### 5.4 Effects of the Various Optimizations

To achieve the results described in the previous section, various optimizations were used in DOSA. These optimizations include lazy object allocation (Section 3.4.3), read aggregation (Section 3.4.6), write aggregation (Section 3.4.5), and compile-time optimization (Section 3.5). In the analysis of the effect of read aggregation, we will also justify our use of the “message rounds” as a metric for the communication cost.

To see what effect each optimization has individually, we performed the following experiments: For each of the optimizations, we compare the performance of DOSA without that optimization to the fully-optimized system. Figure 5.9 shows the speedups for each of the experiments, except for compile-time optimization, for Barnes-Hut, Water-Spatial, and Water-N-Squared. The compile-time optimization is omitted because it only effects SOR. SOR is omitted because the only optimization that has any effect is the compile-time optimization. Table 5.8 provides further detail beyond speedups on the effects of the optimizations on 32 processors.

<table>
<thead>
<tr>
<th>Application</th>
<th>SOR</th>
<th>Water-Nsquare</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Small Tmk</td>
<td>Small Tmk</td>
</tr>
<tr>
<td>Time</td>
<td>2.79</td>
<td>2.88</td>
</tr>
<tr>
<td>Data (MB)</td>
<td>4.89</td>
<td>4.73</td>
</tr>
<tr>
<td>Messages (1000)</td>
<td>2.81</td>
<td>2.82</td>
</tr>
<tr>
<td>Msg rounds (1000)</td>
<td>1.12</td>
<td>1.12</td>
</tr>
<tr>
<td>Memory (MB)</td>
<td>6.35</td>
<td>6.35</td>
</tr>
</tbody>
</table>

**Table 5.5** Detailed statistics for TreadMarks and DOSA on 8 processors for coarse-grained applications SOR and Water-N-Squared.
Table 5.6  Detailed statistics for TreadMarks and DOSA on 16 processors for coarse-grained applications SOR and Water-N-Squared.

<table>
<thead>
<tr>
<th>Application</th>
<th></th>
<th>SOR</th>
<th></th>
<th>Water-Nsquare</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tmk</td>
<td>DOSA</td>
<td>Tmk</td>
<td>DOSA</td>
</tr>
<tr>
<td>Time</td>
<td>1.59</td>
<td>1.56</td>
<td>1.98</td>
<td>2.00</td>
</tr>
<tr>
<td>Data (MB)</td>
<td>10.4</td>
<td>10.3</td>
<td>10.5</td>
<td>10.3</td>
</tr>
<tr>
<td>Messages (1000)</td>
<td>6.08</td>
<td>6.02</td>
<td>6.02</td>
<td>6.02</td>
</tr>
<tr>
<td>Msg rounds (1000)</td>
<td>2.46</td>
<td>2.46</td>
<td>2.40</td>
<td>2.40</td>
</tr>
<tr>
<td>Memory (MB)</td>
<td>3.20</td>
<td>3.20</td>
<td>4.26</td>
<td>4.26</td>
</tr>
</tbody>
</table>

Table 5.7  Detailed statistics for TreadMarks and DOSA on 32 processors for coarse-grained applications SOR and Water-N-Squared.

<table>
<thead>
<tr>
<th>Application</th>
<th></th>
<th>SOR</th>
<th></th>
<th>Water-Nsquare</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tmk</td>
<td>DOSA</td>
<td>Tmk</td>
<td>DOSA</td>
</tr>
<tr>
<td>Time</td>
<td>1.10</td>
<td>1.10</td>
<td>1.32</td>
<td>1.31</td>
</tr>
<tr>
<td>Data (MB)</td>
<td>23.6</td>
<td>23.6</td>
<td>23.6</td>
<td>23.6</td>
</tr>
<tr>
<td>Messages (1000)</td>
<td>12.56</td>
<td>12.56</td>
<td>12.44</td>
<td>12.44</td>
</tr>
<tr>
<td>Msg rounds (1000)</td>
<td>4.96</td>
<td>4.96</td>
<td>4.96</td>
<td>4.96</td>
</tr>
<tr>
<td>Memory (MB)</td>
<td>1.64</td>
<td>1.64</td>
<td>2.18</td>
<td>2.18</td>
</tr>
</tbody>
</table>

Figure 5.9  Speedup comparison between DOSA and DOSA without each of the optimizations on 32 processors.
<table>
<thead>
<tr>
<th>Application</th>
<th>DOSA</th>
<th>w/o Read Aggre.</th>
<th>w/o Lazy Alloc.</th>
<th>w/o Write Aggre.</th>
<th>w/o W.N. Reduc.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Barnes-Hut(lg)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time (sec.)</td>
<td>45.07</td>
<td>139.88</td>
<td>50.90</td>
<td>46.05</td>
<td>49.56</td>
</tr>
<tr>
<td>Data (MB)</td>
<td>245.8</td>
<td>124.7</td>
<td>251.7</td>
<td>246.2</td>
<td>277.4</td>
</tr>
<tr>
<td>Write notices (M)</td>
<td>6.42</td>
<td>6.42</td>
<td>6.42</td>
<td>6.42</td>
<td>35.4</td>
</tr>
<tr>
<td>Messages (thousands)</td>
<td>1027.9</td>
<td>2254.4</td>
<td>1612.3</td>
<td>1027.9</td>
<td>1028.0</td>
</tr>
<tr>
<td>Msg rounds (thousands)</td>
<td>341.3</td>
<td>1123.7</td>
<td>428.5</td>
<td>341.3</td>
<td>341.3</td>
</tr>
<tr>
<td>Mem. allocated (MB)</td>
<td>3.35</td>
<td>3.35</td>
<td>23.2</td>
<td>3.35</td>
<td>3.35</td>
</tr>
<tr>
<td>Write faults</td>
<td>27586</td>
<td>28248</td>
<td>163865</td>
<td>582533</td>
<td>27728</td>
</tr>
<tr>
<td><strong>Water-Spatial (lg)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time (sec.)</td>
<td>8.52</td>
<td>10.54</td>
<td>10.07</td>
<td>8.54</td>
<td>8.75</td>
</tr>
<tr>
<td>Data (MB)</td>
<td>166.8</td>
<td>166.0</td>
<td>167.2</td>
<td>166.7</td>
<td>169.4</td>
</tr>
<tr>
<td>Write notices (M)</td>
<td>0.74</td>
<td>0.74</td>
<td>0.74</td>
<td>0.74</td>
<td>3.30</td>
</tr>
<tr>
<td>Messages (thousands)</td>
<td>109.6</td>
<td>478.7</td>
<td>184.0</td>
<td>109.6</td>
<td>109.6</td>
</tr>
<tr>
<td>Msg rounds (thousands)</td>
<td>41.49</td>
<td>238.3</td>
<td>72.66</td>
<td>41.49</td>
<td>41.49</td>
</tr>
<tr>
<td>Mem. allocated (MB)</td>
<td>2.64</td>
<td>2.64</td>
<td>22.5</td>
<td>2.64</td>
<td>2.64</td>
</tr>
<tr>
<td>Write faults</td>
<td>20989</td>
<td>20777</td>
<td>35277</td>
<td>110864</td>
<td>21062</td>
</tr>
<tr>
<td><strong>Water-N-Squared (lg)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data (MB)</td>
<td>181.3</td>
<td>183.2</td>
<td>181.8</td>
<td>181.4</td>
<td>181.6</td>
</tr>
<tr>
<td>Write notices (M)</td>
<td>0.81</td>
<td>0.81</td>
<td>0.81</td>
<td>0.81</td>
<td>0.97</td>
</tr>
<tr>
<td>Messages (thousands)</td>
<td>101.1</td>
<td>530.8</td>
<td>101.2</td>
<td>101.1</td>
<td>101.1</td>
</tr>
<tr>
<td>Msg rounds (thousands)</td>
<td>44.76</td>
<td>261.7</td>
<td>44.87</td>
<td>44.77</td>
<td>44.76</td>
</tr>
<tr>
<td>Mem. allocated (MB)</td>
<td>1.04</td>
<td>1.04</td>
<td>1.89</td>
<td>1.04</td>
<td>1.04</td>
</tr>
<tr>
<td>Write faults</td>
<td>16953</td>
<td>87498</td>
<td>16978</td>
<td>96589</td>
<td>16968</td>
</tr>
</tbody>
</table>

Table 5.8 Statistics for DOSA and DOSA without each of the optimizations on 32 processors for Barnes-Hut, Water-Spatial, and Water-N-Squared.
5.4.1 Lazy Object Allocation

Table 5.8 shows that without lazy object allocation, DOSA sends 57\% and 68\% more messages and runs 13\% and 18\% slower than DOSA with lazy object allocation, for Barnes-Hut and Water-Spatial, respectively.

Lazy object allocation has no impact on Water-N-Squared because molecules are allocated in a 1-D array, and each processor always accesses the same segment consisting of half of the array elements in a fixed increasing order.

Lazy object allocation significantly benefits irregular applications that exhibit spatial locality of reference in their physical domain. For example, in N-body simulation programs such as Barnes-Hut and Water-Spatial, the physical domain is the 3-dimensional space that contains the particles. Even though the bodies in Barnes-Hut and the molecules in Water-Spatial are input or generated in random order, in the parallel algorithms, each processor only updates bodies or molecules corresponding to a contiguous physical subdomain. Furthermore, data references only happen on the boundary of each subdomain. For such applications, lazy object allocation will only allocate space for objects on a processor that are accessed by that processor. Therefore, a physical page will contain only “useful” objects. With read aggregation, these objects will all be updated in a single round of parallel messages when faulting on the first object. In contrast, without lazy object allocation, objects are allocated on all processors in the same order and at the same virtual address. Thus, the order of the objects in memory reflects the access pattern of the initialization which may differ from the computation. In other words, objects accessed by a specific processor may be scattered in many more pages than in the scenario with lazy object allocation. When accessing these objects, this processor has to fault many more times and send many more rounds of messages in order to update them.

5.4.2 Read Aggregation

The single optimization that most affects performance is read aggregation. Table 5.8 shows that without read aggregation, DOSA sends 2.2, 4.4, and 5.2 times more messages, and 3.3, 5.8, and 5.8 times more data message rounds for Barnes-Hut, Water-Spatial, and Water-N-Squared, respectively. As a consequence, DOSA without read aggregation is 310\%, 24\%, and 22\% slower than DOSA for these three applications.

Intuitively, the potential problem with read aggregation is that DOSA may fetch more objects than necessary. DOSA without read aggregation, however, only fetches
accessed or necessary objects. Thus, by looking at the difference between DOSA with and without read aggregation, we can determine the amount of unnecessary data communicated. Table 5.8 shows that DOSA without read aggregation sends almost the same amount of data as fully-optimized DOSA for Water-Spatial and Water-N-Squared, but half as much data for Barnes-Hut. The data totals for Water-Spatial and Water-N-Squared are nearly identical because lazy object allocation improves the initial spatial locality of the data on each processor. Since the set of molecules accessed by each processor remains static, spatial locality is good throughout the execution. Consequently, objects prefetched by read aggregation are typically used. In Barnes-Hut, however, the set of bodies accessed by a processor changes over time. In effect, when a body migrates from its old processor to its new one, it leaves behind a “hole” in the page that it used to occupy. When the old processor accesses any of the remaining objects in that page, read aggregation will still update the hole.

Message Rounds

Throughout this chapter, we have used the “message rounds” (See Section 3.3.3) as a metric for the communication cost. We argued that the number of message rounds is more meaningful than the message count because the cost of the parallel messages sent out in a message round is inexpensive.

The effect of the read aggregation reported in Table 5.8 does not directly support our argument above. Read aggregation not only overlaps the messages sent to different processors, it also reduces the number of messages by fetching multiple objects from

<table>
<thead>
<tr>
<th>Application</th>
<th>Full Read Aggr</th>
<th>Partial Read Aggr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes-Hut</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time (sec)</td>
<td>45.07</td>
<td>84.87</td>
</tr>
<tr>
<td>Data (MB)</td>
<td>245.8</td>
<td>242.8</td>
</tr>
<tr>
<td>Messages (thousand)</td>
<td>1027.9</td>
<td>1009.2</td>
</tr>
<tr>
<td>Msg rounds (thousands)</td>
<td>341.3</td>
<td>501.9</td>
</tr>
<tr>
<td>Water-Spatial</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time (sec)</td>
<td>8.52</td>
<td>9.10</td>
</tr>
<tr>
<td>Data (MB)</td>
<td>166.8</td>
<td>166.8</td>
</tr>
<tr>
<td>Messages (thousand)</td>
<td>109.6</td>
<td>109.4</td>
</tr>
<tr>
<td>Msg rounds (thousands)</td>
<td>41.49</td>
<td>53.69</td>
</tr>
</tbody>
</table>

Table 5.9 Statistics for DOSA with full and partial read aggregation on 32 processors for Barnes-Hut and Water-Spatial with the large data set.
the same processor at once. To isolate the effect of the message overlapping, we have also performed partial read aggregation. That is, in a page fault, all invalid objects that are in the same page as the faulted object and have the same last writer as the faulted object are brought up-to-date. Other invalid objects are not updated. A program running on DOSA with partial read aggregation will send the same number of messages as on the fully optimized DOSA, but may have fewer message rounds. Table 5.9 reports the performance of Barnes-Hut and Water-Spatial on 32 processors with partial and full read aggregation. Water-N-Squared is excluded because it mainly consists of migratory data, and almost all objects in the same page has the same last writer. SOR is excluded because each object (row) in SOR is larger than a page, and has a single last writer. In both cases, there is no opportunity for message overlapping.

In Barnes-Hut, DOSA with full read aggregation sends 2% more messages than with partial read aggregation. However, the message overlapping in full read aggregation reduces the number of message rounds by 32%, and improves the program performance by 88%. In Water-Spatial, the program sends about the same number of messages on both versions of DOSA. However, the message overlapping reduces the number of message rounds by 5%, and improves the program performance by 7%. This demonstrates the benefit of the message overlapping, and justifies our use of the message rounds as a metric for the communication cost.

### 5.4.3 Write Aggregation

Table 5.8 shows that write aggregation reduces the number of page faults by factors of 21, 5.3, and 5.7 for Barnes-Hut, Water-Spatial, and Water-N-Squared, respectively. As a result, DOSA is one second or 2.2% faster for Barnes-Hut than DOSA without write aggregation. The impact on Water-Spatial and Water-N-Square is less than the measurement error.

### 5.4.4 Write Notice Reduction

Table 5.8 shows that our write notice reduction optimization is highly effective for Barnes-Hut and Water-Spatial. For Barnes-Hut, it reduces the amount of write notice data by a factor of 5.5, resulting in a 10% performance improvement; and for Water-Spatial, it reduces the amount of write notice data by a factor of 4.5, resulting in a 3% performance improvement. This optimization has little effect on the other applications.
5.4.5 Compile-time Optimization

Table 5.10 shows that for SOR, the compile-time optimization can significantly improve the performance. On a single processor, the compile-time optimization improves the performance of the original array-based version of SOR/lg by 20%, and the handle-based version by 69%. On 32 nodes, the improvements are 17% and 40% for the two versions, respectively.

5.5 Discussion

We have already compared our work extensively to TreadMarks [ACD+96] and Shasta [SGT96], using them as examples of coarse-grain and fine-grain DSM systems. Similar comparisons can be made with other DSM systems [LH89, CBZ95, ZIL96].

Dwarkadas et al. [DGK+99] compare Cashmere, a coarse-grain system, and Shasta running on an identical platform – a cluster of four four-way AlphaServers connected by a Memory Channel network. Both systems are designed to leverage the Memory Channel network and take advantage of the hardware shared memory within each SMP node. In general, Cashmere outperforms Shasta on coarse-grained applications and Shasta outperforms Cashmere on fine-grained applications.

All of the applications used in this chapter were also used in their paper. Shasta only outperformed Cashmere for one of these four applications, the fine-grained Barnes-Hut/lg by 350%. Cashmere outperformed Shasta by 50% and 10% for the coarse-grained applications, SOR/sm and Water-N-Squared/4K, respectively. The only surprise is that Cashmere equals Shasta on the fine-grained application Water-Spatial. (The difference is less than 3% in favor of Cashmere.) They attribute this result to the run-time overhead of the cache line validity checks in Shasta. In contrast, DOSA outperforms TreadMarks by 62% on Water-Spatial. We attribute this result to lazy object allocation, which is not possible in Shasta, and read aggregation.

<table>
<thead>
<tr>
<th>Application</th>
<th>Tmk/opt.</th>
<th>Tmk/noopt.</th>
<th>DOSA/opt.</th>
<th>DOSA/noopt.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1-proc</td>
<td>32-proc</td>
<td>1-proc</td>
<td>32-proc</td>
</tr>
<tr>
<td>SOR (lg)</td>
<td>27.57</td>
<td>1.32</td>
<td>33.19</td>
<td>1.54</td>
</tr>
</tbody>
</table>

Table 5.10 Running time (sec.) comparison between SOR with and without the compile-time optimization.
Previous fine-grain DSM systems, such as Shasta and Orca, augment the code with instructions to detect reads and writes. Our results show that the use of handles to detect accesses is more efficient than instrumentation. In our evaluation, the overhead of the extra level of dereferencing in the handle-based versions of the applications is never more than 5.2% on one processor for any of the four applications. Three of the applications, Barnes-Hut, Water-Spatial, and Water-N-Squared, were also used in an evaluation of Shasta [SG97]. The access detection overheads in Shasta for the three applications are 9.6%, 26.5%, and 23.6%, respectively.

5.6 Summary

In this chapter, we have evaluated the performance of DOSA. Our evaluation was conducted on a cluster of 32 Pentium-II processors connected with a 100Mbps Ethernet. We compared the performance of DOSA with that of TreadMarks, a state-of-the-art coarse-grain DSM system.

Our performance evaluation substantiates the following claims:

1. The performance of coarse-grained applications is nearly as good as in TreadMarks (within 6%). Since the performance of such applications is already good in TreadMarks, we consider this an acceptable performance penalty.

2. The performance of fine-grained applications is considerably (up to 98% for Barnes-Hut and 62% for Water-Spatial) better than in TreadMarks.

Among the various optimizations in DOSA, read aggregation is the most important one. It improves the program performance by up to 310%. Lazy object storage allocation also has a non-trivial impact on the fine-grained applications. It improves the performance of the two fine-grained applications by up to 18%, but has no impact on the coarse-grained applications. Write notice reduction and write aggregation can slightly (less than 10%) improve the performance of the applications that have a large number of shared objects. They do not have any negative impact on other applications.
Chapter 6

Evaluation: Garbage Collection

This chapter presents an evaluation of our solutions to garbage collection on DSM systems. Our evaluation seeks to substantiate the following claims:

1. Our new inter-processor garbage collection algorithm allows the overall program performance to approach that of manual memory management on DOSA, and is much better than state-of-the-art DSM garbage collectors.

2. By providing the shared object space abstraction, DOSA allows more efficient intra-processor garbage collections and better overall program performance than conventional DSM systems.

To substantiate the above claims, we have conducted two experiments using three programs modified from real applications. The first experiment evaluates the impact of our new inter-processor collector on overall program performance. We have implemented an inter-processor garbage collector using our adaptive algorithm (See Section 4.2) on DOSA. A generational copying collector is used for the intra-processor collection. We compare the performance of our inter-processor collector with the weighted reference counting algorithm and the stub-scion algorithm (See Sections 2.1.2). Our measurement shows that, with the adaptive collector, the overall program performance is up to 50% higher than with the other two algorithms, and is within 5% of manual memory management. Although our collector theoretically has higher communication and computation overheads, the overheads are insignificant in the test programs. Our evaluation also shows that asynchronous collections are more efficient than synchronous collections in DSM systems (See Section 4.3). In the adaptive algorithm, the asynchronous collections improves the program performance by up to 23% over the synchronous collections. Although we use only DOSA and the copying collector in the evaluation, we explain why the results also apply to other DSM systems and the mark-sweep collector.

Our second experiment evaluates the effect of the shared object space abstraction on intra-processor garbage collection. We implemented two intra-processor garbage
collectors on both TreadMarks and DOSA that are representative of those in common use. One of the garbage collectors is based on mark-sweep, the other is based on generational copying. Both collectors use the adaptive algorithm for inter-processor collections. Our evaluation shows that the shared object space abstraction in DOSA allows better (up to 41%) overall program performance than conventional DSM systems.

In this chapter, we describe the test programs in Section 6.1. The evaluation of the inter-processor collection algorithms is presented in Section 6.2, and the evaluation of the intra-processor collection algorithms is presented in Section 6.3. We summarize the results in Section 6.4.

### 6.1 Applications

In this chapter, we use the same programs described in Section 2.4.2. We manually inserted the handle-dereference code in the DOSA versions of the programs. Table 6.1 presents the sequential execution times and garbage collection times of the programs.

From the statistics in Table 6.1 we draw the following conclusions. First, the extra cost in the garbage collectors to manage the handles on DOSA is low. The largest percentage difference in GC time is between the TreadMarks' mark-sweep collector and its DOSA counterpart in BH (0.70s versus 0.75s), which is a 7.1% difference. In all other cases, the difference ranges between 2.8% and 6.7%. Second, on a single processor, the copying collector generally has lower GC cost than the mark-sweep collector. In Game, the copying collector outperforms the mark-sweep collector by about

<table>
<thead>
<tr>
<th></th>
<th>Manual Tmk</th>
<th>Manual DOSA</th>
<th>Mark-Sweep Tmk</th>
<th>Mark-Sweep DOSA</th>
<th>Copying Tmk</th>
<th>Copying DOSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Game</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>282</td>
<td>291</td>
<td>295</td>
<td>306</td>
<td>286</td>
<td>296</td>
</tr>
<tr>
<td>GC (Free) time</td>
<td>17.5</td>
<td>17.5</td>
<td>31.4</td>
<td>32.3</td>
<td>21.7</td>
<td>22.5</td>
</tr>
<tr>
<td>BH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>27.4</td>
<td>28.2</td>
<td>28.0</td>
<td>28.8</td>
<td>27.6</td>
<td>28.4</td>
</tr>
<tr>
<td>GC (Free) time</td>
<td>0.10</td>
<td>0.10</td>
<td>0.70</td>
<td>0.75</td>
<td>0.30</td>
<td>0.32</td>
</tr>
<tr>
<td>MIP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>583</td>
<td>600</td>
<td>583</td>
<td>600</td>
<td>583</td>
<td>600</td>
</tr>
<tr>
<td>GC (Free) time</td>
<td>0.01</td>
<td>0.01</td>
<td>0.21</td>
<td>0.22</td>
<td>0.20</td>
<td>0.22</td>
</tr>
</tbody>
</table>

Table 6.1 Running time and garbage collection time (in seconds) for TreadMarks and DOSA on 1 processor. For manual memory management, GC time is the time spent by free.
45% on both TreadMarks and DOSA. In BH, the copying collector outperforms the mark-sweep collector by 133% on both DSM systems. The mark-sweep and copying collectors have about equal performance in MIP. This is because there are many live objects in MIP at any time. MIP only perform a few collections, therefore the live objects remain in the youngest generation during the program execution, increasing the copying cost. Finally, the handle dereference cost is low. In all three programs, the DOSA versions of the programs underperform their TreadMarks counterparts by less than 4% on 1 processor.

6.2 Inter-Processor Garbage Collection

We have implemented an inter-processor garbage collector using our adaptive algorithm (see Section 4.2). In this section we evaluate its performance against two state-of-the-art DSM garbage collection algorithms: the weighted reference counting (WRT) algorithm and the stub-scion algorithm. All three algorithms have been implemented on DOSA, with a generational copying collector as the intra-processor collector. To put the performance of the garbage collectors into perspective, we also compare their performance with manual memory management by the programmer. In the evaluation, we use three programs that are modified from real applications. Our measurement seeks to substantiate the following claims:

- Our new collector allows better program performance because it eliminates the adverse side effects in the existing collectors: the delayed reclamation of shared data and the premature flushing of imported references.

- The extra costs in our graph-tracing collector, which include the computation cost to build and trace the object graph and the communication of the garbage collector data, are comparable with that of the WRT or Stub-scion collectors.

As a side note, our measurements will also show that asynchronous collections are more efficient than synchronous collections (See Section 4.3).

In the measurements, we break down the costs of the garbage collection into the following categories:

- Local garbage collection cost. This is the time spent by the intra-processor garbage collectors on each processor. In the measurement of our new collector, we also include the time spent by the processors to gather the extra GC data and to identify the garbage.
• The communication cost of the inter-processor garbage collectors. This is the amount of garbage collector data communicated in the system. This does not include any extra messages, since this data is piggybacked on program initiated messages. The WRT based collector must examine every outgoing message and append some extra data to the message. The appended data lists the references in the message as well as their weights. The stub-scion based collector and our adaptive collector do not have this cost. In all three garbage collectors, a processor must propagate the information gathered from an intra-processor collection to other processors. In the WRT based collector, this is the nack data, which lists the import references that are no longer reachable locally, and the weights of the references. In the stub-scion based collector, a processor sends out the list of live data to other processors after an intra-processor collection. In our adaptive collector, the processors send out the list of live data as well as the connectivity information to the lead processor.

• Synchronous global collection. This happens when there are cyclic garbage in the program.

• The garbage collectors' side effects on the whole program performance, e.g., thrashing (See Section 4.2.1). This is reflected in the total running time and the amount of communication.

6.2.1 Measurement

The results of the measurements are reported below. In all three programs, our adaptive collector used the graph-tracing algorithm and did not fall back to the stub-scion algorithm. Therefore, in the remainder of this section we refer to our collector simply as the graph-tracing collector. In the following tables, WRT refers to the weighted reference counting based collector, Stub refers to the stub-scion based collector, Graph refers to our new adaptive algorithm, and Manual refers to manual memory management.

Table 6.2 reports the total running times of the test programs with the three garbage collectors and with manual memory management. Figures 6.1 through 6.3 present the speedups of the three programs.

Among the three garbage collectors, the graph-tracing collector allows the best program performance. With the graph-tracing collector, the overall program performance is close (within 5%) to that of manual memory management on up to 32
<table>
<thead>
<tr>
<th></th>
<th>WRT</th>
<th>Stub</th>
<th>Graph</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Game</td>
<td>41.9</td>
<td>24.0</td>
<td>18.0</td>
<td>42.0</td>
</tr>
<tr>
<td>BH</td>
<td>9.92</td>
<td>9.34</td>
<td>8.90</td>
<td>9.55</td>
</tr>
<tr>
<td>MIP</td>
<td>83.6</td>
<td>56.7</td>
<td>52.7</td>
<td>83.7</td>
</tr>
</tbody>
</table>

Table 6.2 Total execution time on 8, 16 and 32 processors (in seconds).

<table>
<thead>
<tr>
<th></th>
<th>WRT</th>
<th>Stub</th>
<th>Graph</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running time (sec)</td>
<td>41.9</td>
<td>42.0</td>
<td>41.5</td>
<td>41.1</td>
</tr>
<tr>
<td>Local GC (Free) time (sec)</td>
<td>2.70</td>
<td>2.91</td>
<td>2.66</td>
<td>2.15</td>
</tr>
<tr>
<td>GC Data (MB)</td>
<td>1.63</td>
<td>0.68</td>
<td>0.23</td>
<td>0</td>
</tr>
<tr>
<td>Total data (MB)</td>
<td>13.0</td>
<td>12.2</td>
<td>11.7</td>
<td>11.4</td>
</tr>
<tr>
<td>Total msg</td>
<td>24.0K</td>
<td>24.0K</td>
<td>24.0K</td>
<td>24.0K</td>
</tr>
</tbody>
</table>

Table 6.3 Detailed statistics for Game on 8 processors.

<table>
<thead>
<tr>
<th></th>
<th>WRT</th>
<th>Stub</th>
<th>Graph</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running time (sec)</td>
<td>24.0</td>
<td>26.1</td>
<td>23.5</td>
<td>23.2</td>
</tr>
<tr>
<td>Local GC (Free) time (sec)</td>
<td>2.45</td>
<td>4.80</td>
<td>2.15</td>
<td>1.70</td>
</tr>
<tr>
<td>GC Data (MB)</td>
<td>3.22</td>
<td>1.01</td>
<td>0.34</td>
<td>0</td>
</tr>
<tr>
<td>Total data (MB)</td>
<td>19.2</td>
<td>17.1</td>
<td>16.4</td>
<td>16.2</td>
</tr>
<tr>
<td>Total msg</td>
<td>34.8K</td>
<td>34.8K</td>
<td>34.8K</td>
<td>34.8K</td>
</tr>
</tbody>
</table>

Table 6.4 Detailed statistics for Game on 16 processors.

<table>
<thead>
<tr>
<th></th>
<th>WRT</th>
<th>Stub</th>
<th>Graph</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running time (sec)</td>
<td>18.0</td>
<td>18.2</td>
<td>15.8</td>
<td>15.4</td>
</tr>
<tr>
<td>Local GC (Free) time (sec)</td>
<td>3.04</td>
<td>3.71</td>
<td>1.48</td>
<td>1.20</td>
</tr>
<tr>
<td>GC Data (MB)</td>
<td>5.51</td>
<td>1.23</td>
<td>0.43</td>
<td>0</td>
</tr>
<tr>
<td>Total data (MB)</td>
<td>34.5</td>
<td>30.4</td>
<td>29.7</td>
<td>29.2</td>
</tr>
<tr>
<td>Total msg</td>
<td>69.8K</td>
<td>69.8K</td>
<td>69.8K</td>
<td>69.8K</td>
</tr>
</tbody>
</table>

Table 6.5 Detailed statistics for Game on 32 processors.
Figure 6.1 The speedup of Game.

Figure 6.2 The speedup of BH.

Figure 6.3 The speedup of MIP.

Figure 6.4 Retained garbage in Game with the graph-tracing collector.
processors. In contrast, the WRT and stub-scion collectors do not scale as well. They start to underperform the graph-tracing collector on 16 processors in Game and MIP, and on 8 processors in BH. On 32 processors, the performance difference between the graph-tracing collector and the other two collectors ranges between 14% (in Game) and 50% (in BH).

The performance difference between the three garbage collectors comes from four sources: the local collection cost, the synchronous global collection cost (MIP only), the amount of garbage collector data communicated, and the interaction between the garbage collector and the DSM system.

Tables 6.3 through 6.5 report the detailed statistics for Game on 8, 16, and 32 processors. In these tables, Running time is the total program execution time in seconds. Local GC time is the intra-processor garbage collection time. For the graph-tracing collector, it also includes the time spent by each processor to build and trace the adjacency lists. For manual memory management, this is the time spent by the programs to free the objects. GC data is the amount of garbage collector data exchanged between the processors. In the WRT based collector, the GC data consists of two parts: the weights transferred, and the Nack data, which lists the references that have been removed from a processor and their weights. In the stub-scion based collector, the GC data is the live reference list exchanged by the processors (See
Section 2.2.2). In the graph-tracing collector, the GC data includes the reachable reference list and the adjacency list (See Section 4.2).

In Game, the WRT and stub-scion algorithms result in lower program performance than the graph-tracing collector for two reasons: the higher intra-processor collection costs due to the delayed garbage reclamation, and the higher amount of garbage collector data exchanged between the processors.

The WRT and the stub-scion algorithms result in higher garbage collection cost because they fail to reclaim shared objects quickly. From Table 6.5 we can see that on 32 processors, the intra-processor garbage collection time in the WRT based collector is 105% (3.04 seconds versus 1.48 seconds) more than that of the graph-tracing collector, accounting for 71% of the performance difference between the two collectors. In the stub-scion collector, the intra-processor collection time is 151% (3.71 versus 1.48 seconds) more than that of the graph-tracing collector, accounting for almost all of the performance difference.

As we have said in Section 2.4.3, the WRT algorithm results in long export chains in Game. We have observed that the export chains of the root reference has 7 links on average. This delay increases the size of the live data and the intra-processor garbage collection cost. The generational collector does not help because the live data size is large enough so that the older generation is included in almost every garbage collection. This problem is more severe in the stub-scion based collector. On 32 processors, the length of the export chains in Game is almost always 31. Therefore the size of the retained garbage is even larger. This explains why the intra-processor collection time in the stub-scion collector is much more than the time in the WRT based collector.

The effect of the delayed reclamation is less serious with fewer processors. For example, with the WRT collector, the program performance is close (within 2%) to that with the graph-tracing collector on 8 and 16 processors. With the Stub-scion collector, the performance is within 2% of that with the graph-tracing collector on 8 processors. This is because with fewer processors, not only the length of the export chain is shorter, each processor also performs more intra-processor garbage collections. Therefore the liveness information (nack or the live list) is propagated much faster.

Figures 6.4 through 6.6 shows the amount of retained garbage (See Section 2.4.3) in Game with the three inter-processor collection algorithms. We measured the size of the retained garbage at 10 points during the program execution. From the figures we can see that our graph-tracing collector retains the least amount of garbage. On up to
32 processors, it retains no more than 512K bytes of garbage. More importantly, the amount of retained garbage becomes stable very early in the program execution. In contrast, with the WRT or the stub-scion algorithm, the amount of retained garbage keeps growing for much longer time. This is because in these two algorithms the garbage reclamation delay is determined by the length of the export chains, while the export chain length has no effect on the garbage reclamation delay in the graph-tracing collector.

The conventional wisdom says that because the WRT and stub-scion propagate the object liveness information asynchronously through pair-wise communication, they reduce the communication cost and should scale well with the number of processors [PS95]. However, the trend in the effect of the delayed reclamation in our measurement shows that the amount of garbage they retain increases with the number of processors, creating a new problem to their scalability.

In Game, the graph-tracing collector also sends less GC data than the WRT and stub-scion collectors. From Table 6.5 we can see that on 32 processors, the WRT based collector sends 5.5M bytes of GC data, and the stub-scion collector sends 1.23M bytes of GC data. In contrast, the graph-tracing data only sends 0.43M bytes of GC data. The amount of GC data in the graph-tracing collector is small because the size of the shared trees built in each iteration is relatively small. The GC data in the WRT based collector is about equally divided between the weights appended to every outgoing messages and the nack data which lists the "dead" references and their weights (2.87M bytes and 2.64M bytes, respectively on 32 processors). The other two collectors avoid the first cost altogether, resulting in smaller GC data sizes. It is interesting to note that the stub-scion collector actually sends more data than the graph-tracing collector. This is because in the stub-scion collector each processor must propagate its live list to many other processors, while in the graph-tracing collector each processor only sends the GC data to the leader processor.

In Game, the intra-processor collection time in the graph-tracing collector is close (within 26%) to the time spent in free under manual memory management. This is because Game allocates a large number of short-lived small objects while having a relatively small amount of long-lived objects. The overall program performance with the graph-tracing collector approaches (within 3%) that of programmer memory management.

The statistics for BH on 8, 16, and 32 processors is presented in Tables 6.6 through 6.8. With the graph-tracing collector, the program performance is within
<table>
<thead>
<tr>
<th></th>
<th>WRT</th>
<th>Stub</th>
<th>Graph</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running time (sec)</td>
<td>9.92</td>
<td>9.55</td>
<td>6.75</td>
<td>6.46</td>
</tr>
<tr>
<td>Local GC (Free) time (sec)</td>
<td>0.30</td>
<td>0.30</td>
<td>0.32</td>
<td>0.10</td>
</tr>
<tr>
<td>GC Data (MB)</td>
<td>1.52</td>
<td>0.68</td>
<td>0.14</td>
<td>0</td>
</tr>
<tr>
<td>Total data (MB)</td>
<td>21.2</td>
<td>20.3</td>
<td>19.7</td>
<td>19.5</td>
</tr>
<tr>
<td>Total msg</td>
<td>30.1K</td>
<td>30.1K</td>
<td>17.4K</td>
<td>17.4K</td>
</tr>
</tbody>
</table>

**Table 6.6** Detailed statistics for BH on 8 processors.

<table>
<thead>
<tr>
<th></th>
<th>WRT</th>
<th>Stub</th>
<th>Graph</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running time (sec)</td>
<td>9.34</td>
<td>9.03</td>
<td>6.15</td>
<td>5.92</td>
</tr>
<tr>
<td>Local GC (Free) time (sec)</td>
<td>0.30</td>
<td>0.30</td>
<td>0.32</td>
<td>0.10</td>
</tr>
<tr>
<td>GC Data (MB)</td>
<td>2.50</td>
<td>1.24</td>
<td>0.18</td>
<td>0</td>
</tr>
<tr>
<td>Total data (MB)</td>
<td>30.4</td>
<td>29.1</td>
<td>28.0</td>
<td>27.8</td>
</tr>
<tr>
<td>Total msg</td>
<td>50.3K</td>
<td>50.3K</td>
<td>31.7K</td>
<td>31.7K</td>
</tr>
</tbody>
</table>

**Table 6.7** Detailed statistics for BH on 16 processors.

<table>
<thead>
<tr>
<th></th>
<th>WRT</th>
<th>Stub</th>
<th>Graph</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running time (sec)</td>
<td>8.90</td>
<td>8.65</td>
<td>5.75</td>
<td>5.47</td>
</tr>
<tr>
<td>Local GC (Free) time (sec)</td>
<td>0.30</td>
<td>0.31</td>
<td>0.32</td>
<td>0.10</td>
</tr>
<tr>
<td>GC Data (MB)</td>
<td>3.40</td>
<td>1.98</td>
<td>0.24</td>
<td>0</td>
</tr>
<tr>
<td>Total data (MB)</td>
<td>52.4</td>
<td>50.9</td>
<td>49.3</td>
<td>49.0</td>
</tr>
<tr>
<td>Total msg</td>
<td>74.1K</td>
<td>74.1K</td>
<td>42.7K</td>
<td>42.7K</td>
</tr>
</tbody>
</table>

**Table 6.8** Detailed statistics for BH on 32 processors.
5.1% of that with manual management in all cases. With the WRT and stub-scion algorithms, however, the program performance is between 46% (on 8 processors) and 58% lower (on 32 processors) than manual memory management. The main cause of the underperformance is the problem of thrashing (See Section 4.2.1).

The WRT and stub-scion collectors suffer from the thrashing effect because they cannot identify garbage accurately. In BH, the particles move in each iteration. As a result, the shape of the tree also changes. Because each processor only reads part of the tree, a particle may be locally unreachable on a processor even though it is not garbage. If a collection happens on a processor while a particle appears to be unreachable, the WRT and the stub-scion collectors will reclaim the local storage for the particle. We call this a flush. If later the processor accesses the same particle, the particle will have to be fetched from a remote processor again, resulting in thrashing. Thrashing is particularly bad on DOSA, because there is no aggregation benefit when fetching such objects (See Section 4.2.1). Therefore, each object must be fetched with a separate message, increasing the total number of messages. From Table 6.8 we can see that on 32 processors the WRT and the stub-scion collectors increase the total number of messages by more than 70% (74.1K versus 42.7K). The number of messages is also significantly higher than the graph-tracing collector in the cases of 8 or 16 processors.

The graph-tracing collector also sends less GC data than the other two collectors. It only sends about 0.24M bytes of data, while the WRT collector and the stub-scion collector send 3.40M bytes and 1.98M bytes, respectively. The graph-tracing collector sends little data because in BH the tree nodes are only written by a single processor. Therefore all the other processors only have to send the few top level tree nodes that are reachable from their local roots. They do not have to send the adjacency lists.

The detailed statistics for MIP on 8, 16, and 32 processors is presented in Tables 6.9 through 6.11 In MIP, the graph-tracing collector also allows the best overall program performance among the three inter-processor collection algorithms. With the graph-tracing collector, the program performance is within 1% of that of manual memory management. With the WRT and stub-scion collectors, the program performance gradually becomes lower with the increase in the number of processors. On 16 processors, they underperform by about 6.6%. And on 32 processors, they underperform by 15.8%.

The main reason why the WRT and stub-scion collectors underperform the graph-tracing collector is that they cannot reclaim cycles of garbage, therefore they have to
<table>
<thead>
<tr>
<th></th>
<th>WRT</th>
<th>Stub</th>
<th>Graph</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running time (sec)</td>
<td>83.6</td>
<td>83.7</td>
<td>82.4</td>
<td>82.3</td>
</tr>
<tr>
<td>Local GC (Free) time (sec)</td>
<td>0.20</td>
<td>0.20</td>
<td>0.22</td>
<td>0.00</td>
</tr>
<tr>
<td>GC Data (MB)</td>
<td>1.00</td>
<td>0.49</td>
<td>0.36</td>
<td>0</td>
</tr>
<tr>
<td>Total data (MB)</td>
<td>39.5</td>
<td>39.0</td>
<td>38.5</td>
<td>38.2</td>
</tr>
<tr>
<td>Total msg</td>
<td>49.6K</td>
<td>49.6K</td>
<td>45.4K</td>
<td>45.4K</td>
</tr>
<tr>
<td>Sync global GC time (sec)</td>
<td>1.20</td>
<td>1.20</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sync global GC data (MB)</td>
<td>0.20</td>
<td>0.20</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sync global GC msg</td>
<td>4.2K</td>
<td>4.2K</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 6.9** Detailed statistics for MIP on 8 processors.

<table>
<thead>
<tr>
<th></th>
<th>WRT</th>
<th>Stub</th>
<th>Graph</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running time (sec)</td>
<td>56.7</td>
<td>56.7</td>
<td>53.5</td>
<td>53.2</td>
</tr>
<tr>
<td>Local GC (Free) time (sec)</td>
<td>0.21</td>
<td>0.21</td>
<td>0.22</td>
<td>0.00</td>
</tr>
<tr>
<td>GC Data (MB)</td>
<td>2.13</td>
<td>1.16</td>
<td>0.62</td>
<td>0</td>
</tr>
<tr>
<td>Total data (MB)</td>
<td>82.2</td>
<td>82.0</td>
<td>80.5</td>
<td>79.7</td>
</tr>
<tr>
<td>Total msg</td>
<td>87.5K</td>
<td>87.5K</td>
<td>76.7K</td>
<td>76.7K</td>
</tr>
<tr>
<td>Sync global GC time (sec)</td>
<td>2.80</td>
<td>2.80</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sync global GC data (MB)</td>
<td>0.42</td>
<td>0.42</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sync global GC msg</td>
<td>10.9K</td>
<td>10.9K</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 6.10** Detailed statistics for MIP on 16 processors.

<table>
<thead>
<tr>
<th></th>
<th>WRT</th>
<th>Stub</th>
<th>Graph</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running time (sec)</td>
<td>52.7</td>
<td>53.0</td>
<td>45.8</td>
<td>45.5</td>
</tr>
<tr>
<td>Local GC (Free) time (sec)</td>
<td>0.21</td>
<td>0.21</td>
<td>0.22</td>
<td>0.00</td>
</tr>
<tr>
<td>GC Data (MB)</td>
<td>2.26</td>
<td>2.05</td>
<td>0.77</td>
<td>0</td>
</tr>
<tr>
<td>Total data (MB)</td>
<td>116</td>
<td>118</td>
<td>115</td>
<td>113</td>
</tr>
<tr>
<td>Total msg</td>
<td>122K</td>
<td>122K</td>
<td>99.5K</td>
<td>99.5K</td>
</tr>
<tr>
<td>Sync global GC time (sec)</td>
<td>6.60</td>
<td>6.60</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sync global GC data (MB)</td>
<td>0.79</td>
<td>0.79</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sync global GC msg</td>
<td>22.8K</td>
<td>22.8K</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 6.11** Detailed statistics for MIP on 32 processors.
invoke costly synchronous collections. From Tables 6.9 through 6.11 we can see that the synchronous collection time almost equals the performance difference between the three collectors in all three cases. Also, the synchronous global collection time is proportional to the number of processors. It increases from 1.20 seconds on 8 processors to 6.60 seconds on 32 processors. We can also see that the amount of data communicated during the synchronous collections is not large, but the number of messages is high. There are two reasons for the higher number of messages. First, because every processor may insert new tasks into the queue, the task queue is broken into many small segments. For each processor, a segment is a group of tasks that are contiguous in the queue and have up to date copies on the processor. In a synchronous collection, whenever a processor traces to the end of a segment (i.e. find a reference to an object which does not have a consistent copy on the processor), it has to send a mark message (See Section 2.1.1) to other processors. Second, because of data replication in DSM systems, a processor does not always know exactly which processor has the up to date copy of an object. Therefore it may take several hops before the mark message arrives at the correct processor. This also increases the number of messages.

Besides the amount of communication, synchronous global collections are expensive in MIP for another reason. The shared objects in MIP form a single cross-processor chain. At any time, there are only a small number of processors that are actively tracing while other processors are idling and waiting for mark messages to arrive. There are algorithms that perform the global tracing asynchronously [Ben87, LQP92], but such algorithms will not help MIP: Because of the long cross-processor chains, the asynchronous global tracing will not complete before the processors run out of memory.

In MIP, The graph-tracing collector spends 5% more time in intra-processor collection than the WRT and the stub-scion collectors on 32 processors. This shows that the cost to build and trace the adjacency lists in the graph-tracing collector is small. In MIP, the stub-scion collector sends the most GC data. This is because most of the time during the program execution there are a large number of live objects. With the stub-scion collector each processor must send its live references to many other processors, resulting in larger GC data size. Like in BH, the total garbage collection time is only a small fraction of the total running time.
Synchronous versus Asynchronous Collection

The graph-tracing collector evaluated above works in the asynchronous mode: each processor performs intra-processor collections independently, and sends out the information it has compiled by piggybacking them on messages initiated by the program execution. To evaluate the benefit of the asynchronous collections, we have compared its performance with a synchronous version of the same collector.

Our measurement shows that there is a big performance gap between the synchronous and asynchronous collections when the number of processors is large. On 8 processors, there are virtually no difference in overall program performance between the synchronous and asynchronous collectors. On 32 processors, however, Game completes in 19.4 seconds with the synchronous collector, BH completes in 6.30 seconds, and MIP completes in 47.9 seconds, a slowdown of 23%, 9.6% and 4.5%, respectively. The synchronous and the asynchronous versions of the new garbage collector send the same amount of data, and the synchronous collector only sends a few hundred (about 1%) more messages. The times spent in intra-processor garbage collections by the two collectors are also the same. The cause of the performance difference is in the different ways in which the garbage collector data is exchanged. In the synchronous collector, every processor performs an intra-processor garbage collection, then sends the data to the leader processor. Sending the data synchronously causes a lot of contention on the leader processor. In our measurement, we have found that it takes about 0.1 seconds for the leader to receive a 1 Kbyte message (the average size of the garbage collector messages) from the other 31 processors. It only gets worse with larger messages: it takes 0.9 seconds if the message size is 40 Kbytes. In the asynchronous collector, the garbage collector data is piggybacked in the DSM messages, which arrive at their destinations asynchronously, eliminating the problem of contention.

The Cost of Centralized Tracing

In Section 4.2.2 we argued that the centralized tracing in our algorithm is unlikely to become a performance bottleneck. First, the number of objects traced in our algorithm may not be higher than in the WRT or the stub-scion algorithms because the latter two retain many dead objects. Second, if the object graph is small, the tracing cost does not have much impact on the overall program performance. If the object graph is too large, our algorithm will avoid the centralized tracing cost by
falling back to the stub-scion algorithm. The measurements in this section validated our arguments.

Our measurements validated our first argument. In Game, for example, the WRT and stub-scion algorithms retain a lot of garbage. From Figures 6.4 through 6.6 we can see that the amount of retained garbage is not trivial. In fact, in this program, the amount of the retained garbage exceeds that of the live data. Therefore, the tracing cost in our algorithm is lower than in the other two algorithms.

Our measurements also validated our second argument. In all three programs, the intra-processor garbage collection time (which includes the tracing cost) in our algorithm is no more than 7% higher over that of the WRT and stub-scion algorithms. This shows that the tracing cost is not a performance bottleneck in these programs.

Summary

To summarize, the graph-tracing collector allows better (up to 50%) program performance than the previous collectors. It also performs within 5% of that of the programmer memory management. It achieves better performance by more accurate and timely garbage identification and reclamation, and by gathering the liveness information asynchronously. By identifying garbage more accurately, it avoids the costly synchronous global collection and the thrashing problem. By reclaiming garbage in a more timely fashion, it retains less garbage and reduces the cost of the intra-processor collections. By exchanging the GC data asynchronously, it avoids contention in large-scale systems.

The results also show that the extra cost in the graph-tracing collector is low. The extra time spent by our collector to build and trace the adjacency lists is less than 7% of the garbage collection time. And its effect on the overall program performance is negligible. Although the graph-tracing collector theoretically may send more GC data than the WRT and stub-scion collectors, it actually sends less data in the three programs we evaluated, which use the data structures most commonly found in real parallel programs.

6.2.2 Discussion

Our evaluation has been done on DOSA, with a generational copying collector as the intra-processor collector. We conclude that the graph-tracing collector allows better performance than the previous inter-processor collectors. Here we argue that the
conclusion also holds for conventional DSM systems such as TreadMarks and for the mark-sweep collector.

Our evaluation shows that the graph-tracing collector solves two problems in the other inter-processor collectors on DOSA. First, it reclaims shared garbage, including cycles, earlier. Second, it eliminates the thrashing problem. The first problem is solely the result of the inter-processor collection algorithm and is orthogonal to the DSM system. Therefore, it will also exist in conventional DSM systems. The thrashing problem also exists in conventional DSM systems such as TreadMarks. However, the cost of thrashing is much smaller than in DOSA. Thrashing may still increase the amount of GC data, but it has no effect on the number of messages because the conventional DSM systems do not perform lazy allocation. In BH, which is the only test program that has the thrashing problem, the program performance with the graph-tracing collector is within 5% of the manual memory management on DOSA. This shows that the graph-tracing collector does not incur much extra cost to solve the thrashing problem. Therefore its use will not be harmful on conventional DSM systems.

To summarize, on conventional DSM systems the graph-tracing collector will be beneficial in some programs (such as Game or MIP), and will not be harmful in other programs. Overall, it is a better alternative than the previous inter-processor collectors for DSM systems.

We believe that the graph-tracing collector will also prove to be a better algorithm if the mark-sweep collector is used in the evaluation. The problems of delayed reclamation and thrashing are the results of the inter-processor collector and the DSM design. They are orthogonal to the intra-processor collector implementation. Therefore the use of a mark-sweep collector will not change the results in the evaluation.

6.2.3 Summary

We have evaluated the performance of our new adaptive collector using three programs modified from real applications. We compared its performance with two garbage collectors that are representative of those currently available. To put the program performance in perspective, we also compared it with explicit memory management.

Our evaluation substantiates the following claims:

1. Our collector eliminates the adverse side effects on program performance: it avoids the problems of delayed reclamation, cyclic garbage, and thrashing. It
improves the overall program performance by up to 50%, and performs within 5% of that of programmer memory management.

2. Our collector is efficient when compared with existing collectors: the extra computation cost to build and trace the object graph has negligible impact on the program performance; and the amount of GC data is often less than that in the other collectors.

### 6.3 Intra-Processor Collection

In this section we evaluate the benefit of DOSA on intra-processor garbage collection. We have implemented two intra-processor garbage collection algorithms on both TreadMarks and DOSA that are representative of those in common use. The first algorithm is based on mark-sweep and the second algorithm is based on generational copying. Our adaptive algorithm is used for the inter-processor collection. We compared the performance of these garbage collectors with that of manual memory management by the programmer.

#### 6.3.1 Measurements

**Performance Overview**

Figures 6.7 through 6.12 present the speedups of the test programs on TreadMarks and DOSA with different garbage collectors. Table 6.12 presents the total running time of the three programs on TreadMarks and DOSA on 32 processors. By comparing the performance of a garbage-collected program with that of the manual memory

<table>
<thead>
<tr>
<th></th>
<th>Manual Tmk</th>
<th>DOSA</th>
<th>Mark-Sweep Tmk</th>
<th>DOSA</th>
<th>Copying Tmk</th>
<th>DOSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Game</td>
<td>15.6</td>
<td>15.4</td>
<td>17.2</td>
<td>17.0</td>
<td>19.4</td>
<td>15.8</td>
</tr>
<tr>
<td>BH</td>
<td>7.58</td>
<td>5.47</td>
<td>7.91</td>
<td>5.80</td>
<td>6.00</td>
<td>5.75</td>
</tr>
<tr>
<td>MIP</td>
<td>50.1</td>
<td>45.5</td>
<td>50.3</td>
<td>45.8</td>
<td>64.5</td>
<td>45.8</td>
</tr>
</tbody>
</table>

*Table 6.12* Total running time (in seconds) on 32 processors.
Figure 6.7  The speedup of Game with the mark-sweep collectors.

Figure 6.8  The speedup of Game with the copying collector.

Figure 6.9  The speedup of BH with the mark-sweep collector.

Figure 6.10  The speedup of BH with the copying collector.
management on the same system, we can find out the impact of the garbage collector on the DSM system performance. We make the following observations from the results on 32 processors. First, with either the mark-sweep or the copying collector, DOSA allows better program performance than TreadMarks. With the mark-sweep collector, DOSA outperforms TreadMarks by up to 36%. With the copying collector, DOSA outperforms TreadMarks by up to 41%. Second, the impact of the copying collector is small on DOSA, but not so small on TreadMarks. With the copying collector on DOSA, the program performance is within 5% of that of the manual memory management. On TreadMarks, however, the performance difference ranges between a 26% improvement in BH to a 29% slowdown in MIP. Finally, we note that the copying collector on DOSA allows better program performance than its TreadMarks counterpart in all three programs.

**Detailed Analysis**

Tables 6.13 through 6.15 present the detailed statistics of Game on different number of processors. *GC data* is the amount of garbage collector data exchanged between the processors. During each step in Game, the processors synchronize through locks. Therefore if a processor wants to start a copying collection on TreadMarks, it must interrupt the other processors. *Suspend wait* is the sum of the delay between the time when an interrupt message is sent and the time when the entire system is suspended.
Suspend data is a TreadMarks specific overhead in the copying collector. At the beginning of a copying collection, the system must designate an owner for each page and bring the owner's copy of the page up to date (see Section 2.3.2). The Addr Update Msg is the number of messages needed for the processors to update their references to the moved objects, and the Addr Update Data is the amount of new references propagated between the processors.

Game has similar performance on both TreadMarks and DOSA with either manual memory management or the mark-sweep collector. However, with the copying collector, the performance on TreadMarks is up to 23% (17.6 versus 13.9 seconds) lower than on DOSA. In fact, with the copying collector on TreadMarks, the program performance is lower than that of the mark-sweep collector despite its better performance in the intra-processor garbage collections. This underperformance results from the extra cost to perform copying collections on TreadMarks.

The suspension time increases with the number of processors. From Tables 6.13 through 6.15 we can see that the time to suspend the entire system increases from 0.12 seconds on 8 processors to 0.44 seconds on 32 processors. This is the sum over 20 collections. The Suspend data field is 0 because in this program all writes to the shared objects are synchronized by a lock. Therefore there is a unique last writer for each page at the beginning of each copying collection, and the arbitration mechanism (see Section 2.3.2) is not invoked. The suspension time (0.44 seconds on 32 processors) accounts for about 12% of the total difference between the TreadMarks and DOSA using the copying collector.

The impact of the bogus writes is reflected in the total amount of data communicated. From the tables we can see that with the copying collector TreadMarks sends much more data than DOSA. On 32 processors, after subtracting the GC data from the total, TreadMarks sends 50.3M bytes of program data, while DOSA sends 29.3M bytes of program data. This is a 71% difference. The address updates involve 9900 messages and 1.2M bytes of data. It is difficult to isolate the effect of each of the two factors on the running time. Together they account for 88% of the performance difference between the two systems.

The delayed reuse of the fromspace results in larger memory requirement. For this program, the copying collector on TreadMarks uses 6 semispaces (3 semispaces for each generation) totaling 24M bytes. On DOSA, the copying collector only needs 4 semispaces totaling 16M bytes.
<table>
<thead>
<tr>
<th></th>
<th>Manual Tmk</th>
<th>Manual DOSA</th>
<th>Mark-&amp;-Sweep Tmk</th>
<th>Mark-&amp;-Sweep DOSA</th>
<th>Copying Tmk</th>
<th>Copying DOSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec)</td>
<td>41.1</td>
<td>41.1</td>
<td>43.5</td>
<td>43.5</td>
<td>42.6</td>
<td>41.5</td>
</tr>
<tr>
<td>GC (Free) time (sec)</td>
<td>2.15</td>
<td>2.15</td>
<td>4.55</td>
<td>4.60</td>
<td>2.66</td>
<td>2.66</td>
</tr>
<tr>
<td>Total Data (MB)</td>
<td>12.4</td>
<td>11.4</td>
<td>12.4</td>
<td>11.7</td>
<td>18.3</td>
<td>11.7</td>
</tr>
<tr>
<td>Overlapped data requests</td>
<td>24.7</td>
<td>24.0</td>
<td>24.7</td>
<td>24.0</td>
<td>26.6</td>
<td>24.0</td>
</tr>
<tr>
<td>GC data (MB)</td>
<td>0</td>
<td>0.23</td>
<td>0.23</td>
<td>0.23</td>
<td>0.23</td>
<td>0.23</td>
</tr>
<tr>
<td>Suspend wait (sec)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.12</td>
<td>0</td>
</tr>
<tr>
<td>Suspend data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Addr Update Msg</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2.40K</td>
<td>0</td>
</tr>
<tr>
<td>Addr Update Data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.32</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.13  Detailed statistics for Game on 8 processors.

<table>
<thead>
<tr>
<th></th>
<th>Manual Tmk</th>
<th>Manual DOSA</th>
<th>Mark-&amp;-Sweep Tmk</th>
<th>Mark-&amp;-Sweep DOSA</th>
<th>Copying Tmk</th>
<th>Copying DOSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec)</td>
<td>23.3</td>
<td>23.2</td>
<td>25.3</td>
<td>25.1</td>
<td>25.3</td>
<td>23.5</td>
</tr>
<tr>
<td>GC (Free) time (sec)</td>
<td>1.72</td>
<td>1.72</td>
<td>3.70</td>
<td>3.73</td>
<td>2.15</td>
<td>2.15</td>
</tr>
<tr>
<td>Total Data (MB)</td>
<td>17.6</td>
<td>16.2</td>
<td>17.6</td>
<td>16.4</td>
<td>27.1</td>
<td>16.4</td>
</tr>
<tr>
<td>Overlapped data requests</td>
<td>35.2</td>
<td>34.8</td>
<td>35.2</td>
<td>34.8</td>
<td>39.7</td>
<td>34.8</td>
</tr>
<tr>
<td>GC data (MB)</td>
<td>0</td>
<td>0.34</td>
<td>0.34</td>
<td>0.34</td>
<td>0.34</td>
<td>0.34</td>
</tr>
<tr>
<td>Suspend wait (sec)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.25</td>
<td>0</td>
</tr>
<tr>
<td>Suspend data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Addr Update Msg</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4.80K</td>
<td>0</td>
</tr>
<tr>
<td>Addr Update Data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.60</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.14  Detailed statistics for Game on 16 processors.

<table>
<thead>
<tr>
<th></th>
<th>Manual Tmk</th>
<th>Manual DOSA</th>
<th>Mark-&amp;-Sweep Tmk</th>
<th>Mark-&amp;-Sweep DOSA</th>
<th>Copying Tmk</th>
<th>Copying DOSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec)</td>
<td>15.6</td>
<td>15.4</td>
<td>16.8</td>
<td>16.7</td>
<td>19.4</td>
<td>15.8</td>
</tr>
<tr>
<td>GC (Free) time (sec)</td>
<td>1.20</td>
<td>1.20</td>
<td>2.30</td>
<td>2.35</td>
<td>1.46</td>
<td>1.48</td>
</tr>
<tr>
<td>Total Data (MB)</td>
<td>31.0</td>
<td>29.2</td>
<td>31.0</td>
<td>29.7</td>
<td>51.5</td>
<td>29.7</td>
</tr>
<tr>
<td>Overlapped data requests</td>
<td>71.0</td>
<td>69.8</td>
<td>71.0</td>
<td>69.8</td>
<td>79.7</td>
<td>69.8</td>
</tr>
<tr>
<td>GC data (MB)</td>
<td>0</td>
<td>0.43</td>
<td>0.43</td>
<td>0.43</td>
<td>0.43</td>
<td>0.43M</td>
</tr>
<tr>
<td>Suspend wait (sec)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.44</td>
<td>0</td>
</tr>
<tr>
<td>Suspend data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Addr Update Msg</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9.9K</td>
<td>0</td>
</tr>
<tr>
<td>Addr Update Data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.2M</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.15  Detailed statistics for Game on 32 processors.
Game shows that DOSA is able to take advantage of the better performance of the copying collector to improve the overall program performance. While on TreadMarks the benefit of the copying collector is more than offset by its additional costs.

The detailed statistics for BH is presented in Tables 6.16 through 6.18. Like in Section 2.4.3, the performance of the copying collector on TreadMarks is manually improved. The copying collector uses program information to identify the last writer at the object granularity. We perform this optimization because we want to identify the copying collection cost that is common to all conventional DSM systems. Without this manual optimization the performance of the copying collector on TreadMarks is about the same (within 2%) as that of the mark-sweep collector on TreadMarks.

With the copying collector, the program performance on TreadMarks is within 4% of the performance on DOSA. This difference is much smaller than that in Game. As we have explained in Section 2.4.3, the copying collector on TreadMarks avoids the costs of suspension because BH is a barrier-based program. The cost of bogus writes and address update is also low, due to the generational approach.

The delayed reclamation of the fromspace increases the size of the memory footprint from 4.4M bytes to 6.0M bytes.

With the copying collector on TreadMarks, BH still sends more messages and program data than DOSA. After excluding the address update cost, TreadMarks sends 1.3K more messages and 3M bytes more data than DOSA. This is because on TreadMarks the memory layout is not optimal. For example, assume processor $P_1$ writes objects $O_1$ and $O_2$, and $P_2$ reads those two objects. On TreadMarks, $P_2$ can fetch both objects with a single message only if $P_1$ has placed both objects in one page. On DOSA, $P_2$ can fetch them with a single message as long as $P_2$ itself has placed the two objects in one page in its local memory. Since in both systems the copying collector on each processor moves the objects only according to the partial object graph cached on the processor, on TreadMarks it is unlikely that a single processor can produce the optimal memory layout for all other processors. DOSA avoids this problem and reduces the amount of communication.

The detailed statistics for MIP is presented in Tables 6.19, 6.20, and 6.21. When running on TreadMarks with a copying collector, we manually zero out the references in the dead objects. This is necessary because with a copying collector for the intra-processor collections, the graph-tracing algorithm used for the inter-processor collections cannot reclaim the cyclic garbage (See Section 4.2). Without the manual
<table>
<thead>
<tr>
<th></th>
<th>Manual</th>
<th>Mark-&amp;-Sweep</th>
<th>Copying</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tmk</td>
<td>DOSA</td>
<td>Tmk</td>
</tr>
<tr>
<td>Time (sec)</td>
<td>8.54</td>
<td>6.46</td>
<td>8.85</td>
</tr>
<tr>
<td>GC (Free) time (sec)</td>
<td>0.10</td>
<td>0.10</td>
<td>0.70</td>
</tr>
<tr>
<td>Total Data (MB)</td>
<td>58.3</td>
<td>19.5</td>
<td>58.3</td>
</tr>
<tr>
<td>Overlapped data requests</td>
<td>35.1</td>
<td>17.4</td>
<td>35.1</td>
</tr>
<tr>
<td>GC data</td>
<td>0</td>
<td>0</td>
<td>0.14</td>
</tr>
<tr>
<td>Suspend wait (sec)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Suspend data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Addr Update Msg</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Addr Update Data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.16  Detailed statistics for BH on 8 processors.

<table>
<thead>
<tr>
<th></th>
<th>Manual</th>
<th>Mark-&amp;-Sweep</th>
<th>Copying</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tmk</td>
<td>DOSA</td>
<td>Tmk</td>
</tr>
<tr>
<td>Time (sec)</td>
<td>8.07</td>
<td>5.92</td>
<td>8.36</td>
</tr>
<tr>
<td>GC (Free) time (sec)</td>
<td>0.10</td>
<td>0.10</td>
<td>0.70</td>
</tr>
<tr>
<td>Total Data (MB)</td>
<td>85.4</td>
<td>27.8</td>
<td>85.4</td>
</tr>
<tr>
<td>Overlapped data requests</td>
<td>71.3</td>
<td>31.7</td>
<td>71.3</td>
</tr>
<tr>
<td>GC data</td>
<td>0</td>
<td>0</td>
<td>0.18</td>
</tr>
<tr>
<td>Suspend wait (sec)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Suspend data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Addr Update Msg</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Addr Update Data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.17  Detailed statistics for BH on 16 processors.

<table>
<thead>
<tr>
<th></th>
<th>Manual</th>
<th>Mark-&amp;-Sweep</th>
<th>Copying</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tmk</td>
<td>DOSA</td>
<td>Tmk</td>
</tr>
<tr>
<td>Time (sec)</td>
<td>7.58</td>
<td>5.47</td>
<td>7.91</td>
</tr>
<tr>
<td>GC (Free) time (sec)</td>
<td>0.10</td>
<td>0.10</td>
<td>0.70</td>
</tr>
<tr>
<td>Total Data (MB)</td>
<td>155M</td>
<td>49.0M</td>
<td>159M</td>
</tr>
<tr>
<td>Overlapped data requests</td>
<td>93.0K</td>
<td>42.7K</td>
<td>93.0K</td>
</tr>
<tr>
<td>GC data</td>
<td>0</td>
<td>0</td>
<td>0.24M</td>
</tr>
<tr>
<td>Suspend wait (sec)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Suspend data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Addr Update Msg</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Addr Update Data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.18  Detailed statistics for BH on 32 processors.
<table>
<thead>
<tr>
<th></th>
<th>Manual</th>
<th>Mark-&amp;-Sweep</th>
<th>Copying</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tmk</td>
<td>DOSA</td>
<td>Tmk</td>
</tr>
<tr>
<td>Time (sec)</td>
<td>83.9</td>
<td>82.3</td>
<td>84.0</td>
</tr>
<tr>
<td>GC (Free) time (sec)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.21</td>
</tr>
<tr>
<td>Total Data (MB)</td>
<td>46.7</td>
<td>38.2</td>
<td>46.7</td>
</tr>
<tr>
<td>Overlapped data requests</td>
<td>66.0</td>
<td>44.4</td>
<td>66.0</td>
</tr>
<tr>
<td>GC data</td>
<td>0</td>
<td>0</td>
<td>0.36</td>
</tr>
<tr>
<td>Suspend wait (sec)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Suspend data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Addr Update Msg</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Addr Update Data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.19 Detailed statistics for MIP on 8 processors.

<table>
<thead>
<tr>
<th></th>
<th>Manual</th>
<th>Mark-&amp;-Sweep</th>
<th>Copying</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tmk</td>
<td>DOSA</td>
<td>Tmk</td>
</tr>
<tr>
<td>Time (sec)</td>
<td>56.8</td>
<td>53.3</td>
<td>57.1</td>
</tr>
<tr>
<td>GC (Free) time (sec)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.21</td>
</tr>
<tr>
<td>Total Data (MB)</td>
<td>91.1</td>
<td>79.7</td>
<td>91.1</td>
</tr>
<tr>
<td>Overlapped data requests</td>
<td>115</td>
<td>76.7</td>
<td>115</td>
</tr>
<tr>
<td>GC data</td>
<td>0</td>
<td>0</td>
<td>0.62</td>
</tr>
<tr>
<td>Suspend wait (sec)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Suspend data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Addr Update Msg</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Addr Update Data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.20 Detailed statistics for MIP on 16 processors.

<table>
<thead>
<tr>
<th></th>
<th>Manual</th>
<th>Mark-&amp;-Sweep</th>
<th>Copying</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tmk</td>
<td>DOSA</td>
<td>Tmk</td>
</tr>
<tr>
<td>Time (sec)</td>
<td>50.1</td>
<td>45.5</td>
<td>50.3</td>
</tr>
<tr>
<td>GC (Free) time (sec)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.21</td>
</tr>
<tr>
<td>Total Data (MB)</td>
<td>131M</td>
<td>113M</td>
<td>132M</td>
</tr>
<tr>
<td>Overlapped data requests</td>
<td>152K</td>
<td>99.5K</td>
<td>152K</td>
</tr>
<tr>
<td>GC data</td>
<td>0</td>
<td>0</td>
<td>0.77M</td>
</tr>
<tr>
<td>Suspend wait (sec)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Suspend data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Addr Update Msg</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Addr Update Data (MB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.21 Detailed statistics for MIP on 32 processors.
help, the global synchronous collections would have been necessary. We performed this manual help because we want to isolate the intra-processor collection costs.

On DOSA, the mark-sweep and the copying collectors have about the same performance. They outperform their TreadMarks counterparts. They outperform the mark-sweep collector on TreadMarks by 9.8%, and outperform the copying collector on TreadMarks by 41%.

The copying collector on TreadMarks has the worst performance among all cases. The suspension cost for TreadMarks' copying collector is small in MIP because the program only performs a few collections during the entire program execution. For the same reason as in Game, there is no data exchange during the suspensions.

The main cause for the copying collector's underperformance is the bogus writes. From the table we can see that with the copying collector on TreadMarks the total data communicated is 74% more than that on DOSA. It is also 52% more than that of the mark-sweep collector on TreadMarks. The address updates also have non-trivial cost, sending 51.2K messages. This is because MIP only performs two garbage collections and the objects have not advanced to the older generation when the program ends. Therefore all live objects move during the garbage collections.

The delayed reclamation of the fromspace on TreadMarks also results in higher memory usage. On DOSA, the copying collector requires 16M bytes of memory. On TreadMarks, it requires 24M bytes of memory.

Summary

Our evaluation shows that the shared object space abstraction presents a better interface between the DSM system and the intra-processor garbage collector. It makes the implementation of the intra-processor garbage collector orthogonal to the DSM system. This is particularly beneficial to the copying algorithms. It not only results in simpler implementation of the copying collector, it also allows better program performance and less memory usage. In our test programs using the copying garbage collector, the shared object space abstraction improves the program performance by 3% to 41%, and reduces the memory usage by up to 33%.

6.3.2 Discussion on Fine-grain DSM Systems

The results of TreadMarks should apply to other coarse-grain DSM systems as well. Here we make an educated guess on how the fine-grain systems will be affected.
The poor spatial locality as a side effect of the mark-sweep collectors only increases the communication cost in coarse-grain systems. The fine-grain systems are immune to this problem since they do not perform page-based aggregation. However, as our measurement has shown, the performance of the mark-sweep collectors may be inferior to that of the copying collectors. If the fine-grain systems limit themselves to mark-sweep collectors, they are putting themselves at a disadvantage.

In the case of the copying garbage collector, our measurement has shown that the major costs in the copying collector on TreadMarks is the bogus writes and the address updates. Both costs are not limited to TreadMarks, they exist in all conventional DSM systems, including the fine-grain DSM systems. Therefore, we believe that a copying collector on conventional fine-grain DSM systems will have lower performance than on DOSA.

6.4 Conclusions

In this chapter we have evaluated our solutions to garbage collection on DSM systems. We conducted two experiments. The first experiment evaluates the impact of our adaptive collector on overall program performance. The second experiment evaluates the effect of the shared object space abstraction on intra-processor garbage collections performed by each processor.

In the first experiment, we have implemented our adaptive collector on DOSA. We compared its performance with two existing DSM garbage collectors. One of them is based on the weighted reference counting method, the other on the stub-scion method.

This experiment substantiates the following claims:

1. Our collector eliminates the adverse side effects on program performance: it avoids the problems of delayed reclamation and the premature flushing. It improves the overall program performance by up to 50%. Furthermore, with our new collector, the program performance is within 5% of that of explicit memory management by the programmer.

2. Our collector itself is efficient when compared with previous collectors: the extra computation cost to build and trace the object graph amounts to less than 7% of the garbage collection cost, and has negligible effect on the overall program performance; and the amount of GC data is often less than that in the other collectors.
In the second experiment, we implemented two intra-processor garbage collectors that are representative of those in common use on both TreadMarks and DOSA. One of the garbage collectors is based on mark-sweep, the other is based on generational copying. The adaptive algorithm is used for the inter-processor collections. This experiment demonstrates that the shared object space abstraction in DOSA allows better (up to 41%) overall program performance than conventional DSM systems.
Chapter 7

Conclusions and Future Work

7.1 Conclusions

The goal of this thesis was to provide efficient support for distributed sharing of data using typed programming languages, such as Java and Modula-3. We have achieved this goal. In this thesis we provided efficient support for transparent sharing of data in distributed environments using DSM systems. We demonstrated that the typed programming languages provide new opportunities to improve the system performance, and we effectively dealt with the implementation challenge posed by garbage collection. The following paragraphs summarize our main results.

We have designed a new DSM system, DOSA, that takes advantage of the typed languages to efficiently support both coarse-grained and fine-grained sharing patterns, and to support efficient garbage collection. DOSA requires that the programming language provide sufficient information at run-time so that it allows an unambiguous determination of whether a location contains an object reference or not. In addition, in the case of a reference, the type and size of its referent must be known at run-time. DOSA also assumes that the programs written in such languages behave safely, e.g., an object access cannot go beyond the end of the object. Many modern languages provide such support, including Java and Modula-3. We compared the performance of DOSA with that of TreadMarks, a DSM system that is efficient at handling coarse-grained sharing. Our evaluation shows that: the performance of coarse-grained programs on DOSA is comparable (within 6%) with TreadMarks, and the performance of fine-grained programs is significantly better (up to 98%) than TreadMarks.

We have designed and implemented an adaptive algorithm for inter-processor garbage collection. Our algorithm identifies garbage earlier and more accurately than the previous algorithms with commonly used parallel data structures. Our algorithm improved the overall program performance by up to 50% than previous algorithms in the test programs. We also used the abstraction of a shared object space to make the implementation of the intra-processor collector orthogonal to the DSM operations,
eliminating the negative impact of the intra-processor collector on the DSM performance. Our evaluation showed that the shared object space abstraction improves the overall performance by up to 41% over the previous DSM collectors. Combined, the two solutions above allowed the overall program performance of the garbage-collected programs to approach (within 5%) that of manual memory management by the programmer.

Overall, our results indicate that the transparent sharing of data in distributed environments can be efficiently supported for typed programming languages.

7.2 Future Work

A key question in determining both the long-term relevance of this work and the directions of the future work is what types of parallel applications and programming environments are likely to be predominant in the future. Most of the existing work, including this one, have been implemented in homogeneous environments, and evaluated with scientific programs which have static access patterns and complete between a few seconds and a few minutes. As parallel systems enter mainstream business, we expect them to face heterogeneous environments and long running applications with dynamic sharing patterns. Therefore we expect the following two directions for future research to be productive: the heterogeneous DSM systems and the support of dynamic sharing patterns.

7.2.1 Heterogeneous DSM System

Heterogeneity exists in many computing environments. It is therefore highly desirable to integrate heterogeneous machines into a coherent distributed system and to share resources among them. A heterogeneous DSM system will provide an ideal programming platform in such environments.

Several heterogeneous DSM systems have been designed in the past [ZSLW92, BF88], with Mermaid [ZSLW92] as a notable example. However, such systems have not come to common use, mainly because they still do not support fully transparent distributed data sharing. Take Mermaid for example, the system is built around the support of the C programming language. Therefore the hardware difference is still exposed to the programmer. For each basic data item, the programmer sees the difference in its sizes on different machines. For example, aggregate structures are padded so that they are of the same size on all machines, but the padding and the
possible difference in the ordering of fields are visible to the programmer. Although Mermaid can automatically perform data format conversion between machines, the conversion requires non-standard compiler support, limiting its appeal to a general audience.

The combination of Java and a DSM system would provide a better programming platform for heterogeneous environments. The strength of Java is in its inherent portability and its strong type system. There are no implementation-dependent aspects of the Java language specification. Therefore an object will present the same view to the programmer regardless of the machine architecture. The strong type system allows automatic data format conversion under the standard compilers. With the combination of Java and a DSM system, the DSM system would provide transparent data sharing, while Java would completely hide the hardware difference between the processors from the programmer. We have already demonstrated [YC97] that the Java/DSM combination is much easier to use than the Remote Method Invocation (RMI) mechanism. What remains to be done is the issue of performance.

DOSA, which already provides a programming model very similar to Java’s and can use the type information to efficiently support different sharing granularities, is a good candidate to provide the DSM support for Java.

### 7.2.2 Supporting Dynamic Sharing Patterns

Currently, most of the research on parallel systems has focused on the scientific applications that have static sharing patterns. As parallel systems enter mainstream business, they are likely to face other types of applications. Many commercial applications, for example, server applications, run for a long time and must respond to outside requests. Therefore, they are more likely to exhibit different sharing patterns in different phases of the program execution.

Since DOSA allows each processor in the system to freely reorganize its memory without any negative side effect on the system performance, it is possible for DOSA to reorganize the memory layout on each processor in response to the changes in the data sharing pattern. A reorganization would move heavily used objects together, improving the spatial locality and the program performance.
Bibliography


