

INFORMATION TO USERS

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

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

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

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

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

UMI

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

RICE UNIVERSITY

**Automatic Data Aggregation for Software
Distributed Shared Memory Systems**

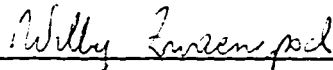
by

Karthick Rajamani

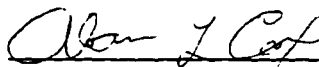
A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

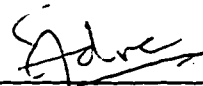
APPROVED, THESIS COMMITTEE:



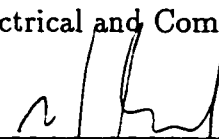
Dr. Willy Zwaenepoel, Chairman
Professor
Electrical and Computer Engineering



Dr. Alan Cox
Assistant Professor
Computer Science



Dr. Sarita Adve
Assistant Professor
Electrical and Computer Engineering



Dr. Peter Druschel
Assistant Professor
Computer Science

Houston, Texas

February, 1997

UMI Number: 1384403

UMI Microform 1384403
Copyright 1997, by UMI Company. All rights reserved.

**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

ABSTRACT

Automatic Data Aggregation for Software Distributed Shared Memory Systems

by

Karthick Rajamani

Software Distributed Shared Memory (DSM) provides a shared-memory abstraction on distributed memory hardware, making a parallel programmer's task easier. Unfortunately, software DSM is less efficient than the direct use of the underlying message-passing hardware. The chief reason for this is that hand-coded and compiler-generated message-passing programs typically achieve better data aggregation in their messages than programs using software DSM. Software DSM has poorer data aggregation because the system lacks the knowledge of the application's behavior that a programmer or compiler analysis can provide.

We propose four new techniques to perform automatic data aggregation in software DSM. Our techniques use run-time analysis of past data-fetch accesses made by a processor, to aggregate data movement for future accesses. They do not need any additional compiler support.

We implemented our techniques in the TreadMarks software DSM system. We used a test suite of four applications - 3D-FFT, Barnes-Hut, llink and Shallow. For these applications we obtained 40% to 66% reduction in message counts which resulted in 6% to 19% improvement in execution times.

ACKNOWLEDGEMENTS

I express my deep gratitude to Dr. Alan Cox for his continuous guidance and advise without which I could not have completed this thesis. I thank the other members of my committee Dr. Willy Zwaenepoel, Dr. Sarita Adve and Dr. Peter Druschel for their invaluable comments and suggestions. I also thank my friends Ramakrishnan Rajamony and Parthasarathy Ranganathan for their continuous help throughout the course of my work.

Contents

Abstract	ii
Acknowledgments	iii
List of Tables	vi
List of Illustrations	vii
1 Introduction	1
1.1 Automatic Aggregation based on run-time history of access patterns .	2
1.1.1 Case Study - Large Coherence Units	2
1.2 The Need for Dynamic Analysis	3
1.2.1 Solution to the Communication Overhead Problem	5
1.3 Techniques for Automatic Aggregation	6
1.4 Results	6
1.5 Overview	7
2 Techniques for Automatic Aggregation	8
2.1 Classification of Automatic Aggregation Techniques	8
2.2 TreadMarks	10
2.3 Aggregation Techniques	12
2.3.1 Creation and Usage of Page-groups	12
2.3.2 Algorithm	13
2.3.3 Dynamic Resizing of Pages with Code-Memory locality	14
2.3.4 Dynamic Resizing of Pages with No-Code-Memory locality . .	15
2.3.5 History Dependent Prefetching with Code-Memory locality . .	15
2.3.6 History Dependent Prefetching with No-Code-Memory locality	15
2.3.7 Dynamic Resizing of Pages under the buddy-system	16
2.4 Implementation issues for the Aggregation Techniques	20
2.4.1 Granularity of <i>diffs</i> for DRP schemes	20
2.4.2 Window of Combining	20
2.4.3 Optimistic Schemes	21

3	Experiments and Results	22
3.1	Experiments	22
3.1.1	Performance evaluation	22
3.1.2	Overhead measurement	23
3.1.3	Analysis of thresholds in DRP-buddy-system	23
3.2	Applications	23
3.2.1	3D-FFT	23
3.2.2	Barnes-Hut	24
3.2.3	Ilink	25
3.2.4	Shallow	25
3.3	Performance Evaluation	26
3.3.1	3D-FFT	26
3.3.2	Barnes-Hut	29
3.3.3	Ilink	32
3.3.4	Shallow	35
3.4	Overhead Measurement	38
3.4.1	3D-FFT	38
3.4.2	Barnes-Hut	39
3.4.3	Ilink	39
3.4.4	Shallow	40
3.5	Analysis of thresholds in DRP-buddy-system	40
3.6	Summary	42
4	Related Work	43
4.1	Messaging Overheads	44
4.2	Variation in Performance with Size of Coherence Unit	44
4.3	Dynamic Resizing of Coherence Unit	45
4.4	Adapting based on data-sharing patterns	46
5	Conclusions	48
5.1	Goal	48
5.2	Conclusions	49
5.3	Future Work	50
	Bibliography	51

Tables

1.1	4 processor Execution Time for Quicksort, 1M elements	5
3.1	Inputs for the Test Suite	26
3.2	Overhead Statistics for 3D-FFT, 64x64x64, 100 iterations	39
3.3	Overhead Statistics for Barnes-Hut, 16384 bodies, 20 iterations	39
3.4	Overhead Statistics for Ilink, CLP data set	40
3.5	Overhead Statistics for Shallow, 50 iterations	41

Illustrations

1.1	Variation of Execution Time with page size	4
2.1	History-collection-and-fetch phase	14
2.2	DRP_B Algorithm - Terminology	17
2.3	DRP_B Algorithm for Grouping	19
3.1	Execution time for 3D-FFT, normalized w.r.t Tmk-4K	27
3.2	Total Message Count for 3D-FFT, normalized to Tmk-4K	27
3.3	Data Traffic for 3D-FFT, normalized w.r.t Tmk-4K	28
3.4	Execution time for Barnes-Hut, normalized w.r.t Tmk-4K	30
3.5	Total Message Count for Barnes-Hut, normalized to Tmk-4K	30
3.6	Data Traffic for Barnes-Hut, normalized w.r.t Tmk-4K	31
3.7	Execution time for Ilink, normalized w.r.t Tmk-4K	33
3.8	Total Message Count for Ilink, normalized to Tmk-4K	33
3.9	Data Traffic for Ilink, normalized w.r.t Tmk-4K	34
3.10	Execution time for Shallow, normalized w.r.t Tmk-4K	36
3.11	Total Message Count for Shallow, normalized to Tmk-4K	36
3.12	Data Traffic for Shallow, normalized w.r.t Tmk-4K	37

Chapter 1

Introduction

In recent years, networked workstation have gained importance as a cost-effective platform for parallel computing. They are more affordable than dedicated parallel machines or supercomputers. With the advent of high-speed general-purpose networks and extremely versatile processors, networked workstations have increasingly become more powerful. Soon, they could prove a viable alternative for supercomputers.

In the absence of specialized hardware for providing shared memory, such clusters were running parallel programs chiefly through message-passing libraries such as PVM [8], TCGMSG [9] and Express [16]. But writing parallel programs with a message-passing model is much less intuitive than with a shared-memory model. With the message-passing model the programmer has to decide on when to initiate communication, between whom to perform the communication and what needs to be communicated. This burden on the programmer with message passing has spurred the interest in software Distributed Shared Memory (DSM) systems like TreadMarks [12] [11], which provide a shared memory abstraction over a distributed memory hardware.

However, software DSM systems generally incur greater communication overheads than message-passing systems. As the DSM system operates with little or no knowledge of the application, it is not able to optimize communication. In contrast, a programmer in a message-passing environment has complete knowledge of the data access patterns (which is required for correctness) and so can effectively optimize communication.

The larger communication overhead for DSM systems can be reduced if the number of messages exchanged for maintaining coherency between the workstations is reduced. This can be achieved through the aggregation of data into fewer messages. Reducing the message count results in reduced number of processor interrupts and

reduced number of calls that invoke the network support system.

In this thesis, we present some techniques for doing automatic aggregation of data that is transferred between processors. Our aggregation techniques are based on the run-time analysis of data-fetch patterns. Based on the patterns observed in the past, future access patterns are predicted. The system uses these predictions to aggregate data transfers in order to reduce message overheads.

1.1 Automatic Aggregation based on run-time history of access patterns

DSM systems encounter a large variety of programs with quite different behaviors. This prevents a generalized static approach to aggregation of data that can be applied to all programs. Program-specific approaches to data aggregation, using compiler analysis or the programmer's knowledge of the application, may also not prove very useful when the data movements depend on the program inputs, or are of a dynamic nature. Thus we require a flexible approach to aggregation that can adapt to the application's behavior.

In DSM systems messages are exchanged between processors for maintaining a coherent picture of the different copies of shared memory. For better efficiency and ease of implementation, there is a minimum size or granularity at which the memory units are made coherent with one another. Memory units of this size are referred to as **coherence units**. When a processor requires the current data for a location in the shared space from remote memory, the current data for the entire coherence unit containing that location is obtained. Thus, a simple approach to reduce message counts, that does not require run-time analysis, is to have a larger coherence unit. We present the large coherence unit method as a case study for showing the difficulties in adopting a static approach to reducing message traffic.

1.1.1 Case Study - Large Coherence Units

The size of the coherence unit is a critical factor in the performance of a DSM system. A direct effect would be on the number of messages sent. As the data for the shared

regions is obtained through message exchanges that take place at one exchange per coherence unit, a larger coherence unit could potentially cut down the number of messages sent for maintaining consistency. Also, larger coherence units could provide the effect of prefetching, bringing in the data for shared locations before they are required, if the locations happen to lie within the same coherence unit.

However, a potential problem with large coherence units is false-sharing. Locations sharing the same coherence unit are invalidated together and the data for them obtained together. Even if only part of the coherence unit is truly shared, the program would incur the penalties for making the rest of the coherence unit consistent. The penalties would be the extra bandwidth consumed by the data movement for the falsely-shared portions in the coherence unit, and the extra faults encountered by the program when the invalidated falsely-shared portions are accessed. In systems that use sequential consistency, there is the additional overhead of extra invalidation messages (with relaxed consistency models, invalidations are sent together at synchronization accesses and the problem of invalidations that ping-pong is vastly reduced).

Different applications require coherence units of different sizes for optimal performance. Typically, the requirement is governed by the shared data structures, their location in the shared memory and the pattern of accesses made to them. Figure 1.1 shows the difference in performance with the size of coherence unit for four applications run on the TreadMarks DSM system. TreadMarks has the virtual memory page as the coherence unit. The programs were run on 8 SparcStation20s connected by a 155Mbps ATM network. The figure shows a clear difference in performance with respect to page size for most of the applications. It also shows a lot of variation in the preferred page size for an application. The figure illustrates how crucial the correct coherence-unit size is for the performance of a program, and how dependent the optimal value for the size of the coherence unit could be on the actual application.

1.2 The Need for Dynamic Analysis

In some cases it could be possible to identify the optimal size for the coherence units with some static analysis but that would imply extra effort on part of the programmer or the availability of a superior compiler. Some of the factors favoring a particular

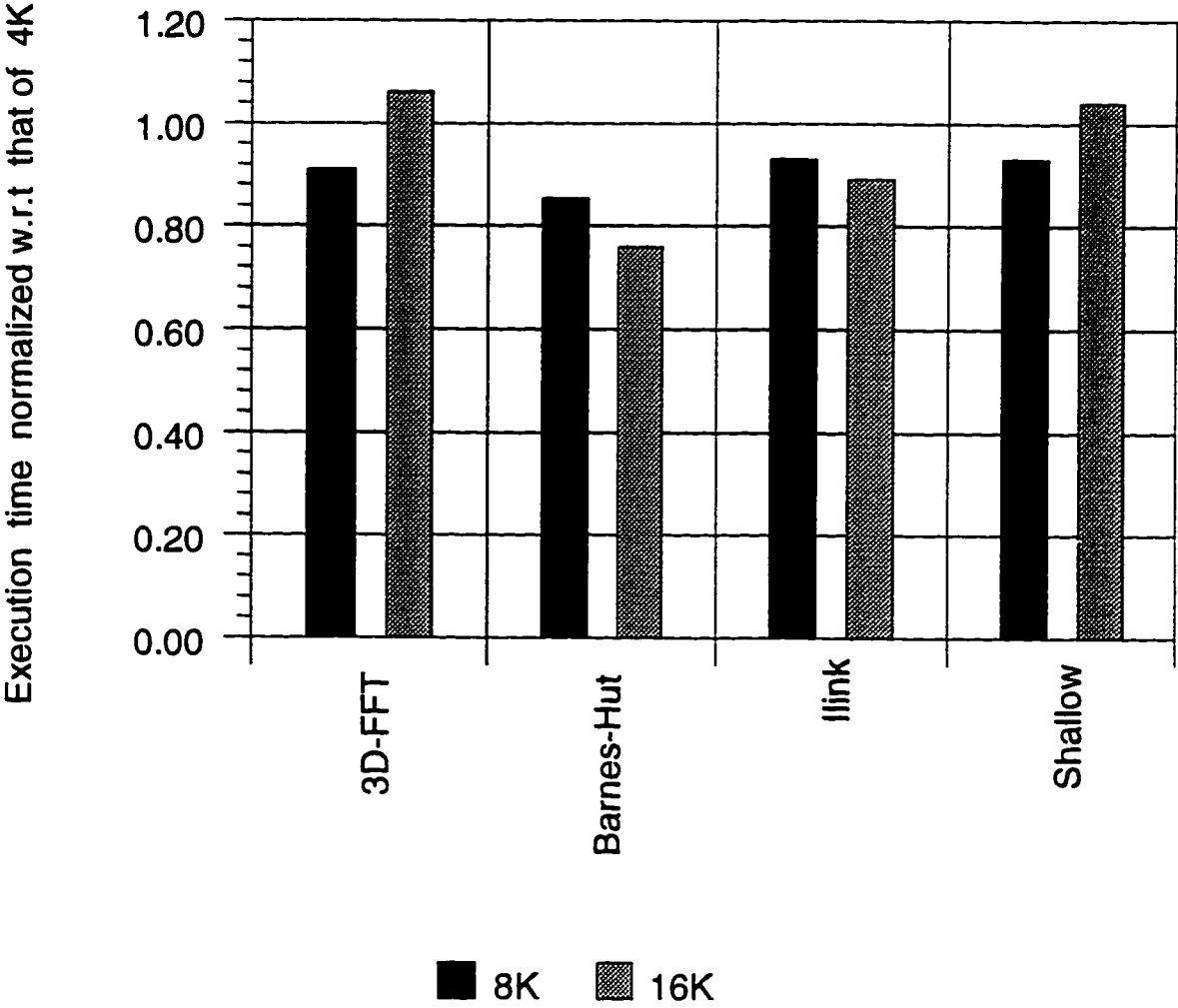


Figure 1.1 Variation of Execution Time with page size

value for the size of the coherence unit are also affected by the input to the application. As an example of this the four processor execution time for the Quicksort algorithm on the TreadMarks system is given in Figure 1.1. It can be seen that for the input parameter Bubble Threshold, a value of 4096 shows the 8K page giving better performance, while a value of 1024 shows the 4K page giving better performance. Hence, the optimal size for the coherence unit is potentially difficult to ascertain statically and sometimes variable for even a particular application.

It is also possible that the optimal page size changes over the execution of an application. It could also vary with different regions in the shared address space. In such cases, a statically determined coherence-unit-size that is uniform over the address space cannot not be the optimal solution to obtain the best performance from the DSM system.

1.2.1 Solution to the Communication Overhead Problem

In order to do the data aggregation without increasing the burden on the programmer and without the need for specialized compilers, we incorporated automatic aggregation techniques into the run-time system implementing the DSM. Because of the large variety of applications run on DSM systems many of whose data-sharing patterns may not be easily predictable, we decided to implement the aggregation based on facts obtained by studying the actual execution of the application. We believe that a large class of applications exhibit a regular data access pattern for a significant part of the execution. In such a case, the knowledge of the access patterns at an earlier phase of the execution could be used to correctly predict the accesses at later phases, which would aid in reducing the data-transfer messages for the later phases.

Bubble threshold	1024	1024	4096	4096
Page size	4K	8K	4K	8K
Exec. time (sec)	15.29	16.40	8.37	8.07

Table 1.1 4 processor Execution Time for Quicksort, 1M elements

In the next section we introduce our techniques for doing automatic aggregation.

1.3 Techniques for Automatic Aggregation

The size of the data transfer unit has a significant effect on the overheads due to data transfers in a software DSM. To optimize these transfers we designed techniques that change the size of the transfer unit to adapt to the run-time requirements. These techniques are called the Dynamic Resizing of Pages (DRP) techniques. We refer to the coherence unit as seen by the data-fetch operations, as a *fetch-unit*. With DRP, the fetch-unit for a region of shared-memory is changed from a single page to a group of contiguous pages. By grouping the contiguous pages in different ways for that region of memory, different size requirements for the fetch-unit can be met. The grouping of pages is changed depending on the requirements of the program execution, which is indicated by the data-fetch patterns of previous phases of the execution. The other set of techniques, History Dependent Prefetching (HDP), while sharing the principle of using past data-fetch patterns to create page groups that will be used as fetch-units, do not have the restriction of requiring only contiguous pages to form a group.

Aggregation is achieved by fetching data for all the pages within a group by combining all the individual messages for each page within the group. Our techniques are further divided on the basis of the scope of the page-groups, which can be global (NCM) or local (CM). Thus we have four techniques resulting from the different groupings and their scopes - DRP_NCM, DRP_CM, HDP_NCM and HDP_CM.

1.4 Results

The test suite for our techniques consisted of four applications - 3Dfft, Barnes-Hut, Ilink and Shallow. In the presence of some predictable access pattern in the application all our techniques showed a significant performance enhancement over the regular TreadMarks implementation. This was the case with 3D-FFT, Barnes-Hut and Ilink. Our techniques obtained a reduction in execution time from 6% to 19% depending on the implementation and the application, and provided a reduction in message count that ranged from 40% to 66%. Even in the case of Shallow where the potential

for combining requests was much smaller, we obtained a 6% reduction in execution time with a 40% reduction in message count with some of our techniques. For the set of applications we considered, we found that HDP_NCM, which was the least restrictive in its criteria for group formation, performed the best for all the applications.

1.5 Overview

The rest of the thesis is organized in the following manner. Chapter 2 discusses the aggregation techniques in detail, going into the algorithmic details and their implementation. In chapter 3 we discuss the set of experiments performed for analyzing the techniques, present the four applications in our test suite and discuss the results of our experiments in detail. In chapter 4 we present the related work in this area. In chapter 5 we present the conclusions and future directions for our continuing research.

Chapter 2

Techniques for Automatic Aggregation

We present our techniques for automatic aggregation in this chapter. There were two orthogonal parameters that affected the behavior and performance of our implementations. In section 2.1, we discuss these two parameters and classify each of our techniques according to these parameters. In section 2.2 we present a brief overview of the underlying TreadMarks DSM system. In section 2.3 we give a detailed presentation of our four techniques for automatic aggregation with a description of their algorithms and implementations. We also discuss the implementation of a contemporary technique, proposed for cache-coherent multi-processing environments, in the context of a software DSM system. We present some of the issues involved in the implementation of the aggregation techniques in section 2.4 of this chapter.

2.1 Classification of Automatic Aggregation Techniques

The run-time analysis incorporated in the DSM system identifies the set of pages requests for whose data can be sent together. The DSM system groups pages in each set into *page-groups*. When sending the request for the data of a page, the page-group that the page belongs to (if any) is identified, and combined requests are sent for all the pages in that group for which the most recent data is not available. We classify our techniques for automatic aggregation according to two parameters – a) the domain for page-groups, and b) the scope of page-groups. Each of our four techniques is a distinct implementation arising out of the different combinations of the choices for these parameters.

While considering the domain for forming page-groups, we explored two options – a) consider only adjacent pages for forming page-groups, or b) have any set of pages form a group. The first option is a natural extension of the idea of large coherence units. Since contemporary research [6] [19] [15] has found an application-dependent

performance variation with the size of coherence units we explored the benefit of dynamically aggregating requests for adjacent pages, under the first option. As a group formed from adjacent pages grows or shrinks in size, depending on program requirements, the contiguous region of memory mapped to the pages in the group correspondingly grows or shrinks. We termed the techniques for managing groups of adjacent pages as techniques for Dynamic Resizing of Pages (DRP).

The second option eliminates the restriction of grouping only adjacent pages to create page-groups. If the program's data-fetch patterns indicate a preference for larger page sizes (i.e they favor larger groups of adjacent pages), we felt this option could subsume the benefits from grouping together just adjacent pages (as it does not prevent adjacent pages from being part of the same group). As these techniques use past accesses as the only criterion for combining requests for different pages (forming page-groups) we refer to them as History Dependent Prefetching (HDP) techniques.

The second parameter that we use to classify our techniques is the scope of page-groups. Pages that require remote-data fetch, due to accesses within the same critical section, are grouped together. Once a group is thus formed, its scope can be extended to all critical sections or for just the one in which it was created. The first option yields the global scope of page-groups and the second yields the local scope of page-groups. The techniques where the scope is restricted to being local, imply a affinity between the instructions within the concerned critical section and the memory accessed by them. We refer to this affinity as Code-Memory locality (CM). The techniques that attribute a local scope for page-groups operate under the assumption of CM locality, while those that attribute a global scope to page-groups assume a form of spatial locality, that we term simply as NCM (non-CM).

To better illustrate this point, consider the execution of a critical section S_0 , in which accesses to both pages A and B required remote data-fetch operations resulting in both A and B being put together in a page-group PG. Then under the local-scope for page-groups, in a later execution of the same critical section (same piece of code) when either of the pages require remote data-fetch, the most recent data for the other page would also be obtained (implying an affinity between the memory region consisting of pages A and B and the code executed in section S_0). But, if during a later execution of another critical section S_1 , data was required for either A or B, then

data for the other page need not be obtained. Whereas, under the global scope of page-groups, PG would be a valid page-group even for S_1 , and so when data is fetched for either page A or B during S_1 's execution, data for the other page would also be obtained (if it was invalid).

Thus based on these two page-group parameters we obtain four techniques :

- Dynamic Resizing of Pages without CM - DRP_NCM.
- Dynamic Resizing of Pages with CM - DRP_CM.
- History Dependent Prefetching without CM - HDP_NCM.
- History Dependent Prefetching with CM - HDP_CM.

2.2 TreadMarks

We implement all our techniques as part of the TreadMarks distributed shared memory (DSM) system. In this section we discuss some aspects of the TreadMarks system that are required to understand the implementation of our techniques.

TreadMarks [11] is a software DSM system that provides a shared-memory abstraction on a workstation cluster connected by any general-purpose network. TreadMarks is based on a relaxed consistency model, *Lazy Release Consistency* (LRC) [10] [12], and implements a *multiple-writer* protocol. With a multiple-writer protocol the distinct locations in the same coherence unit can be simultaneously written to by more than one processor. The system uses the virtual memory page as the default coherence unit. It uses the virtual memory support provided by the hardware and operating system for *write-detection*.

The TreadMarks system has an invalidation-based coherence protocol. Processors that write to a page send invalidations to other copies of that page at appropriate times. On receiving the invalidations, the other processors prevent accesses to those copies by using the Unix `mprotect` system call. When the program on another processor accesses one of those copies, it faults. The fault-handler then brings in the

data for the accessed page, updates it, unprotects it and then permits the continuation of the program. The invalidations are referred to as *write-notices* in TreadMarks terminology.

TreadMarks has two types of synchronization objects - barriers and locks. The synchronization operations are `Tmk_lock_acquire`, `Tmk_lock_release` and `Tmk_barrier`. All three help in making the entire shared memory on a processor consistent. In the LRC model, the basic synchronization primitives are *releases* and *acquires*. `Tmk_lock_release` is modeled as a *release*, `Tmk_lock_acquire` is modeled as an *acquire* and `Tmk_barrier` is modeled as a *release* followed by an *acquire*. A *release* and the corresponding *acquire* provide synchronization between the processors making either call. `Tmk_barrier` synchronizes all the processors used by an application.

As TreadMarks uses a multiple-writer protocol, there could be many writers for a single page at any one time. In order to make a page consistent, a processor has to fetch all the previous modifications to the page made by the other processors and apply them in the correct order. The modifications to page that a processor makes are stored in a special structure known as a *diff*. For making a page consistent, a processor has to fetch all the *diffs* for that page created by the other processors and apply them in the correct order. In order to provide this order, TreadMarks divides the execution of each process into *intervals*. A new interval is begun on a processor every time an *acquire* or *release* is executed by it. Intervals on the same processor are totally ordered by program order, while those on different processors are *partially* ordered. An interval on processor *p* precedes an interval on processor *q* if the interval on *q* began with the *acquire* corresponding to the *release* that concluded the interval on processor *p*. Release consistency requires that before a processor *p* continues past an *acquire* the modifications made in all the intervals that precede it (including those of the other processors) must be visible at *p*.

With Lazy Release Consistency, the propagation of the modifications is postponed till the time of an *acquire*. It is only at an *acquire*, that the processor performing the *acquire* obtains the *write_notices* from the releasing processor. On a subsequent access to a page for which the processor has obtained a *write_notice*, the processor fetches the *diffs* for that page from the processors that wrote to it earlier, makes the page consistent by applying those *diffs* to that page, and then proceeds with the

program execution.

2.3 Aggregation Techniques

In this section, we present four new techniques for performing data aggregation. Each of these techniques is a unique combination of the domain for page-groups and the scope of page-groups. For comparison, we also present the implementation of a fifth technique that is the adaptation of a technique proposed for a cache-coherent multi-processing environment. We first present the algorithmic framework for all of our techniques and then describe each of the individual techniques.

2.3.1 Creation and Usage of Page-groups

For all our techniques, page-grouping is done on the basis of information recorded about the accesses that required remote-data fetch in previous phases of execution. The system preserves the history of accesses in the form of a list of those pages which require remote data-fetch during the current execution *interval*. This list is referred to as a *fetch-list*.

During the execution of every interval, a fetch-list is built up. Under the TreadMarks system, all invalidations take place only at acquires. Hence, the information (list) collected since the last acquire would not be useful until the pages in that list get invalidated again, which cannot happen until the next acquire. Hence, we initiate the creation of new lists at every acquire. Thus, the interval for which a list of pages requiring remote data-fetch is created, spans the code between one acquire to another. This interval bounded by two acquires is termed as an *acquire-to-acquire interval*.

During the execution of an acquire-to-acquire interval, a corresponding fetch-list is created. For the pages in the fetch-list, data was obtained from remote memory when the code in the acquire-to-acquire interval was previously executed. This is the history of data-fetch accesses that the system uses to create, modify and destroy page-groups. Every time an acquire-to-acquire interval is executed a new fetch-list is created.

Under the adjacent-page grouping scheme (DRP) any set of pages that form a contiguous region in the shared memory and occur in the fetch-list created for an acquire-to-acquire interval would be considered as a page-group. Under the no-restriction grouping scheme, HDP, all the pages that occur in the fetch-list created for an acquire-to-acquire interval form a single page-group. Under both the schemes, requests for the data of all the invalidated pages within a page-group are sent together, combining the requests for multiple-pages whenever possible.

Under the techniques that attribute a global scope (NCM) to page-groups, the pages of a page-group are recognized as a group when remote-data fetch is required for any of those pages anywhere during the execution. For the techniques that attribute only a local scope (CM) to the page-groups, the pages are identified as a group only for the particular acquire-to-acquire interval in the code, from whose past fetch-list the group was created.

All requests for data that might be needed in shared-memory accesses between one acquire to another are sent out in an aggregated manner only after the first acquire. This ensures that the maximum number of invalidations for the concerned group are obtained before combined requests for the invalidated pages in the group are sent out. This increases the potential for combining requests compared to when the requests are sent at any time before the first acquire.

2.3.2 Algorithm

The algorithm for the techniques work in two phases. The first one is the History-collection-and-fetch phase, that begins at an acquire and ends in the next acquire occurring in the execution (during the execution of an acquire-to-acquire interval). During this phase the fetch-list for the acquire-to-acquire interval bounded by the two acquires is built up. The system utilizes the page-group information to send requests and obtain data in an aggregated manner during this phase. This phase is common to all our techniques and is outlined in Figure 2.1.

The second phase is the Group-creation phase. This occurs just after the completion of an acquire. During this phase the fetch-list created during the execution of the

```

for Faulting page P
    Add P to the Fetch-list of acquire-to-acquire interval A
    if P is invalid then
        Determine X, the page-group that P belongs to;
        for every page p in X
            if p is invalid add p to set Y;
        for all the pages in Y
            Send out requests for data, combining them whenever
            possible;

```

Figure 2.1 History-collection-and-fetch phase

acquire-to-acquire interval preceding the acquire is used to modify the set of page-groups in the system. This phase is dependent on the way the individual techniques maintain each of their page-groups and is discussed separately for each technique in the following sections.

2.3.3 Dynamic Resizing of Pages with Code-Memory locality (DRP_CM)

The DRP_CM technique allows only pages that are mapped to contiguous portions of the shared-memory address space to be part of the same page-group. The scope of a page-group is restricted to the acquire-to-acquire interval from whose fetch-list that group was created.

As the groupings are specific to each acquire-to-acquire interval, they are contained in a list specific to each acquire-to-acquire interval. After the execution of the interval, the fetch-list is examined for pages mapped to contiguous memory regions. All such pages are grouped together and the groups are added to the list for that acquire-to-acquire interval. When pages are common between a new group and a group already in the list, the old group is discarded. Thus, the most recent remote-data-fetch information is considered to be representative of future requests for that region of shared-memory.

2.3.4 Dynamic Resizing of Pages with No-Code-Memory locality

Like `DRP_CM`, the dynamic page-resizing technique with no-code-memory locality (`DRP_NCM`) allows only the grouping of pages which are mapped to contiguous portions of the shared-memory address space. However, the scope of the page-groups is not restricted to just that acquire-to-acquire interval from whose fetch-list the groups were created. So, a single list contains all the groupings. The groupings, as in the case of `DRP_CM`, are stored in ordered structures containing the page ids of the first and last pages in the group.

2.3.5 History Dependent Prefetching with Code-Memory locality

Unlike the `DRP` techniques, history dependent prefetching (`HDP`) techniques allow the grouping of any set of pages that are present in the fetch-list of an acquire-to-acquire interval. Thus all the pages in the fetch-list of an acquire-to-acquire interval are considered as a part of a single group. Since the groups are specific to the particular acquire-to-acquire interval with code-memory locality (`CM`), each such interval has its own list of groups. The entire fetch-list for such an interval is added as a new group to the list of groups attached to that interval. Within a list, a page can exist in only one group, while it could be present in different groups in the lists of different acquire-to-acquire intervals. Just as in the case of the `DRP` techniques, the new group is added to the acquire-to-acquire interval's group-list at the execution of the acquire following that interval.

2.3.6 History Dependent Prefetching with No-Code-Memory locality

Under history dependent prefetching with no-code-memory locality (`HDP_NCM`), all the pages in a fetch-list are part of the same page-group. With `NCM`, the scope of the groupings is global. Hence, there is no list of groupings that is specific to any particular acquire-to-acquire interval. Instead, the groupings are kept track of by linking all the pages in a fetch-list to form a doubly-linked list, so that all the members in a group can be identified given any one member of that group. This arrangement also ensures that no page can exist in more than one group, which is necessary under the global scope of page-groups.

2.3.7 Dynamic Resizing of Pages under the buddy-system

Dynamic resizing of pages under the buddy-system (DRP_B) is a version of the DRP_NCM technique. Its implementation for cache-coherent multi-processing systems had been described by Dubnicki and LeBlanc in their work on adjustable block sizes for coherent caches [3]. Their idea for adjustable-sized cache blocks was adapted to work for the shared-memory pages in TreadMarks.

DRP_B follows a buddy-system style of page-groupings. The groups are modified through a pair of operations - *merge* and *split*. A merge operation increases the size of a page-group by combining two adjacent page-groups (pages) of the same size into one larger group. A split operation partitions a larger page-group into two equal-sized smaller page-groups which would be adjacent in the shared-memory space.

DRP_B also has a history-collection-and-fetch phase that functions in exactly the same way as in our techniques. After the fetch-list has been created during an acquire-to-acquire interval the list is utilized in a similar manner as in the case of the other DRP techniques. Adjacent page-groups, all of whose constituent pages required the obtaining of data from across the network in the past interval, are considered as good candidates for merging by the algorithm, as data for both of them could have been obtained together. A page-group, only some of whose constituent pages required data from other processors' memory, is considered a good candidate for splitting, so that unnecessary movement of data for the non-accessed pages of the page-group could be avoided if the access pattern was to repeat itself. The details of DRP_B are outlined in the following sections.

Algorithm for grouping

The algorithm allows the combining of two adjacent virtual memory pages into a *dual* or two adjacent *duals* into a *quad*. Both the dual and quad are also called as *Mega-pages*. Hence every page-group (virtual memory page or mega-page) can be one of three types - single, dual or quad - only; and so every virtual memory page can be in one of three states - single, part of a dual, or part of a quad. The virtual memory pages making up a mega-page, whether a dual or a quad are termed the *Components* of the mega-page. Any merging involves only two equal-sized adjacent

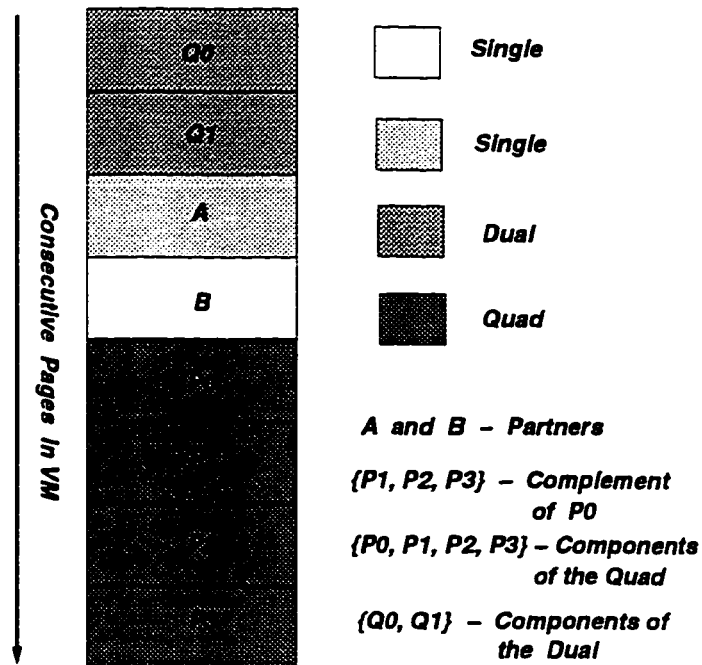


Figure 2.2 DRP_B Algorithm - Terminology

page-groups. A page-group **B** that is being considered for merging with a page-group **A**, is called the *Partner* of **A**. At any one time, a page-group **A** would have only one unique partner **B** i.e if **A** has to merge and become part of a larger mega-page, then it can do so only by combining with **B**. For every page-group **A**, its partner **B** is fixed by the nature of the algorithm. If a page-group **P** is part of a mega-page **M**, the *Complement* of **P** is the partner of **P**, merging **P** with which would yield **M**. Figure 2.2 contains an illustration of these definitions.

Each virtual memory page has some *count* fields associated with it. The value in a count field is a measure of the tendency for the page to move from a certain state to another. Logically there need to be two count fields for every state, one to indicate the tendency to change state to become part a smaller page-group (split) and another to indicate the tendency to change state to become part of a larger page-group (merge). But the states at the extremes, i.e the single and quad need only one count field, a merge-count field for the single state and a split-count field for the quad state.

All the virtual memory pages in a mega-page have the same split-count value and all the virtual memory pages in both the page-groups that are being considered for merging have the same merge-count value. The count values are changed only at grouping time. At that time, the corresponding fetch-list is examined and the count values are modified. The modifications would be an increment of the split-count values for all the pages in the mega-page if only some of the pages in the mega-page were found in the fetch-list, and an increment of the merge-count values of all the pages in the relevant (determined under buddy-system restrictions) page-groups if all the pages in both page-groups were found in the fetch-list. When the split count value exceeds a certain *Split-Threshold* then the decision is made to split the mega-page. Similarly when the merge count value exceeds a certain *Merge-Threshold* then the decision is made to merge the page-group with its Partner.

A pseudo-code representation of the algorithm for grouping pages is presented in Figure 2.3.

Implementation

In the TreadMarks system every processor maintains a directory of the shared pages which has an entry for every page in shared memory. An *accessed* field is added to every entry in the directory structure for the virtual memory pages. This is set if the page happens to be in the *fetch-list*. This field is maintained for making the grouping algorithm more efficient.

Each directory entry has a *state* field that specifies whether a page is single, or part of a dual, or part of a quad. In case the page is part of a dual or quad, the state field also specifies which one of the dual-set (first or second) or quad-set (first, second, third, or fourth) the page is. There are two *count* fields for every directory entry. Only one field is used if the page is single or part of a quad. Both the fields are used if the page is a dual (one for merge-count, the other for split-count).

Our implementation differs from the adjustable cache block idea because of the multiple-writer protocol associated with the TreadMarks DSM system as opposed to single-writer protocols assumed by Dubnicki and LeBlanc. Because of this we need to allow for different groupings of pages for the same region of shared-memory on

```

for every page P in Fetch-list
    if all pages in the Complement of P are also in the list then
        Decrement appropriate split-count values for pages in mega-page S
        containing P;
        if the Components of Partner S' of S are also in Accessed list then
            Increment appropriate merge-count values for S and S';
            if the merge-count values of S and S' are greater than the
            corresponding Merge-Threshold then
                Merge S and S';
                Reset their counters;
    else
        Increment the appropriate split-count values of P and those of the
        pages in its Complement;
        if the split-count value is greater than the corresponding
        Split-Threshold then
            Split S;

```

Figure 2.3 DRP_B Algorithm for Grouping

different processors. Also we provide for the ability to get the data for the different pages of a page-group from different processors. In the hardware implementation, obtaining data from different processors for the different blocks in a cache-line is not required because of the single-writer protocol.

2.4 Implementation issues for the Aggregation Techniques

2.4.1 Granularity of *diffs* for DRP schemes

The *diffs* are maintained at the granularity of a virtual memory page and not at one per page-group, even though such an arrangement could reduce the costs associated in the creation and maintenance of *diffs*. This is essential to allow different groupings of the pages on different nodes while having a simple means of identifying the data to be sent to the requester.

Maintaining *diffs* at one-per-page also helps in reducing false-sharing and in reducing data movement when the page-groups are larger than the trivial case of a single-page page-group. Memory is invalidated at the granularity of the virtual memory pages and so data is sent for just those pages of the page-group that are actually invalid. For example, if at node N_i , A and B are two virtual memory pages making up a page-group D , and A alone was invalid, then B could still be accessed at N_i , without requiring any new data from another node. On the other hand, if B was also invalid, then an access to either A or B would result in the system obtaining the data for both A and B .

In the case of the HDP schemes, since the page-groups can be made of any arbitrary collection of pages, *diffs* can be maintained for no larger a memory region than that of a page.

2.4.2 Window of Combining

Due to system constraints, we were restricted to sending combined requests for at the most four pages (each of size 4K) at a time. In order to reduce the complexity of implementation, for all our techniques we pick the pages whose requests would be

combined by examining a page-group through a window of size four i.e for combining requests we look at four pages from a group at a time. For these four pages all requests that can be combined are sent through a single message, the rest are sent through separate messages. Then the next four pages in the group are handled and so on. Though this reduced the complexity of the implementation it could also reduce the benefit that can be obtained from combining requests.

2.4.3 Optimistic Schemes

With optimistic implementations when the set of page-groups do not change in two successive acquire-to-acquire intervals, the system assumes the same set of page-groups for the rest of the execution. The History-collection part of the first phase of the algorithm and the Group-creation phase are both eliminated for the rest of the execution. These schemes encounter a comparably much smaller (about 40% to 50% reduction) number of faults as compared to the non-optimistic techniques. They could be quite beneficial in a system where the memory faults are costly. On the down-side they can be more inaccurate when compared to the regular implementations. We present the results for the implementation of an optimistic HDP_CM scheme in chapter 3.

Chapter 3

Experiments and Results

In this chapter, we present the set of experiments we performed to evaluate our aggregation techniques. We then present the application suite that we used to test the techniques. Finally, we discuss the results of the experiments for each technique and the conclusions we drew from them.

3.1 Experiments

We outline the set of experiments we performed to evaluate our techniques in this section.

3.1.1 Performance evaluation

Our first set of experiments were common to all the techniques and were concerned with overall performance evaluation. The experiments compared the performance of the regular TreadMarks implementation and the implementations with our aggregation techniques. The metrics for the comparison consisted of three fundamental execution statistics – execution time, data traffic (in terms of the total number of bytes transferred over the network) and the message count (total number of messages exchanged over the network). The last two metrics directly reflect the aggregation of data-transfers obtained with our techniques, and the execution time is a measure of the net benefit the program gets from that aggregation of data transfers.

In addition to the fundamental statistics, we also obtained information like the break-up of data traffic and message count into those for diff movement, page movement and synchronization operations (for barriers and locks). The number of faults as seen by the system and its break-up into different categories were other available statistics.

In the case of the regular TreadMarks implementation, we ran the applications with 3 different page sizes – 4K, 8K and 16K – to provide a static form of aggregation through increase in size of the coherence unit. We compared the dynamic aggregation provided by our techniques to the static aggregation obtained with larger page sizes.

3.1.2 Overhead measurement

Our techniques utilize the memory protection calls to keep track of the page-access patterns. This adds an overhead to the performance of the system. We enumerate the memory-fault counts for all our executions to get a measure of this overhead. We also take up the HDP_CM scheme as a case-study to judge whether we can reduce this overhead by limiting the history-collection phase through *optimistic* schemes.

3.1.3 Analysis of thresholds in DRP-buddy-system

In this experiment we studied the effect of thresholds on the DRP technique. We collected statistics from many executions of DRP_B with different starting sizes and different set of threshold values for each execution and analyzed the relationship between the thresholds, starting size of page-groups and program behavior.

3.2 Applications

We used four applications – 3D-FFT, Barnes-Hut, Ilink and Shallow – to evaluate the performance of all the techniques.

3.2.1 3D-FFT

3D-FFT is a program from the NAS benchmark suite [1]. It numerically solves certain partial differential equations using three-dimensional forward and inverse Fast Fourier Transforms (FFTs). For an input array A of complex numbers organized in row-major order with dimensions $N_1 \times N_2 \times N_3$, 3D-FFT first performs a N_3 -point one-dimensional FFT on each of the $N_1 \times N_2$ vectors. Then it performs a N_2 -point 1-D FFT on each of the $N_1 \times N_3$ vectors. The resulting array is transposed into a $N_2 \times N_3 \times N_1$ array B and a N_1 -point 1-D FFT is applied to each one of the $N_2 \times N_3$ vectors.

The computation is distributed along the first dimension of A , such that for any i , all elements of the matrix $A_{i,j,k}$, $0 \leq j \leq N_2$, $0 \leq k \leq N_3$ are assigned to a single processor. So no communication is needed in the first two 1-D FFT phases. There is communication during the transpose. After this the processors access a different set of elements. Synchronization between processors is through barriers.

3.2.2 Barnes-Hut

Barnes-Hut is an application from the SPLASH [17] benchmark suite. It is an N-body simulation, in which every body is modeled as a point mass and exerts gravitational force on all other bodies in the system. The physical space is represented by a tree-structure, each leaf of the tree representing a body. Each internal node of the tree represents a *cell*, a collection of bodies in close proximity. Two arrays, one representing the bodies and another the cells, are the two main data structures. The array of bodies is shared and the array of cells is private. The simulation proceeds over time-steps, each step computing the net force on every body and then updating the body's position and other attributes.

The code for each time step is broken into five phases :

- **MakeTree:** Construct the Barnes-Hut tree.
- **GetMyBodies:** Partition the bodies among the processors.
- **Compute Forces:** Compute the forces on the bodies (each processor does it for its own set of bodies).
- **Update facts :** Update positions and velocities of bodies (each processor in-charge of its set).
- **Compute new dimensions :** Compute dimensions for next iteration.

MakeTree is sequential, as its parallel version is slower. In MakeTree each processor goes over the entire shared array for the bodies and reads the data for each body and then makes the tree. So each processor accesses the entire list of shared pages during this process. This is the main communication phase for the program. In GetMyBodies, dynamic load balance is achieved by use of the cost-zone method, in which each processor collects a set of logically consecutive leaves by doing a tree-walk.

The force computation segment is the compute intensive phase. There are barriers after GetMyBodies, Update and the new dimensions' computation.

3.2.3 Ilink

Ilink is a genetic linkage analysis program that locates specific disease genes on chromosomes. The program traverses the input *family trees* and visits each *nuclear family*. The main data structure in Ilink is a pool of *genarrays*. A genarray contains the probability of each genotype for an individual. The genarray is sparse and an array of pointers to nonzero values in the genarray is associated with each one of them. A pool of genarrays large enough to accommodate the biggest nuclear family is allocated at the beginning of the program, and is reused for each nuclear family. For a new nuclear family, the pool is reinitialized for each person in the current family. The computation either updates a parent's genarray conditioned on the spouse and all children, or updates one child's conditioned on both parents and all siblings.

The parallel algorithm described in Dwarkadas et al. [5] is used. Updates to each individual's genarray are parallelized. A master processor assigns the nonzero elements in the parent's genarray to all processors in a round robin fashion. After each processor has worked on its share of nonzero values and updated the genarray accordingly, the master processor sums up the contributions. The pool of genarrays is in shared memory and barriers are used for synchronization.

3.2.4 Shallow

Shallow is a benchmark from the National Center for Atmospheric Research. It is a code in Fortran that solves difference equations on a two dimensional grid, for weather prediction. Each processor is allocated a particular band, the sharing is only at the edges of the bands. Barriers are used for synchronization.

Application	Input
3D-FFT	64 x 64 x 64 – size of input vector, 100 iterations
Barnes-Hut	16384 bodies, 20 iterations (21, with timing after 1st iteration)
Ilink	CLP data set, with allele product of 2 x 4 x 4 x 4
Shallow	1024 x 256, 20 timesteps, 50 iterations

Table 3.1 Inputs for the Test Suite

3.3 Performance Evaluation

The experimental setup consisted of 8 SparcStation20s running SunOS 4.1.3 connected by an 155Mbps ATM network. All applications were run on 8 processors.

The results for the performance evaluation of our techniques are presented application-wise. In the discussion of the results, any reference to an application includes an implied reference to the input for the application (Table 3.1). This is because the same application can show a variation in behavior with change in the input due to change in false sharing and data access patterns with respect to the input.

In the rest of this section we present the results of the comparisons between regular TreadMarks and our techniques on the basis of execution time, data traffic and message count. In the following discussion, Tmk-**XK** indicates figures for regular TreadMarks with the page size of **XK**.

3.3.1 3D-FFT

3D-FFT was run for an input of size 64X64X64 for 100 iterations. Figures 3.1, 3.2 and 3.3 contain the relevant statistics.

The execution time and message count graphs show that 8K is the best fixed page size for 3D-FFT. This is primarily because of increased false sharing with a 16K page size. This causes a large increase in data traffic and allows only an insignificant decrease in the message count. In contrast, on moving from the 4K to 8K page size there is no false sharing. This results in a significant reduction in message count. This accounts for the reduction in execution time when moving from 4K to 8K pages.

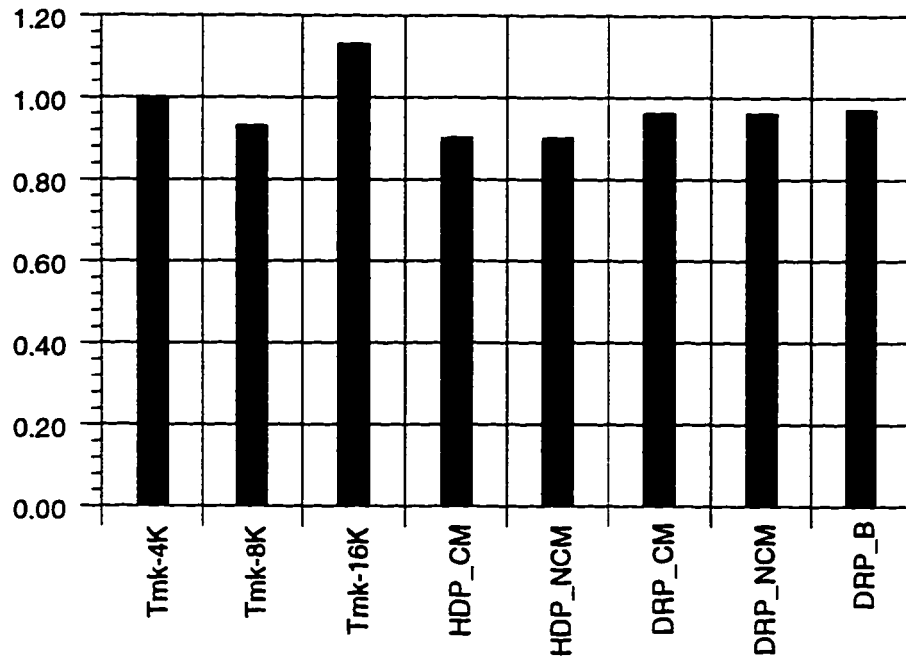


Figure 3.1 Execution time for 3D-FFT, normalized w.r.t Tmk-4K

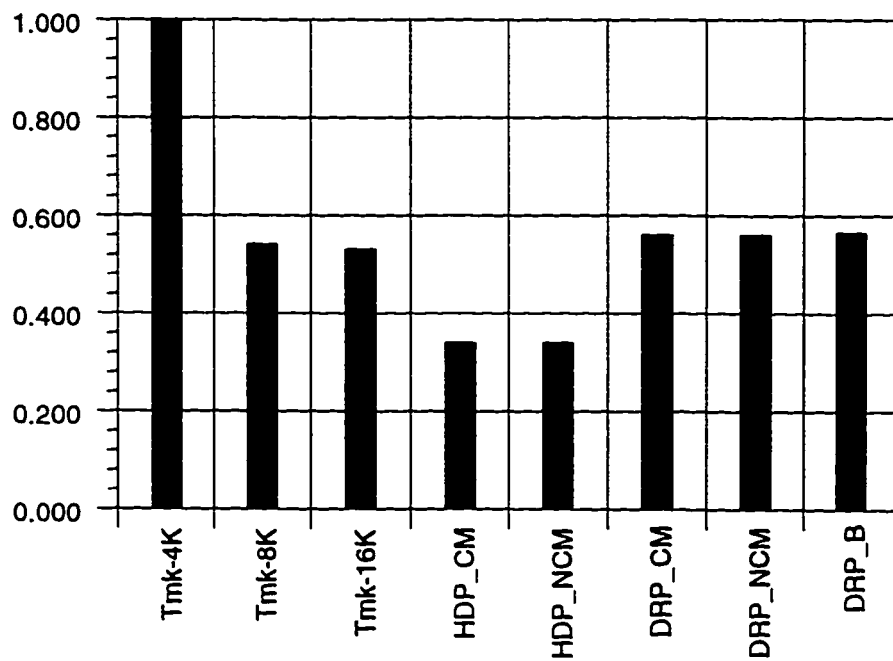


Figure 3.2 Total Message Count for 3D-FFT, normalized to Tmk-4K

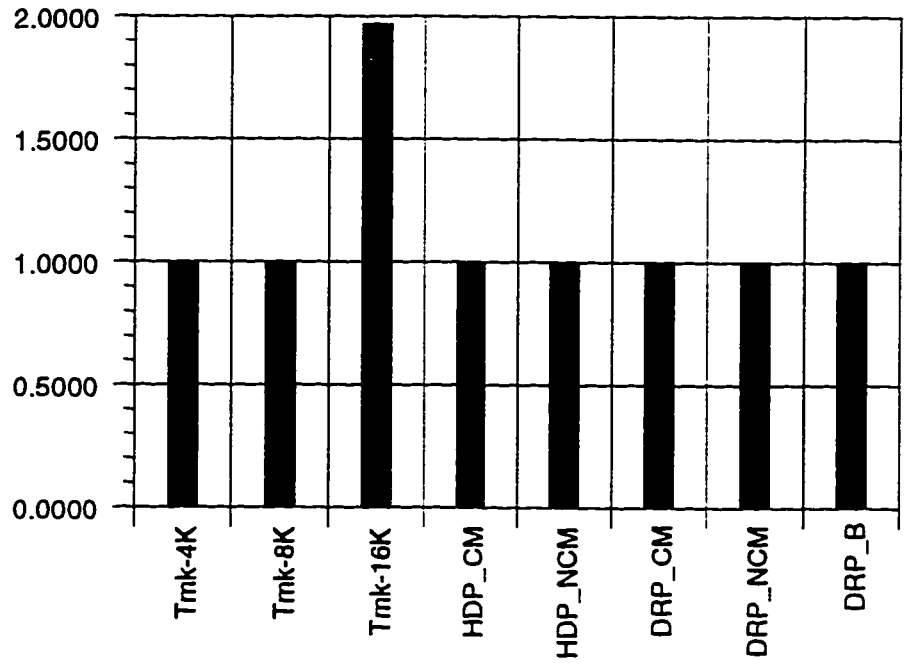


Figure 3.3 Data Traffic for 3D-FFT, normalized w.r.t Tmk-4K

3D-FFT has a well-defined and fixed access pattern. So all our techniques are able to quickly determine the page groupings, which remain the same from the beginning of the execution until the end. Because of the ability of the HDP schemes to combine requests for even non-adjacent pages, they provide a much bigger reduction in the message count than static 8K page size or any of the DRP schemes. All DRP schemes form page groups of size two and so their message count total is similar to that of Tmk-8K. The slightly larger message count for the DRP schemes is because of the initial single-page iterations – one for DRP_CM and DRP_NCM or two for DRP_B, with a merge threshold of one – when the page groups are still being formed, while Tmk-8K begins execution with the pages already in groups of two. The page groupings do not vary between critical sections, hence the performance of the NCM techniques is similar to that of the CM techniques.

The HDP techniques provide around 10% reduction in execution time through a 66% reduction in message count while the DRP techniques provide a 4% reduction in execution time through a 44% reduction in message count. All of them avoid the increase in data traffic which Tmk-16K exhibits. The automatic aggregation techniques could yield even better execution times in the absence of any overhead for the excess memory faults (resulting in a fault-count greater than Tmk-8K's, and the same as for Tmk-4K) that they incur to track the access patterns.

3.3.2 Barnes-Hut

Barnes-Hut was run for 21 iterations with statistics collected for the last 20 iterations with 16384 bodies. The first iteration was skipped to avoid the effect of cold-start misses. The execution statistics are presented in Figures 3.4, 3.5 and 3.6.

16K is the best fixed page size for Barnes-Hut. There appears to be no increase in false sharing when the page size is increased from 4K to 16K. So the reduction in message count with increase in page size brings the expected benefits in the case of Barnes-Hut.

The MakeTree phase of Barnes-Hut is instrumental for initiating the grouping of pages for the aggregation techniques. This is because almost all of the shared pages

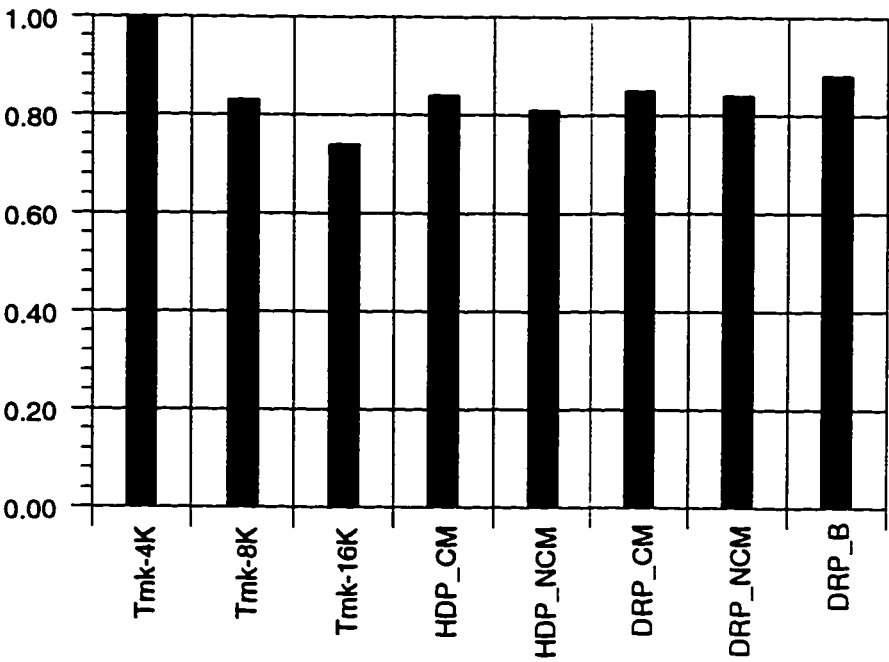


Figure 3.4 Execution time for Barnes-Hut, normalized w.r.t Tmk-4K

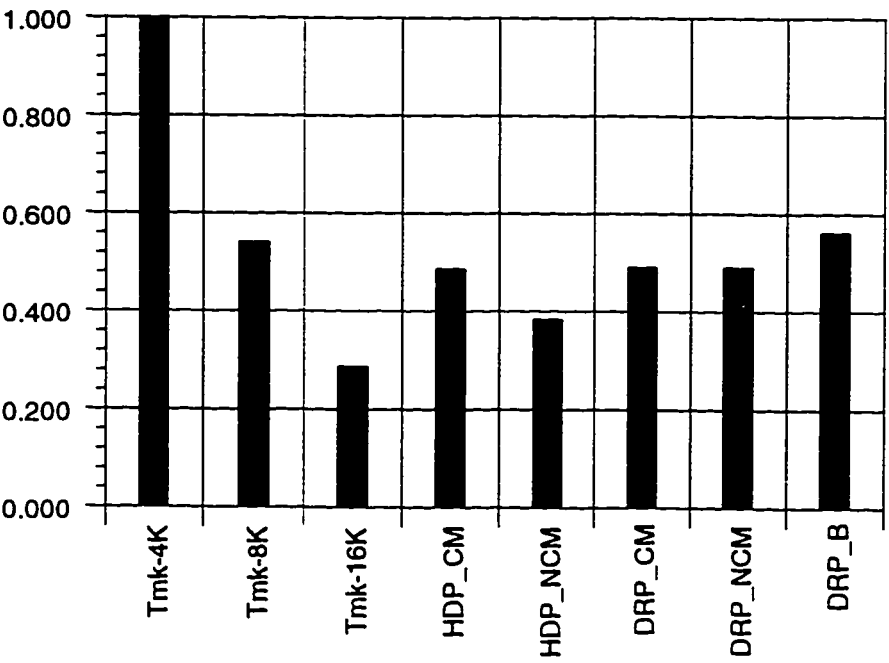


Figure 3.5 Total Message Count for Barnes-Hut, normalized to Tmk-4K

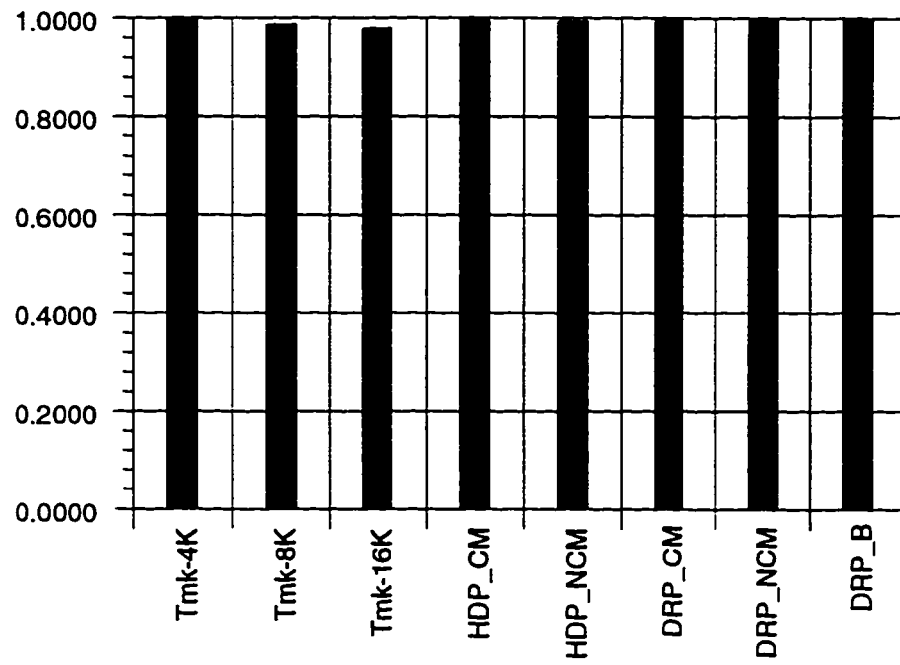


Figure 3.6 Data Traffic for Barnes-Hut, normalized w.r.t Tmk-4K

are accessed within MakeTree. Unfortunately, the accesses in later phases are not very consistent from one iteration to another. This is because the shared tree structure is divided up between the processors in different ways in each iteration. Thus, each processor could potentially access a different set of shared pages in the post-MakeTree phases in each iteration. But, since the set of pages which are accessed together is relatively stable, the NCM techniques which assume a global scope of groupings perform better than the CM techniques that assume the association of groups with critical sections. The restrictive nature of the buddy-grouping system causes it to perform the poorest among the automatic aggregation techniques.

An implementation characteristic of TreadMarks is further responsible for the performance difference between Tmk-16K and our techniques. TreadMarks has a memory reclamation phase called *Tmk_repo*. This is executed when the memory used for maintaining the consistency information has grown too large. At this point, this memory is reclaimed by ensuring that just a single copy of every shared page exists in the system. In such a situation, very little information needs to be stored for ensuring consistency. At this stage, there is considerable data-movement all of which can be made only at the granularity of a page. Tmk-16K, with its much larger page, sends only about one-fourth of the messages sent by our techniques during this phase. This is almost entirely responsible for the difference in message count between HDP_NCM and Tmk-16K.

The automatic aggregation techniques obtain a 12% to 19% reduction in execution time through a 44% to 62% reduction in message count, compared to the regular Tmk-4K execution.

3.3.3 Ilink

Ilink was run with the Cleft-Lip and Palette (CLP) data set and its execution statistics are presented in Figures 3.7, 3.8 and 3.9.

Tmk-16K gives the best performance for Ilink among the fixed-page-size executions. This is because there is no increase in false sharing when the page size goes from 4K to 16K. So the reduction in message count with larger page size directly

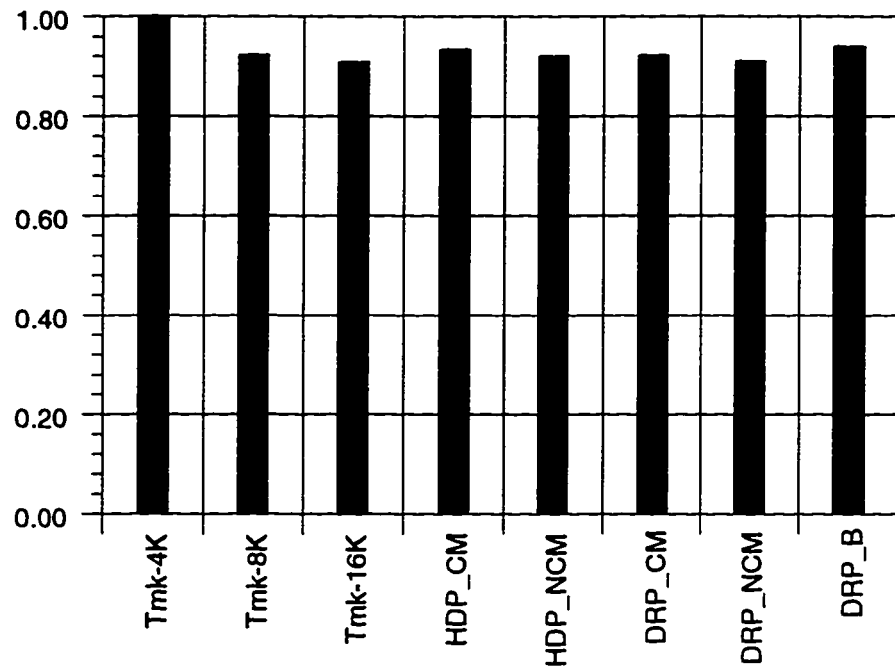


Figure 3.7 Execution time for Ilink, normalized w.r.t Tmk-4K

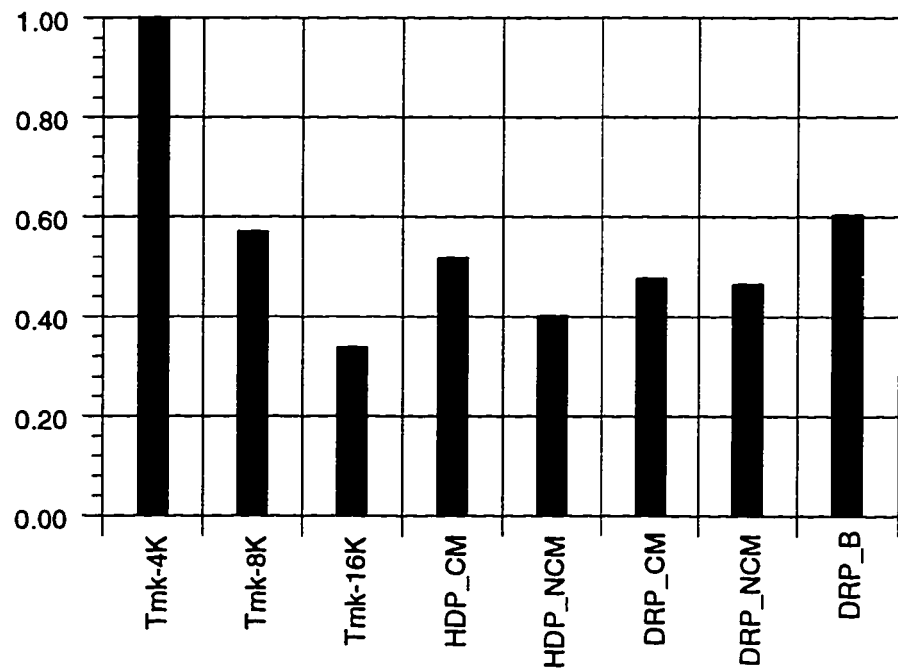


Figure 3.8 Total Message Count for Ilink, normalized to Tmk-4K

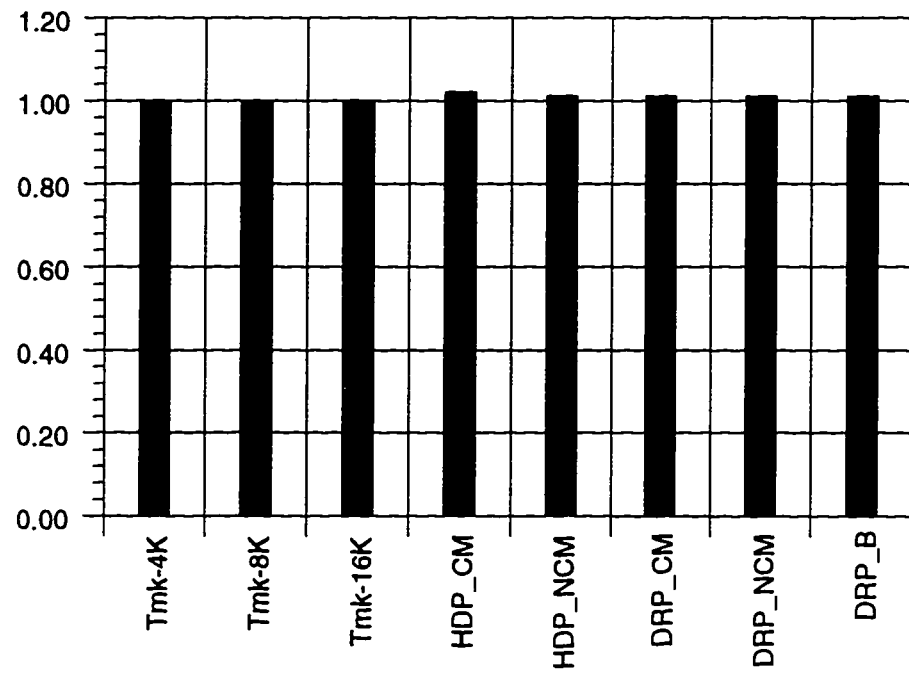


Figure 3.9 Data Traffic for Ilink, normalized w.r.t Tmk-4K

results in a reduced execution time.

The performance of the automatic aggregation techniques with Ilink is very similar to their performance in the case of Barnes-Hut. In Ilink the reuse of the shared data-structures, **genarrays**, for different data sets cause varying access patterns to the shared pages resulting in non-optimal aggregation with our techniques. The access patterns are dependent on the data contained in the structures. The reuse of memory for different data reduces the correlation between the access-history and the future accesses, resulting in poorer aggregation with the history-based techniques. The NCM techniques again perform slightly better than the CM techniques because the groupings are more relevant in a global scope than to particular critical sections.

Just as in the case of Barnes-Hut, the difference in the message count between HDP_NCM, the automatic technique with the lowest message count, and Tmk-16K is due to the data-movement at page-granularity during the *Tmk-repo* phase.

The automatic aggregation techniques provide a 6% to 9% reduction in execution time through a 40% to 60% reduction in message count in the case of Ilink.

3.3.4 Shallow

Shallow is run with a size of 1024X256 for the 3 input arrays, for 50 iterations. Figures 3.12, 3.10 and 3.11 contain the execution statistics.

Shallow exhibits an increase in false sharing with the increase in page size. The benefits due to reduction in message count with the increase in page size is more than offset by the increase in data traffic due to false sharing, with the result that Tmk-16K has a slightly larger execution time compared to Tmk-8K or Tmk-4K.

With the current problem size, there is not much opportunity for combining requests, with the result that the benefits from the aggregation techniques are quite small. Again, HDP_NCM provides the maximum reduction in message count. There is not much benefit in grouping only adjacent pages (because of the small size of the problem) and the groupings are relevant on a global range and not restricted to any

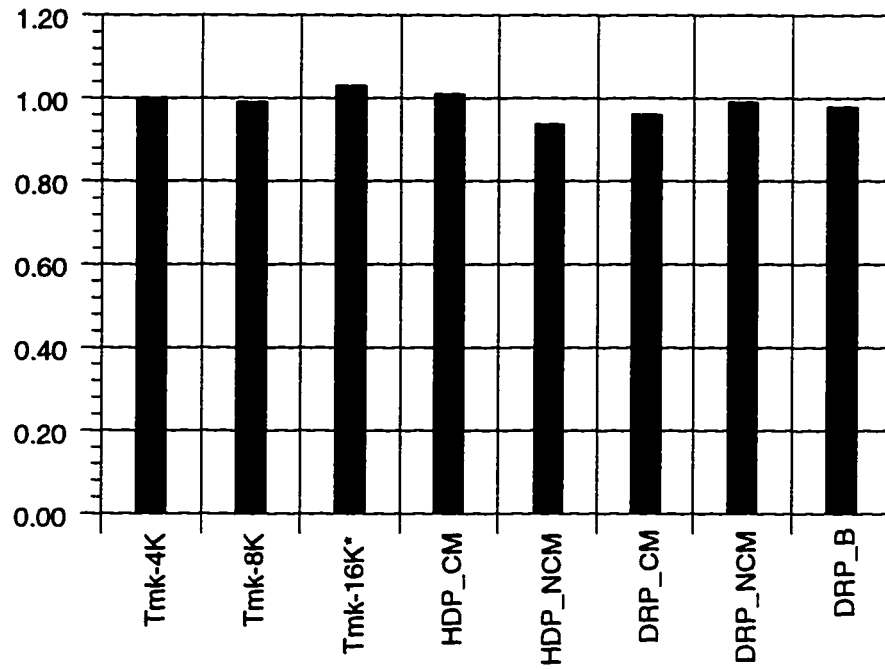


Figure 3.10 Execution time for Shallow, normalized w.r.t Tmk-4K

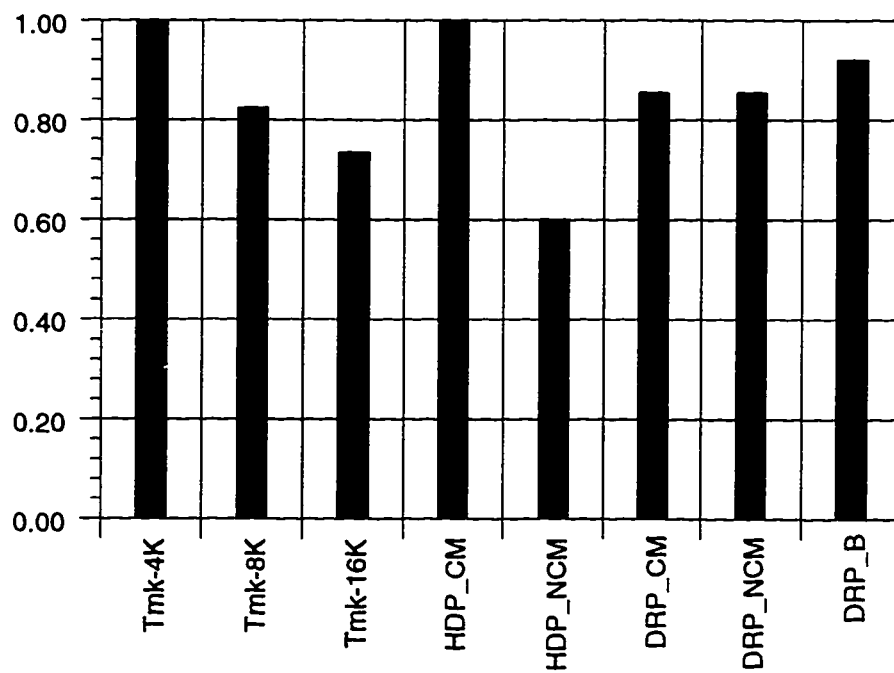


Figure 3.11 Total Message Count for Shallow, normalized to Tmk-4K

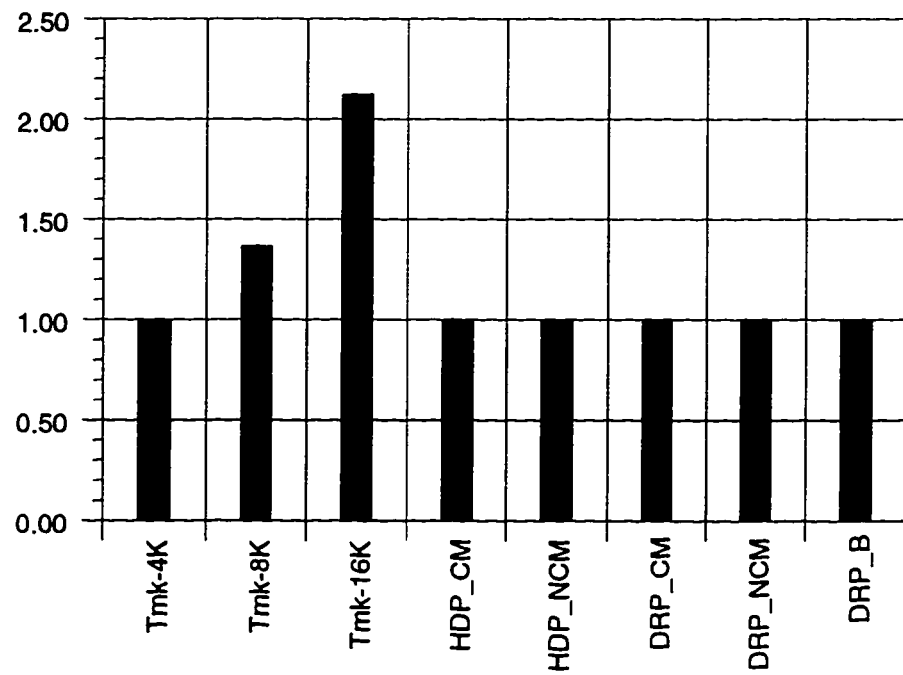


Figure 3.12 Data Traffic for Shallow, normalized w.r.t Tmk-4K

particular critical section. In our current implementation, HDP_CM has only one grouping per acquire-to-acquire interval which is recreated with every pass through that interval. This could explain the poorer performance of HDP_CM compared to DRP_CM, which can retain multiple-groupings (of different pages) per acquire-to-acquire interval).

The maximum improvement that our techniques can achieve on Shallow is a 6% reduction in execution time with a 40% reduction in message count which happens in the case of HDP_NCM.

3.4 Overhead Measurement

We present the results for the comparison between the performances of the regular HDP_CM implementation and its *optimistic* counterpart in this section. We also present the statistics for regular Tmk-4K and the best fixed size implementation for a measure of the overhead of our aggregation techniques due to tracking the access patterns using memory protection calls. The results are again application dependent, and are therefore presented separately for each application. We give just the figures for execution time (in seconds), total message count, and total number of memory protection faults seen for each application. The data totals for both optimistic and regular implementations do not vary significantly.

3.4.1 3D-FFT

Tmk-8K is the best fixed-page-size implementation for regular TreadMarks. The number of faults seen by Tmk-8K is around half of that seen by Tmk-4K. HDP_CM encounters the same number of faults as Tmk-4K as it tracks all data-fetch accesses at the granularity of a 4K page. HDP_CM-Opt has only 50% of the number of faults that HDP_CM has. This is almost the same as those seen by Tmk-8K. This brings down the execution time for the HDP implementation by another 1.6%.

Thus 3D-FFT illustrates that optimistic approaches are profitable when the access pattern is quite fixed and determined by the early stages of execution.

Implementation	Tmk-4K	Tmk-8K	HDP_CM	HDP_CM-Opt.
Time (sec)	144.17	134.09	129.30	127.01
Message Count	201677	108045	68621	68621
Faults	182755	91811	182755	95731

Table 3.2 Overhead Statistics for 3D-FFT, 64x64x64, 100 iterations

3.4.2 Barnes-Hut

With Barnes-Hut, Tmk-16K is the best fixed-page-size implementation. Compared to Tmk-4K, we see a 75% reduction in the number of faults encountered (Figure 3.3). In the case of HDP_CM, the optimistic approach obtains a 50% reduction in the number of faults seen by the execution. As can be seen from the Figure 3.3, the reduction in fault count with the optimistic approach is roughly proportional to the reduction in message count obtained by the non-optimistic aggregation technique.

In addition, the optimistic technique also brings a reduction in the message count. For Barnes-Hut the optimistic approach seems a better predictor of future access patterns than the regular approach, i.e, a repeated fetch list for an acquire-to-acquire interval is a better indicator of future data-fetch patterns than the fetch list of the immediately preceding execution of that interval.

3.4.3 Ilink

Tmk-16K is the best fixed-page-size implementation for Ilink. It brings a 69% reduction in the number of faults seen by the execution. The optimistic approach, on the other hand, brings almost no reduction to the number of faults seen by the HDP_CM

Implementation	Tmk-4K	Tmk-16K	HDP_CM	HDP_CM-Opt.
Time (sec)	118.12	87.38	99.34	93.89
Message Count	572452	164304	278196	238417
Faults	101758	26853	101758	50624

Table 3.3 Overhead Statistics for Barnes-Hut, 16384 bodies, 20 iterations

execution. This is because in our optimistic approach the scheme turns optimistic only after identifying an exact repetition in the fetch list corresponding to the particular acquire-to-acquire interval. With Ilink, such a situation does not arise until almost the end of the execution. So, the optimistic implementation also functions as a regular implementation for the most part of the execution. Hence, it does not obtain any reduction in fault count for Ilink.

3.4.4 Shallow

Tmk-8K is the best fixed-page-size implementation for Shallow. But, because of false sharing, the reduction in both message count and fault count is much less than 50% of the values for Tmk-4K. HDP_CM obtains no improvement in performance for Shallow, because of the low availability of combinable requests as detected by this technique. As in the case of Ilink, the fetch lists do not repeat themselves, so the optimistic scheme functions just like a regular HDP_CM implementation.

3.5 Analysis of thresholds in DRP-buddy-system

In this section, we analyze the DRP_B scheme separately, as it is somewhat different in its approach to aggregation, when compared to our other techniques. The existence of hysteresis in making the grouping or splitting decisions is what makes DRP_B different from the DRP_CM technique. As far as the implementation aspects are concerned, DRP_B is restricted to grouping adjacent pages following a buddy-style approach, while DRP_CM can group together any set of adjacent pages. The benefits of this freedom in DRP_CM can be seen from the better execution figures it obtained for all the applications as presented in the section 3.1.1. In this section we

Implementation	Tmk-4K	Tmk-16K	HDP_CM	HDP_CM-Opt.
Time (sec)	277.84	252.60	259.30	260.39
Message Count	255054	86204	132085	132062
Faults	130836	40830	131277	130069

Table 3.4 Overhead Statistics for Ilink, CLP data set

Implementation	Tmk-4K	Tmk-8K	HDP_CM	HDP_CM-Opt.
Time (sec)	34.72	34.42	35.10	35.18
Message Count	39114	32243	39044	39045
Faults	29766	24914	29766	29766

Table 3.5 Overhead Statistics for Shallow, 50 iterations

present the result of our study on the variation of DRP_B performance with change in the threshold values.

In DRP_B, with three possible sizes for the Mega-page – 4K, 8K and 16K – there are four possible transitions of page-states – by splitting from 16K to 8K or from 8K to 4K and by merging from 4K to 8K or from 8K to 16K. Hence we have four different threshold values – split thresholds for splitting from 16K to 8K and from 8K to 4K and merge thresholds for merging from 4K to 8K and from 8K to 16K. We studied the effect of threshold values by running all four applications with the three different initial sizes for the Mega-pages, with different settings for the 4 threshold values. Setting the merge threshold to zero results in merging being performed if the accesses in the last interval favor merging. With higher merge thresholds, merging would be performed only when accesses made in more than one interval in the immediate past favor merging. A similar relation is true for the split thresholds with respect to splitting.

The results showed that the threshold settings that yielded best performance, were found to be not only dependent on the application but also on the initial size for the Mega-pages. For applications that favored larger Mega-pages, we found that setting the merge thresholds to zero for smaller initial sizes gave some improvement in the execution statistics. For example in the case of Barnes-Hut we obtained a 3% reduction in execution time and 22% reduction in message count with a starting size of 4K for the Mega-pages and a zero value for both the merge thresholds. The optimal values for the split thresholds were dependent on the initial Mega-page size. Applications that favored a 8K Mega-page favored a large split threshold for the transition from 8K to 4K and a smaller one for the transition from 16K to 8K. Applications that favored a 16K Mega-page favored large split thresholds for both the transitions, this

was because only 8K Mega-pages could merge to 16K, so discouraging splitting from 8K state was also required to attain 16K state.

Our results for these threshold tests led us to conclude that there is no optimal set of threshold values that is common to all applications independent of the initial state of the pages. Our default settings of a value of one for all thresholds were found to be adequate to obtain reasonable improvement in performance with the `DRP_B` technique. But a knowledge of the optimal page size for an application could be used to guide the setting of the threshold values in such a way as to favor transitions to that size.

3.6 Summary

Our automatic aggregation techniques based on the history of data-transfer patterns were found to obtain improvements in performance for all the four applications in our test suite. They obtained reductions in execution time that ranged from 6% to 19%, and reductions in message count that ranged from 40% to 66%, without having any adverse effects on the amount of data-transferred across the network. Among the aggregation techniques the HDP schemes were found to perform better than our DRP techniques which in turn performed better than the `DRP_B` technique.

With the optimistic implementation of `HDP_CM`, `HDP_CM_Opt`, we obtained a 50% reduction in fault counts for 3D-FFT without any adverse effect on the message count or data traffic as seen by the execution. In the case of Barnes-Hut, in addition to a 50% reduction in the fault count, the `HDP_CM_Opt` also obtained a 14.3% reduction in message count resulting in a 5.5% reduction in execution time. With both Shallow and Ilink there was no significant gain or loss with the optimistic implementation.

The experiments analysing the effectiveness of thresholds in the `DRP_B` implementation proved inconclusive. No particular set of threshold values yielded the best performance consistent across all the applications or the initial states of the Mega-pages.

Chapter 4

Related Work

Each section of this chapter concentrates on a set of work that is similar to ours in one particular aspect. In the section 4.1, we look at work that addresses the message overhead problems in software Distributed Shared Memory (DSM) systems. The two groups in section 4.2 show the variation in application behavior with size of coherence unit in cache-coherent shared memory systems. In the section 4.3, we look at an implementation for dynamically resizing the coherence unit for a cache-coherent multiprocessor system. The work in section 4.4 presents coherence protocols that adapt themselves depending on the data-sharing patterns of the application.

The problem with software Distributed Shared Memory systems (DSMs) that our techniques address is message overheads for maintaining coherence. In the work presented in the first section other people address the same problems of message overheads, with similar approaches such as bulk-fetching and prefetching, but without run-time analysis of the sharing patterns.

The coherence unit for cache-coherent shared memory multiprocessing systems is the cache block size. Both the groups we look at in the second section show that performance is affected by the relationship between cache block size and sharing patterns exhibited by a program. This is similar to our results for page-size dependent application performance page-based software DSMs. The work in section three looks at an implementation of a cache-coherent multi-processing system with the cache-block size adjustable according to the application's requirements, predicted using previous access patterns.

In the fourth section we look at work that present on-line algorithms based on run-time data-sharing patterns where the algorithms adapt the system behavior to meet application requirements. Both the papers presented, look specifically at adaptive protocols that optimize coherence actions for migratory data in cache-coherent

multiprocessor environment. They are similar to our work, in that we too utilize the history of data-fetch accesses made by the program in order to reduce coherence overhead and optimize performance.

4.1 Messaging Overheads

Honghui Lu [15] et al. look at the performance difference between Parallel Virtual Machine (PVM), a message passing system, and TreadMarks for parallel programming on a network of workstations. They identify four factors that slow down TreadMarks - (1) extra messages due to separation of synchronization and data transfer, (2) extra messages to handle access misses caused by the use of an invalidate protocol, (3) false sharing, and (4) *diff accumulation* for migratory data. Their results show that the problem of extra messages could be mitigated by the use of *bulk-transfer* or data aggregation. This has a significant improvement for TreadMarks, obtaining a 3% - 28% improvement in the relative speedups (as a percentage of PVM's speedup) obtained for different applications in their test suite. For applications where data aggregation was significantly beneficial they found a 40% - 72% reduction in message count.

In [4], Dwarkadas et al. present the results of integrating compile-time and run-time support for software DSM. Their compiler computes the data access patterns in individual processes of an explicitly parallel shared memory program. It then inserts calls in the program to inform the run-time system of the computed data access patterns. The run-time system uses this information to aggregate communication, to aggregate data and synchronization into a single message and to replace global synchronization with cheaper point-to-point synchronization whenever possible. The authors find a significant improvement in performance due to communication aggregation for three of the four applications in their test suite. The improvements in program speedup range from 3% - 45% due to communication aggregation.

4.2 Variation in Performance with Size of Coherence Unit

Eggers and Katz showed in [6] and [7] that the performance of coherent caches depend on the relationship between the cache block size and the granularity of sharing and locality exhibited by a program, for bus-based shared memory multiprocessors.

They show that large cache blocks improve performance for applications with substantial processor and spatial locality, and cause an increase in the *miss ratio* for applications with fine-grain sharing patterns. Their results indicate that the optimal cache block size varies for different sets of applications.

In [19] Gupta and Weber examined the effect of cache block size on the number and size of invalidations in a multiprocessor system with a directory-based cache coherency protocol. They too noticed that different applications gave their best performances for different cache block sizes. The source for this variation in performance with cache block size was traced to the different sharing patterns among the applications.

Both the groups conclude that the different sharing patterns among applications require different sizes for the coherence unit to give their best performance. That provided some of the motivation for our attempting a dynamic page resizing scheme to reduce message overheads.

4.3 Dynamic Resizing of Coherence Unit

In [3] Dubnicki and LeBlanc proposed a scheme to reduce the impact on performance due to a mismatch between the cache block size and the sharing patterns exhibited by a given application. Their idea was to adjust the amount of data stored in a cache block according to recent reference patterns. In their scheme cache blocks were split across cache lines when false sharing was suspected and merged back into a single cache line otherwise, to exploit spatial locality. They quantified the performance benefits of adjustable cache block size by running traces of some parallel benchmarks on a simulated scalable multiprocessor with a write-invalidate ownership protocol for maintaining cache coherency. They used the average waiting time per memory reference and the number of 4-byte word transfers per memory reference as parameters for performance evaluation. They compared the adjustable block size implementation with various fixed size cache line organizations with the sizes ranging from 1 byte to 32 bytes.

Dubnicki and LeBlanc found that the adjustable cache-block-size implementation did better than the best fixed-size implementations for most of the program traces. They found that for every fixed block size, at least one program incurred a 33% increase in the average waiting time per reference, and a doubling of the average number of words transferred (over the network) per reference, when compared to the adjustable block size cache. They found only two fixed block sizes (4 bytes and 8 bytes) that were comparable in performance to the adjustable block size for most program traces. In the few cases where the fixed-block-size caches did better, the adjustable-block-size cache was within 7% of their performance.

We based our DRP-buddy-style implementation on this adjustable-block-size scheme.

4.4 Adapting based on data-sharing patterns

Cox and Fowler [2] presented an adaptive protocol for optimizing coherence actions for migratory data in cache-coherent shared memory architectures. Their implementations do not require any run-time or compile-time software support, but need minor modifications to the coherence unit part of the hardware. They advocate a *migrate-on-read-miss* policy instead of a *copy-on-read-miss* policy for migratory data. This moves the migratory block from one cache to another on a read-miss at the second cache, in one transaction as opposed to two transactions required with a *copy-on-read-miss* policy. As this could give poor performance for non-migratory data, it is necessary to use this policy only for migratory data.

The coherence protocol keeps track of the data access pattern to detect migratory data. For bus-based multiprocessors the implementation adds two new states - Migratory-Clean and Migratory-Dirty - to the states of the MESI protocol. The transitions among the Invalid, Migratory-Clean and Migratory-Dirty states implement the migrate-on-read-miss policy. The protocol adaptively switches between replicate-on-read-miss and migrate-on-read-miss as the detected data access pattern changes. An implementation of the protocol for directory-based Cache-Coherent Non-Uniform Memory Access (CC-NUMA) machines is also given in their paper.

Their work uses both trace and execution driven simulations to compare the adaptive protocols with standard write-invalidate protocols. Their simulations indicated that adaptive protocols could almost halve the inter-node communication compared to conventional protocols for the applications that had a larger portion of data-sharing as migratory, and reduce execution time by almost 19% for some applications.

Stenström et al. [18] also looked at adaptive protocols for optimizing coherence actions for migratory sharing. Their idea is very similar to that of Cox and Fowler. They suggest an implementation scheme for extending a protocol similar to the directory-based protocol of Stanford DASH [13] to include the adaptive capability. To evaluate its performance they use program-driven simulation of a DASH-like architecture [14] and a set of four benchmarks, three of which were from the SPLASH suite [17].

They concluded that by reducing the number of read-exclusive requests and by contention reduction (by reducing number of messages), their protocol could reduce the read-stall time under both sequential and relaxed consistency models. They also found a 20% reduction in network traffic for applications that exhibited migratory sharing. With reduced cache size (when replacement misses become significant) they found that the gain from the adaptive protocol decreased, but was still effective in detecting migratory sharing. They also performed a quantitative evaluation of the stability of their protocol - from this they found that the migratory sharing that their protocol detected was quite stable at least for the set of applications that they considered.

Thus both the papers prove the viability of on-line algorithms that guide the coherence actions to match application requirements, that are detected by examining data-sharing patterns during run-time.

Chapter 5

Conclusions

5.1 Goal

Our goal in this thesis, was to develop effective automatic data aggregation techniques to reduce message overheads in software Distributed Shared Memory systems. We used the history of accesses requiring remote data-fetch operations as a guide for predicting future data movement. Based on this prediction, we developed heuristics to combine the requests for remote data. The combining of requests reduces the message overhead by amortizing message costs over larger data-packets and by reducing the message count.

Our techniques varied in their approaches to aggregation on two parameters. The first was the domain of pages that can be grouped together. While the HDP techniques allowed any set of pages that required remote data fetch within the same acquire-follow-segment to be grouped together, the DRP restricted the set of combinable pages to those mapped to contiguous regions in the shared-address space. The second dimension was the scope of the page-groups once they were created. Under the CM classification, techniques allowed groups to have only a local scope (restricting it to just the acquire-to-acquire interval from whose access patterns the group was created), while the NCM techniques allowed a global scope for the page-groups.

In addition to the four techniques that we developed, we also implemented a technique proposed in the hardware DSM literature, DRP-buddy-style (DRP_B). All techniques were implemented on the TreadMarks DSM system. Our experimental setup consisted of 8 SparcStation20 workstations connected by a 155Mbps Fore ATM network. We demonstrated the effectiveness of our aggregation techniques by running many sets of experiments, using the code for four popular parallel applications - 3Dfft, Barnes-Hut, Ilink and Shallow. We had significant improvements for 3Dfft, Barnes-Hut and Ilink with all our techniques and a moderate improvement for Shallow

with some of our techniques. There was no significant degradation in performance for any of the applications in our test suite. Thus, we achieved our goal of developing automatic message-aggregation techniques through the design, development and verification of our implementations.

5.2 Conclusions

The success of our techniques indicate that automatic message aggregation is a viable idea for reducing message overheads in software DSM systems. From the detailed analysis of the results for each application (chapter 3), we can see the wide variation in behavior among the applications. The improvement in performance that our techniques bring to this varied set of applications indicate the suitability of our techniques for a wide set of applications that are run on software DSMs.

The usage of run-time analysis of the past access patterns to predict future accesses has also been demonstrated to be profitable over a range of applications. The success with Barnes-Hut and Ilink with only limited correlation between past and future accesses shows that a coarse indication from the access history is adequate to provide improvement in performance, when a large part of the memory sections get repeatedly accessed. The performance improvement with 3D-FFT shows the extent to which our techniques can benefit from a fixed well-defined access pattern even if it is not too straight-forward.

The dominance of the HDP techniques over the DRP techniques suggest that restricting the groupings to only adjacent pages may not be a good idea atleast for applications similar to those in our test suite. For applications where program exhibits group accesses to adjacent pages, the HDP techniques can extract the same benefit as the DRP applications as shown in the case of 3D-FFT. Under the DRP schemes, the better performance of both DRP_CM and DRP_NCM over DRP_B again suggest that restrictions on the grouping of pages hurt performance. The usage of thresholds for grouping might still be an idea worth exploring for our techniques.

The better performance of the NCM techniques over the CM techniques suggest that for the application we have seen, page-groups have a wider scope than a scope

restricted to just the regions whose access patterns led to their creation. This explains HDP_NCM getting the best results for almost all the applications.

We also found the HDP techniques easier to implement, as they avoided the checks for adjacency of pages and their ordering. So, both in terms of performance and ease of implementation, the HDP techniques seem to win over the DRP techniques.

5.3 Future Work

As we had mentioned earlier in chapter 4, there are ongoing efforts for integrating compiler-directives with run-time support for issuing combined requests for data. We could attempt to combine such work with the run-time information that we obtain for our techniques to get the best of both static and run-time analysis. This would, though, remove a major thrust for run-time analysis - that of using commercially available compilers without requiring any modifications in them. But a significant performance improvement when combining both could be sufficient to justify such an idea.

Bibliography

- [1] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. Technical Report 103863, NASA, July 1993.
- [2] A.L. Cox and R.J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.
- [3] Cezary Dubnicki and Thomas J. LeBlanc. Adjustable Block Size Coherent Caches. In *19th Annual International Symposium on Computer Architecture*, pages 170–180, May 1992.
- [4] S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.
- [5] S. Dwarkadas, A.A. Schäffer, R.W. Cottingham Jr., A.L. Cox, P. Keleher, and W. Zwaenepoel. Parallelization of Genetic Linkage Analysis Problems. *Human Heredity*, 44:127–141, 1994.
- [6] S.J. Eggers and R.H. Katz. A Characterization of Sharing in Parallel Programs And Its Application to Coherency Protocol Evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 373–383, May 1988.
- [7] S.J. Eggers and R.H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, April 1989.
- [8] G.A. Geist and V.S. Sunderam. Network-Based Concurrent Computing on the PVM System. In *Concurrency: Practice and Experience*, pages 293–311. June 1992.

- [9] R.J. Harrison. Portable Tools and Applications for Parallel Computers. In *International Journal of Quantum Chemistry*, volume 40, pages 847–863, February 1990.
- [10] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.
- [11] P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.
- [12] Peter Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Rice University, January 1995.
- [13] Daniel E. Lenoski, James P. Laudon, Kourosh Gharachorloo, Anoop Gupta, and John L. Hennessy. The Directory-based Cache Coherence Protocol for the DASH Multiprocessor. In *17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.
- [14] Daniel E. Lenoski, James P. Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John L. Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH Multiprocessor. *IEEE Computer Magazine*, pages 63–79, March 1992.
- [15] H. Lu, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. Quantifying the Performance Differences Between PVM and Treadmarks. *Journal of Parallel and Distributed Computing*, June 1997. To appear.
- [16] Parasoft Corporation, Pasadena, CA. Express user’s guide. version 3.2.5, 1992.
- [17] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):2–12, March 1992.
- [18] P. Stenström, M. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

- [19] W.D. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.