学位論文

Study on Optimizing Compilers to Support Software Distributed Shared Memory System

ソフトウェア DSM を支援する 最適化コンパイラに関する研究

平成 11 年 12 月 博士 (理学) 申請

東京大学理学系研究科情報科学専攻

丹羽 純平

Abstract

A coherent shared address space provides an attractive programming environment for parallel computing. If such an address space is to be used in a distributed-memory system without dedicated communication hardware, software remote-data caching mechanism (software Distributed Shared Memory: DSM) must be used. Optimizing methods are indispensable for improving the performance of Software DSM schemes. That is, compiler optimization, protocol optimization, run-time system optimization, and the interfaces that enable these optimizations are required.

We have introduced two compiler-assisted software DSM schemes as these interfaces. One is a page-based system (Asymmetric Distributed Shared Memory: ADSM) that uses the TLB/MMU mechanisms only for cache-misses. The other is a segment-based system (User-level Distributed Shared Memory: UDSM) that uses user-level checking codes and consistency management codes for software caching.

We have proposed a compiler optimization framework that directly analyzes the explicitly parallel shared-memory source programs and optimizes them. The optimizing compiler exploits the capabilities of middle-grained/coarse-grained shared-memory access to reduce the volume of communications and to reduce the overhead for the user-level checking codes. It performs interprocedural points-to analysis and interprocedural shared-access set calculations by using interval analysis to solve redundancy elimination equations along with lazy release consistency model. We have implemented this optimizing compiler, Remote Communication Optimizer :RCOP.

We have developed the cache-coherence protocols that follow the lazy release consistency model. Our experiment shows that SAURC (Software emulation of Automatic Update Release Consistency) protocol provides the best performance among the protocols that follow lazy release consistency model. We have developed the lightweight run-time system for cache-coherence based on SAURC, and we have implemented the run-time system for ADSM and UDSM on an SS20 workstation cluster. Both systems provide high speed-up ratios for the SPLASH-2 benchmark suite. The experimental results show that the combination of the optimizing compiler and Software DSM is very effective. The experimental results also show that the performance of the ADSM scheme is limited by the communication of unnecessary data, while that of the UDSM scheme is limited by the instrumentation overhead.

The results of this study indicate that executing parallel shared-memory programs with automatic optimizations for remote communications under a general-purpose operating system on a stock network of workstations is feasible.

Acknowledgments

I wish to thank Prof. Kei Hiraki and Mr. Takashi Matsumoto. They have provided me with advice, encouragement, and support througout my graduate career. I would also like to thank Mr. Tatsushi Inagaki (presently with Tokyo Research Laboratory, IBM Japan, Ltd), who taught me various things about compilers and have worked with me a few years. I cannot gain my result without his sincere advise.

My time at Hiraki Laboratory was made special by many friends. Junji, Kiyohumi and Ryouta (presently with ACCESS CO., LTD) all helped make my life at Hiraki Laboratory happy. I am grateful to my seniors, Mr. Shin'ich Furuso (presently with Sony Corporation) and Mr. Kanemitsu Ootsu (presently with University of Faculty of Engineering, University of Utsunomiya) for their kindnesses. I am also indebted to the system administrators at Hiraki Laboratory for maintenance of the research environment. I would like to thank all the other people of Hiraki Laboratory. Special thanks are also due to Mrs. Akiko Shintani, a super librarian of Dept. of Information Science. She always helps me search for literatures. I am greatly thankful to Mrs. Mary Inaba for treating me many times. I also thank for financial help by JSPS Research Fellowships for Young Scientists.

I am thankful to my parents and to my sister for their understanding.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Parallel processing has been developed because many recent applications require more computational power than is available from sequential processing. Parallel systems, however, have not yet been widely used because they tend to be expensive and to require a great deal of programming effort. There is, therefore, a need for general-purpose scalable parallel computer systems using off-the-shelf hardware.

Distributed-memory computer systems are called NUMA (nonuniform memory access) architectures, and are more suitable for scalable parallel computer systems than are UMA (uniform memory access) architectures such as SMP (symmetric multiprocessors) because NUMA architectures support more processors and because the fast local-memory-access in NUMA architectures is advantageous.

Recent progress in micro electronics has led to distributed-memory parallel computers, called computer clusters, consisting of off-the-shelf processors and networks. The performance of the off-the-shelf processors used in the workstations and personal computers is much the same as that of MPP (massively parallel processor). Fast commodity networks such as the Fast Ethernet and Gigabit Ethernet have come into wide use recently, and high-performance workstations and personal computers in offices or in laboratories are often connected by such commodity networks. The computer clusters thus formed are NUMA architectures and can be considered general-purpose scalable parallel computer systems.

1.1 Programming Model and Its Implementation

In the programming of parallel applications on computer clusters, automatic-parallelization of sequential applications is best for the programmer as long as it provides high performance. But because the current automatic-parallelizing compilers [7, 9, 49, 85] support only a limited class of applications such as regular computation, programmers need to write the parallel applications themselves. There are two programming models:

- Distributed-memory programming model

  This model, also called the message passing model, supposes that an address space is fragmented. That is, each process has its own address space and can access only the local memory of the processor executing that process. Programmers must distribute the data and explicitly specify the communication codes for accessing data in the local memories of the other processors. This makes the parallel programs difficult to design and debug. In addition, the algorithm choice is often limited by the data distribution.

- Shared-memory programming model

  This model supposes a single shared-address space: all memory locations are accessed by each processor. There is no need to distribute the data, and irregular computation (such as fluid dynamics computation and molecular dynamics computation) is easier to handle when all the processors have direct access to all the shared data.

The shared-memory programming model can thus be expected to make it much easier for programmers to write parallel applications, and to further simplify the programming, we need general-purpose scalable parallel computer systems that provide this model.

Programmers can easily write data parallel programs which are defined as single-threaded, shared-memory and loosely synchronous parallel computation. An effective way to execute these kinds of programs on distributed-memory parallel computers efficiently is to use an optimizing compiler, that translates these programs into SPMD (single-program multiple-data) programs [14, 25, 75, 48]. We call this approach the compile-time approach.

In this approach, shared data is distributed among the local memory spaces of the various processors according to programmer-supplied directives like those used in HPF (`ALIGN`, `DISTRIBUTE`, and so on) [38]. Computation is partitioned so that communication is reduced. Data is transfered among processors in explicit communication codes inserted by the optimizing compiler, and these codes are often aggregated into one large message to reduce communication overheads.

The explicit communication codes can be inserted only when the optimizing compiler analyzes the set of data that needs to be transferred by the processors (communication pattern). In many applications, however, such as those using a sparse matrix or an unstructured mesh, the communication pattern is not known at compile-time. These cases require the use of code-generation methods that enable run-time optimization.

The optimizing compiler translates the original computation loop into two phases. One is the pre-process routine, called inspector. The inspector calculates the communication pattern at run-time before the actual computation. The other is the computation loop, called executor. The executor consists of aggregated communication and the actual computation. The aggregated communication is performed using the communication pattern obtained by the inspector. This run-time scheme is called the inspector/executor scheme [25, 75, 48].

The overhead for the inspector in the inspector/executor scheme is large because the routine uses all-to-all communication. The inspector/executor scheme, therefore, is not efficient when the inspector is executed frequently. The compiler analysis needed to reduce the number of inspector routines is complicated [2], and this compile-time approach anyway does not easily handle pointers and multilevel indirect accesses [18]. In addition, data parallel programs cannot perform dynamic task scheduling using the lock mechanism.

Consequently, programmers often write explicitly parallel shared-memory programs that it is difficult or impossible to express in a data parallelism. To execute explicitly parallel shared-memory programs efficiently on distributed-memory parallel computers, we need to provide a shared-address space at run-time. This is called the distributed shared memory (DSM) scheme [16, 45, 50, 52, 53, 74], and it, of course, executes data parallel programs efficiently. In the DSM scheme, processors use memory loads and stores for communications. Every processor can directly address not only its own local memory but also the remote memories of every other processor. Remote-memory accesses are detected dynamically and the necessary communication is invoked. This run-time approach can handle pointers and multilevel indirect accesses easily because applications share a single address space. It can thus be used with applications that perform dynamic task scheduling and construct complex objects and data-structures.

The fact that the run-time approach can accept wider class of applications than can the compile-time approach suggests that the run-time approach is superior to the compile-time approach from the viewpoint of the user-interface.[1] The question asked here is whether or not the DSM scheme can provide high performance.

1.2 Challenges of Software DSM

In the DSM scheme, a remote-memory access through the interconnection network can cause a large latency, and this latency degrades the system performance. Keeping the memory accesses local is, therefore, important because the latency of a local memory access is much smaller than that of remote memory access. This is done by caching remote data in the local memory. This caching, however, introduces coherence problems in that processors must create local copies of remote data and keep the copies up-to-date.

The hardware approach, called Hardware DSM [16, 52, 50, 74] provides high performance but requires dedicated communication hardware and has a high implementation cost. The software approach called Software DSM [11, 17, 44, 46, 53, 39, 79] has several advantages.

[1]It should be noted that the run-time approach is not incompatible with compile-time approach.

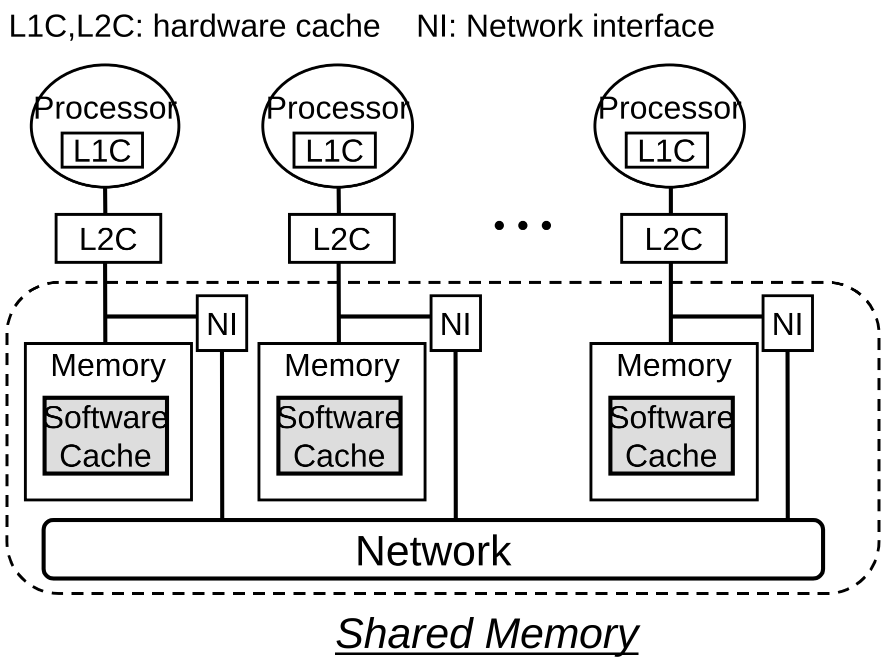


Figure 1.1: Software distributed shared memory (DSM).

- Its hardware overhead is low,
- It is easily implemented on the existing distributed-memory computers with commodity hardware, and
- Software-based coherence is more refined than hardware-based coherence.

Software DSM, thus, seemed more suitable for scalable parallel computer systems with off-the-shelf hardware, and the question, thus, becomes one of whether or not high performance could be obtained with Software DSM.

The coherence management techniques in Software DSM are based on those used in cache-coherent Hardware DSM, and the approaches to Software DSM can be classified into two kinds: page-based and segment-based. We are not concerned here with systems that force programmers to rewrite existing programs [11, 80, 44] because our goal is to make parallel programming easy for programmers.

Page-based approach

This approach, called Shared Virtual Memory (SVM), uses hardware page-management mechanisms [17, 46, 53, 39]. That is, page-fault traps and write-protection traps are used to trigger the coherence-management mechanism. Accordingly, the size of the coherence unit (i.e., the size of software cache) is equal to that of memory-page. A large page size leads to problems with false sharing [10] because conventional coherence protocols [51] force processors to gain sole ownership of the software cache (i.e., page) before they modify it. Even when multiple processors attempt to modify disjoint sets of data on the same page, they contest the ownership of the page. The "ping-pong" situation developing when page contents are transfered through interconnection network repeatedly, degrades the system performance.

A relaxed coherence mechanism was therefore developed, called lazy release consistency (LRC) [46]. It is a refinement of release consistency [30]. In LRC, coherence actions are postponed as long as possible. The write results of one processor in a critical section are not visible to another processor until it enters the critical section, or the write results of one processor are not visible to another processor until barrier synchronization is executed. LRC has been implemented in many systems [45, 39] together with multiple writer protocol [17] that allows multiple processors to write the different portions of the same page concurrently. This improves SVM performance because multiple writer protocol helps prevent false sharing.

Problems, however, remain in these systems using LRC with multiple writer protocol. One is that the overheads for coherence actions at shared writes are large because coherence actions are invoked before the corresponding store instruction and they must save the copy of the page to compute modification to the page. Another is that unnecessary data is transfered because the coherence size cannot be adjusted to the size most suitable for the application. This results in network saturation and degrades the system performance [27].

Segment-based approach

In this approach, all the coherence-management mechanisms are implemented in software [19, 79]. The coherence-management operations are statically inserted before each shared load and shared store by an optimizing compiler transparently rewriting the application executables. Because in this approach the size of the coherence unit is flexible, fine-grained accesses, which reduce false sharing and unnecessary data transfer, are supported.

There are several problems, however. One is that it is hard to make the best use of hierarchical structures (i.e., procedure calls and loops) in the program being compiled when it is a binary one. As a result, fine-grained communications may be invoked frequently. This is not desirable in computer clusters with off-the-shelf network hardware because such hardware has non-negligible communication overhead.

Another problem is one of overheads for coherence management. Since coherence management is implemented in software entirely, it is important to reduce instrumentation overheads. Even the peephole optimization performed by analyzing binary codes reduces the instrumentation overhead for coherence management, and it seems likely that performance can be further improved by analyzing the source program directly.

1.3 Outline of Our Approach

Our objective is to make it possible to run explicitly parallel shared-memory programs on distributed-memory computer systems with off-the-shelf communication hardware efficiently. The software DSM mechanism is used and remote-data-access latency is reduced by using the local processor memory as a software cache for remote data. The communication overheads and instruction overheads for coherence-management are minimized by using user-level software coherence-management and an optimizing compiler along with the relaxed coherence mechanisms LRC and multiple writer protocol.

The optimizing compiler directly analyzes explicitly parallel shared-memory source programs. In general, source codes of parallel applications are widely available and they are recompiled if we run them under the different platforms. It thus seems appropriate for the optimizing compiler to compile explicitly parallel shared-memory programs directly.

Although the optimizing compiler feeds the source program information back to the Software DSM run-time system, the optimizing compiler does not generate message-passing programs. It inserts in the source program the cache-coherence-management routines that invoke the run-time library. In other words, these routines inform the run-time library that the processor itself will soon issue shared reads from the region specified by the parameters or that the processor itself has already issued shared writes to the region specified by the parameters.

Two features of this approach are the followings:

- Specific cache-coherence-management routines are inserted.

  The coherence management is separated from the corresponding memory-accesses (loads/stores).

- Coherence actions are not invoked at each memory-access (load/store).

  The use of the relaxed coherence mechanism LRC together with multiple writer protocol enables the processor to postpone the coherence actions as long as possible.

These facts enable optimization. Roughly speaking, the optimizing compiler attempts to eliminate as many as possible of the redundant cache-coherence-management routines acting on the same region (i.e., cache-coherence-management routines whose parameters are same). The optimizing compiler also tries to issue cache-coherence-management routines with larger size (e.g., the array-size, the object-size, etc) than each memory-access size (e.g., byte, word, double-word). These optimizations make a program more efficient by reducing the overhead for cache-coherence management. The resultant program is compiled by a sequential compiler making good use of the platform information, then linked with the run-time library for user-level cache-coherence management to generate executable code.

1.4 Compiler-Assisted Software DSM

The optimizing compiler enables the Software DSM mechanism that accepts a wide range of shared-memory applications to execute these applications efficiently. It introduces integrated compile-time and run-time Software DSM schemes. In these schemes, a compiler/programmer exploits application's semantics to improve performance (e.g., aggregates the communication) while application can use the shared address directly. Software DSM schemes used in this thesis are as follows:

1. Asymmetric DSM[62, 60] (ADSM)

   In page-based approaches, the coherence actions are executed in trap handlers. Therefore, the optimization exploiting application's semantics cannot be performed. When LRC is used with multiple writer protocol in these approaches, the overheads for coherence actions at shared writes are quite large because of the copies created [45, 17].

   For instance, consider dynamic, irregular computation that requires run-time support. In inspector/executor schemes, the inspector overheads for identifying communication patterns are large. In the current page-based Software DSM schemes, on the other hand, the overheads for computing modification are large because the coherence actions are triggered only by write-protection traps.

   Remote accesses are detected dynamically when the run-time approach is used, and most of the inspector overheads in inspector/executor schemes are thus eliminated. On the other hand, if the compile-time approach is used along with a relaxed coherence mechanism, modifications to shared data are calculated efficiently in the executor inserted by the compiler, thus eliminating the coherence management overheads associated with current page-based Software DSM schemes.

   This example shows the operation of a hybrid of the page-based DSM scheme and the inspector/executor scheme along with relaxed coherence mechanism. In this hybrid scheme, remote-accesses are detected by hardware page-management mechanisms. On the other hand, the run-time system executes cache-coherence-management routines inserted by the compiler for shared-writes. That is, store instructions no longer trigger coherence actions at the shared-writes. The coherence management overheads are eliminated and the necessary communications are aggregated.

2. User-level DSM [60, 63] (UDSM)

   The problem in page-based approaches is the false sharing and transferring unneeded data that occurs because of the fixed size of the coherence unit (i.e., the software cache). If the volume of communications is to be reduced, the size of coherence unit must be flexible. This need for flexibility inspires the segment-based approach, but that approach has overheads for coherence management entirely in software and for the fine-grained communications that may be invoked frequently.

   Off-the-shelf network hardware has non-negligible overheads for communication and are therefore not well-suited to fine-grained communications. It is, thus, important to reduce not only the volume but also the number of communications in distributed-memory computer systems with such hardware. Code-generation methods that aggregate the necessary communications is required. There is thus a need for a segment-based Software DSM scheme integrated with inspector/executor scheme.

   In this scheme, the optimizing compiler directly analyzes not the binary program but the source program. The optimizing compiler, thus, makes best use of the hierarchical structures of the application (e.g., loops and procedure calls). Therefore, along with the relaxed coherence mechanism, the cache-coherence-management routines required for each shared-memory access are aggregated by using inspector/executer code-generation methods. This eliminates fine-grained coherence-management and communication. Consequently, the instruction overheads and the communication overheads are reduced.

   In this scheme, not only consistency-management at shared-writes but also the detection of a cache-miss/cache-hit are executed as user-level codes. When the platform supports user-level communication, the communication is also executed as user-level codes. It is appropriate to consider that this scheme still leaves much room to optimize coherence-management/communication for programmers/optimizing compilers.

1.5 Optimization Using Application's Semantics

Since the optimizing compiler directly analyzes a explicitly parallel shared-memory applications, it can perform various optimizations, making the best use of an application's semantics. The cache-coherence-management routine is the key to making the best use of the application's semantics. It is inserted by the optimizing compiler and it is executed as user-level codes by the run-time system. There are two primary interfaces: the cache-state checking routine and consistency-management routine.

1. Cache-state-checking routine

   This user-level routine informs the run-time system that shared-memory accesses will occur in a contiguous shared region. The parameters for this routine are the initial address and the size of the region. This routine ensures the coherence. That is, it checks the cache-state and invokes the cache-miss handler if the state is invalid. It is required before each shared-load and shared-store in UDSM.

2. Consistency-management routine

   This routine informs the run-time system that shared-writes have occurred in a contiguous shared region. The parameters for this routine are also the initial address and the size of the region. This routine executes the consistency-management at shared-writes. It is inserted after each shared-store.

1.5.1 Compiler Optimization

Non-negligible overheads are incurred when cache-coherence-management routines are inserted for each memory access. The optimizing compiler, therefore, performs the following optimizations.

- Precise shared-access detection

  There is no need to insert cache-coherence-management routine for memory accesses to private data such as stack data and static data because private data is never shared by processors. It is, thus, necessary to detect which accesses are shared and which accesses are private.

  Because the purpose of our system is to accept wide range of applications, it needs to handle applications written in C language. When the input programs are written in C, a shared address can be contained in a pointer variable and can be passed across procedure calls. Therefore, when a memory access through a pointer exists, a shared data may be accessed. It thus seems that cache-management routines for all the indirect accesses must be inserted. This approach causes non-negligible overheads for cache-coherence management.

  Therefore, the precise points-to information across procedures is essential to reduce these overheads. This information is obtained by using interprocedural context-sensitive points-to analysis [28, 82, 84].

- Relaxed coherence mechanism utilization

  In LRC, modifications to shared data by one processor are not visible until another processor executes the corresponding synchronization statements. This leaves much room for compiler optimization.

  Consider the following example. Multiple shared writes are issued onto contiguous locations and the program does not reach the synchronization point. It is not necessary for consistency-management routines to be issued immediately after each shared store. The consistency-management routine has only to be issued onto the coarse-grained region (i.e., the entire array) at once after all the stores. Figure 1.2 shows the example of this optimization.

  Consider the another example. Multiple shared writes to the same location are issued and the program does not reach the synchronization point. It is not a: shared array of doubles W: consistency-management routine

```
for (i = 0; i<n; i++) {
  a[i] += alpha * b[i];
  W (&a[i], 8);
}
```

⇓ Optimization

```
for (i = 0; i<n; i++) {
  a[i] += alpha * b[i];
}
W (a, 8 * n);
```

Figure 1.2: Optimization utilizing LRC in ADSM.

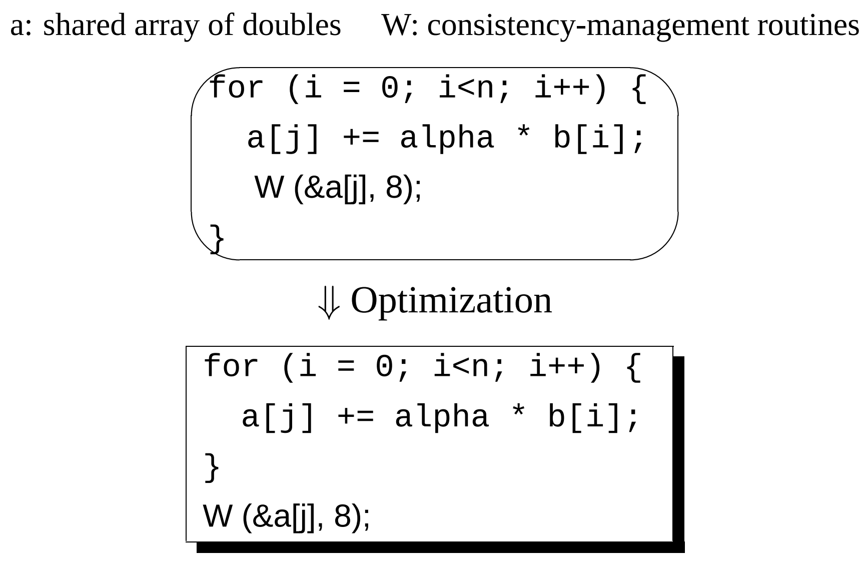
necessary for consistency-management routines to be issued exactly after each shared store. The consistency-management routine has only to be issued onto the region at once after the last store.

Advantage of cache-coherent-management routines from the view point of compiler

This interface considerably lightens the burden on the optimizing compiler. The above compiling techniques at first sight seem to be complicated, but whether or not a process issues the contiguous memory-accesses onto the shared region has nothing to do with other processes of the parallel program. It is shown in this thesis that explicitly parallel shared-memory programs can be efficiently analyzed interprocedurally.

Most of the current optimizing compilers [7, 34, 31], in contrast, try to parallelize sequential programs. They calculate the relations between production and consumption of data to directly insert communication codes. This code-generation method requires the solution of dataflow equations of array-reference, and only a limited class of programs can be analyzed in this way.

The other optimizing compilers [14, 2] try to generate message-passing program


a: shared array of doubles   W: consistency-management routines

```
for (i = 0; i<n; i++) {
  a[j] += alpha * b[i];
  W (&a[j], 8);
}
```

⇓ Optimization

```
for (i = 0; i<n; i++) {
  a[j] += alpha * b[i];
}
W (&a[j], 8);
```

Figure 1.3: Optimization utilizing LRC in ADSM (2).

from the data parallel program. They can handle irregular applications using the inspector/executer scheme, but this approach requires complex analysis techniques if the inspector overheads are to be reduced [35, 3].

In a parallel program, a parallel process is assigned to a physical processor dynamically. Where the written contents should be sent therefore cannot be calculated statically at compile-time. This problem is solved by the run-time system.

1.5.2 Run-time Optimization

When a cache-coherence-management routine is executed, the run-time library manages the coherence of all the caches within the region specified in the parameters of the routine.

The run-time overhead for collecting the written contents that is incurred in current page-based Software DSM schemes[17, 45] are eliminated by using consistency-management routines. Furthermore, the address translation overheads required in the inspector/executer scheme are not incurred because applications handle the shared address directly.

When communication is required in the cache-coherence management routines, aggregated communication is issued using the second parameter of cache-coherence-management routine. When the communication is invoked in the cache-state-checking routine, –that is, when a cache-miss is detected–, it is necessary to wait for the communication to be done because the data in the cache is used after the subsequent corresponding memory-accesses.

In the relaxed coherence mechanism LRC, in contrast, consistency management at shared writes has only to be done at the subsequent synchronization primitive. Hence, the communication in the consistency-management routine is invoked asynchronously. When the compile-time optimization does not work well, the consistency-management routine is issued for each shared-write. In this case, fine-grained communication packets may be issued frequently. Therefore, fine-grained communication packets whose destinations are the same processor can be combined into a large packet at run-time. Then, the subsequent synchronization primitive must wait for the completion of all the communication invoked in the previous consistency-management routines. If at this time there are combined packets not yet transfered, they are transferred to the corresponding processor.

In this way, the aggregated communication is executed efficiently by integrating compiler support with run-time support and using the relaxed coherence mechanism.

Asynchronous communication has a hidden cost, however, because it requires buffering when the receiver is not ready. Recently, commercial distributed-memory computers provide a direct remote-memory access mechanism [37, 71] (such as put/get), and even workstation clusters with commodity networks can supply this kind of mechanism [59]. Therefore, when the platform system provides the direct remote-memory access mechanism, we can further reduce the communication overheads by utilizing it as much as possible.

1.5.3 Protocol Optimization

Another advantage of our approach is that coherence management becomes more effective because the cache-coherence protocol is tailored to the application's semantics.

Multiple consistency protocols are supported by modifying the behaviors of cache-coherence-management routines to reduce the number of communications[56, 60, 61].

The performance of parallel shared-memory applications is determined by the cache-size. In ADSM, cache-misses are detected by hardware page-management mechanisms. The cache-size therefore must be the same as the underlying page-size. In UDSM, in contrast, programmers can freely specify the cache-size.

The performance of parallel shared-memory applications is affected by the cache-coherence protocol implementation. The relaxed coherence mechanism LRC is implemented in various ways (invalidate, update, etc). When the communication happens and what volume of data are transferred depends on the cache-coherence protocol implementation. The LRC has been implemented together with multiple writer protocol in three ways, so that the effect of the implementation on the performance could be studied [65, 68, 69]. The implementations, obtained by modifying the behaviors of cache-coherence-management routines, are the following:

1. Invalidate protocol

   The pages modified by other processors are invalidated based on LRC model. At a cache-miss, not the whole block but the modified part is obtained from the other processors. Multiple processors may have to be visited to make the cache-block up-to-date.

2. Home-Update protocol

   A home for each cache-block to which all updates are propagated and from which all copies are derived is maintained. The home is managed by an update scheme, while the other copies are managed by invalidation scheme.

3. Hybrid protocol

   This is a hybrid of Invalidate and Home-update protocols. It maintains for each cache-block a home, to which all updates are propagated. But, not the whole block but the modified part is derived from the home at a cache-miss.

Experimental results show that the home-update protocol provides the best results and that calculation of which data is modified requires a large amount of memory and a large number of messages. They also shows that the cost of maintaining a conventional time-stamp procedure is non-negligible. The timestamp procedure associated with each cache is used to determine what data needs to be exchanged [46].

Because the experimental results indicate that the home-update protocol should be implemented with the low-overhead timestamp mechanism, the home-update protocol has been implemented by maintaining for each cache the one-bit timestamp indicating whether or not it has been modified since the last barrier operation [41]. This lightweight run-time system has been implemented on distributed-memory computers with commodity network hardware [41, 67, 70].

The performance of parallel shared-memory applications is also influenced by the access pattern of the applications. For instance, system performance is markedly degraded by cache-miss traffic caused by the false sharing at shared writes. This traffic cannot be avoided by only the run-time support. The optimizing compiler avoids this traffic by omitting the cache-state checking routine before the shared stores in the UDSM scheme.

Our system, of course, provides an interface that allows the programmers to explicitly specify the coherence protocol, the cache-size, the cache-coherence-management routines.

1.6 Contributions

The main contributions of this work are the following five.

- Experimental verification of interprocedural context-sensitive points-to analysis

  Experiments using SPLASH-2 applications [86] have shown that interprocedural context-sensitive points-to analysis [28, 84, 82] detects shared memory accesses precisely.

- Extension and implementation of interprocedural shared-access summary analysis

Methods for summarizing the shared-memory accesses of explicitly parallel programs written on the relaxed consistency model have been improved. The previous work [41] has been extended substantially. Redundant shared-memory accesses are removed by using an interval-based approach to solve redundancy elimination dataflow equations. In the interval analysis, shared-memory-accesses are merged by using loop structures and procedure calls. This analysis has been fully implemented and used to show experimentally that SPLASH-2 applications are compiled into efficient codes.

- Development of optimizing methods using cache-coherence-management routines

  The above analysis reduces the instruction overheads for software cache-coherence management, and the run-time system issues the aggregated communication by using the compiler-inserted routines. Communication overheads are, therefore, reduced by integrating compile-time support with run-time support. Experiments using SPLASH-2 applications demonstrate that both instruction and communication overheads are reduced considerably.

  The relaxed coherence mechanism has been implemented in various ways, and empirical data showing that it is important to maintain a home for each cache-block to which all updates are propagated and from which all copies are derived has been collected. We have improved the home-update protocol with using low-overhead timestamp mechanism from this data.

- Identification of factors that affect the performance of shared-memory parallel applications

  Shared data allocation, access pattern, and the points of synchronization all influence the performance of explicitly parallel shared-memory applications, and the traffic caused by cache-misses at fetch-on-write is a bottleneck in several applications. It has been demonstrated that this bottleneck is eliminated by combining the compiler support and the run-time support.

- Demonstration of the feasibility of constructing scalable shared-memory parallel computers with commodity hardware

The lightweight run-time system has been implemented under a general-purpose scalable OS, SSS–CORE [60, 62], on the SS20 workstation cluster connected with a 100BASE-TX Ethernet. The results of our experiments with this implementation show that our approach achieves a high speed-up ratio with the SPLASH-2 applications.

The results indicate that explicitly parallel shared-memory programs for Hardware DSM can be executed efficiently on a distributed-system without dedicated communication hardware by integrating an optimizing compiler support with general-purpose OS support. Therefore, it is possible to make a general-purpose scalable shared-memory parallel computer by using commodity hardware.

1.7 Thesis Organization

Chapter 2 describes conventional Software DSM schemes, their problems, and the target DSM schemes solving these problems. Chapter 3 describes the compiler developed for explicitly parallel shared-memory programs. It also describes the design and implementation of the various optimizing methods used in this "Remote Communication Optimizer"(RCOP). Chapter 4 discusses the implementation of the lightweight run-time system and implementation of the cache-coherence protocols. Chapter 5 shows the effectiveness of the optimizing methods by presenting experimental data obtained when using the SPLASH-2 applications. It also discusses the evaluation of parallel performance and compares the ADSM and UDSM schemes quantitatively to demonstrate the effects of compiler optimization. Chapter 6 discusses related work, mainly about compiler-assisted DSM and interprocedural optimizing compilers. Chapter 7 concludes by summarizing this thesis and indicating directions for proposed optimization techniques and compiler-assisted Software DSM.

Chapter 2

Target DSM Schemes

Software DSM provides a shared address space at run-time and accepts wider range of applications, and it is easy to implement on the existing systems with commodity hardware. Optimizing methods are indispensable for improving the performance of Software DSM schemes. Therefore, the interfaces that enables optimizations are required.

These interfaces used in this theses are Asymmetric Distributed Shared Memory (ADSM) [60, 62] and User-level Distributed Shared Memory (UDSM) [60, 63]. This chapter explains these two compiler-assisted DSM schemes and highlights their features by comparing them with the conventional Software DSM scheme.

2.1 Basic Assumptions

The platform we assume here is a distributed-memory computer system with commodity network hardware, such as a workstation cluster. Each processor in the system is assumed to have hardware page-management mechanisms (TLB and MMU). The virtual address space of each processor in the cluster is divided into a private region and a shared region. Data in the shared region can be cached by multiple processors at the same time, with copies residing at the same virtual address on each processor. The shared address space is divided into ranges of memory called blocks. All the data in a block is fetched and kept coherent as a unit.

2.2 Background and Motivation

Consider that the caching mechanism used in the Hardware DSM is strictly emulated in all software. In this approach, a processor needs to test for cache-miss/cache-hit, to create of data copies, and to manage consistency every time there is a memory-access. Overheads for the software execution of the cache-coherence management are thus considerably large.

The most common approach of Software DSM, called Shared Virtual Memory (SVM) [17, 39, 46, 53], uses hardware page-management mechanisms. That is, TLB and MMU are used to decide whether there is a cache-miss or a cache-hit and to execute consistency-management codes at write operations. SVM was first implemented in the IVY system [53, 54].

The IVY system reduces the coherence-management overhead when a processor hits the cache, but it makes this overhead when a processor misses the cache or modifies the cache comparatively large because a processor executes a trap or system call routine. Furthermore, the large size of the page leads to problems with false sharing [10]. The IVY system supports only conventional coherence protocol, that is, sequential consistency (SC) [51]. In SC, the value returned by the read operation is always the same value as the most recent write operation to the same address. Therefore, processors are forced to gain the sole ownership of the software-cache (i.e., page) before it is modified. Even if multiple processors attempt to modify disjoint sets of data on the same page, they contest the ownership of the page and the page contents are transfered repeatedly, degrading the system performance.

Taking account of the large overheads for software coherence management, we can see that the coherence mechanism must be optimized. Release consistency (RC) is proposed by Gharachorloo et al. [30]. It was implemented in Hardware DSM mechanism [52]. In RC, the processor is allowed to postpone making its modifications to shared data visible to other processors until it executes a subsequent synchronization primitive. For instance, consider the example shown in Figure 2.2. Each processor has a cache y. P_1 and P_2 also have a cache x. acq (acquire) and rel(release) respectively correspond to the synchronization primitives lock and unlock. Lock and unlock

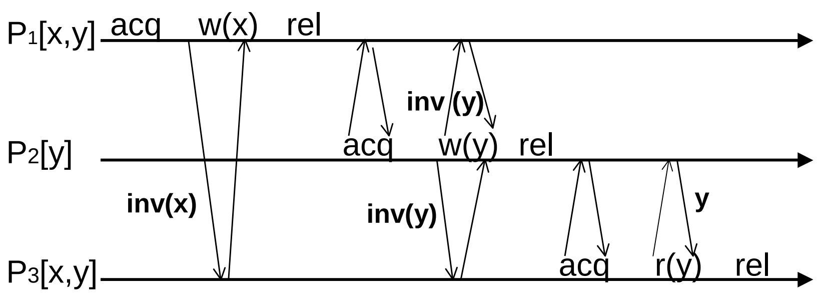


Figure 2.1: Sequential consistency (Invalidate protocol).

are used to ensure mutual exclusion among processes. When P_2 modifies the cache y after acquiring the lock, its modification is not visible to P_1 and P_3 at this time. When P_2 comes to the subsequent synchronization point (i.e., the release point), P_2 must propagate its modification of the cache y to the processors P_1 and P_3. Munin [17, 42] adopted RC.

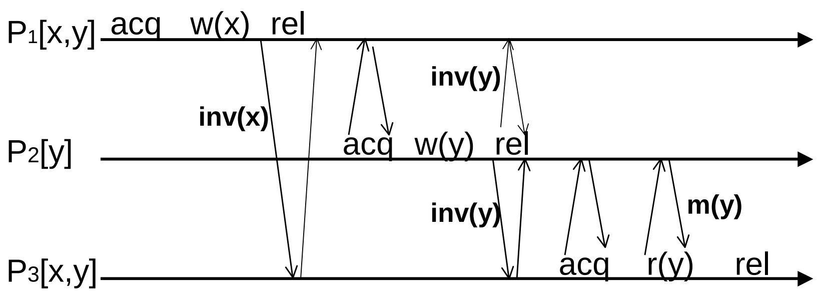


Figure 2.2: Release consistency (Invalidate protocol).

Lazy release consistency (LRC) [46] was introduced in TreadMarks [45]. It is extension of RC, and differs from RC in the way the modifications to the data are propagated. LRC postpones propagating the write results until a lock is acquired, as opposed to before the corresponding lock release. For instance, consider the example shown in Figure 2.3. This example is the same as the example 2.2 except that it is under LRC. When P_2 releases the lock after modifying the cache y, P_2 does not propagate its modification. Subsequently, when P_3 acquires the same lock, the modification of the cache y is propagated to P_3. The modification of the cache x by P_1 is also propagated to P_3 at this acquire.

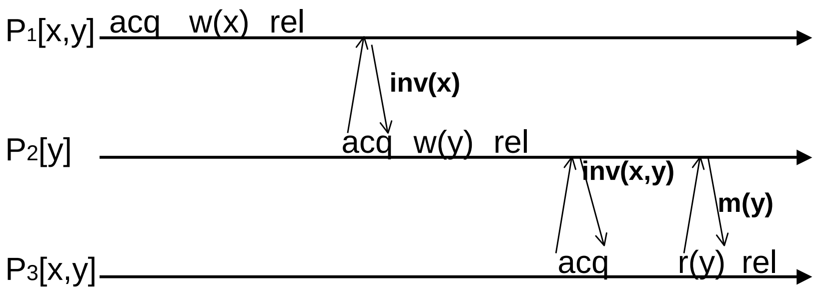


Figure 2.3: Lazy release consistency (Invalidate protocol).

In RC and LRC, multiple writer protocol is used to reduce the false sharing within a cache (i.e., page). This protocol enables multiple processors to write to the different parts of the same cache concurrently. In both Munin and TreadMarks systems, before the cache is modified, the copy of the page must be created to compute modification of the cache [45, 17] as shown in Figure 2.4. The copy is called twin. The subsequent release operation requires modifications of the cache to be collected. The collection is done by comparing the current cache with the twin as shown in Figure 2.4. The collection of modifications is called diff.

It should be noted, however, that there are still several problems. For one thing, the overheads for coherence actions at shared-writes and release operations are large because the twin and diff operations cause considerable overheads [39]. For another, more than one remote processor may have to be visited in order to obtain diffs at the cache-miss.

There is an approach that reduces overheads for twin and diff operations by using additional communication hardware support [39, 40]. Since the network-interface hardware forwards local writes to the remote memory transparently, the twin and diff operations are no longer required. This approach uses a home memory for every cache (i.e., page). Writes to the other copies of the cache are propagated to the home

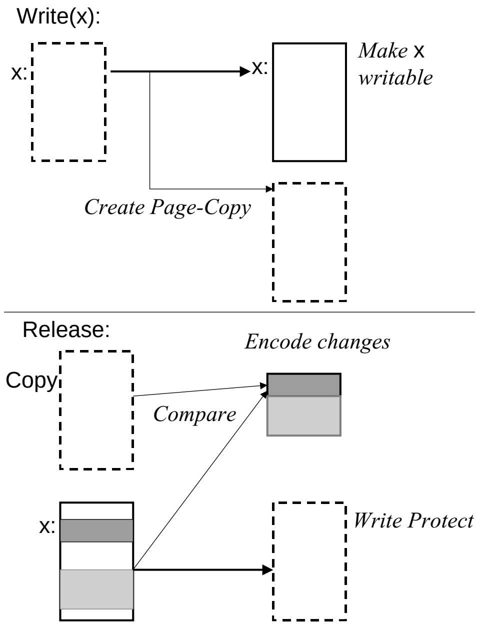


Figure 2.4: Diff creation.

by snooping all the write traffic on the memory bus . In this way the home is used to collect updates from multiple writers and is always kept up-to-date, while the other copies are updated on demand. This mechanism was first implemented in SHRIMP multiprocessors [12] and is called automatic update release consistency (AURC).

The AURC scheme, however, also has several problems. For one thing, additional communication hardware is undesirable for a general-purpose scalable parallel computer system with commodity hardware. For another, the AURC scheme generates a large amount of communication traffic because communication occurs every time there is a shared-write to the copy page and the entire page is transferred at a cache-miss.

These SVM approaches always issue shared-memory-accesses as ordinary memory-accesses (loads/stores). A coherence management mechanism is triggered only by the trap. The collection of updates from multiple writers either requires large software overheads, such as those for twin and diff operations or requires a especial hardware mechanism. It is safe to say that these SVM approaches do not provide interfaces that enable compilers/programmers to perform optimization.

There is an object-based approach that allows programmers to perform various optimizations [11, 80, 18]. Programmers specify a coherence protocol for each shared data and explicitly declare association between data and synchronization. False sharing does not occur in this approach, but, there are always overheads for packing/unpacking messages. Moreover, existing shared-memory applications require rewriting. The effort required for porting them can be substantial.

There is segment-based DSM called Shasta [79] that instruments loads and stores in application binary to check for accesses to remote data. This approach provides shared memory in all software and supports fine-grained sharing, thus reducing false sharing and unnecessary data transfer. The instrumentation overhead is reduced by optimizing techniques, such as batching and invalid flag techniques. Batching is a technique merging checks of multiple loads and stores only when their base registers are the same and their offsets are less than or equal to the line-size (64-128 bytes). Consequently, Shasta does not utilize coarse-grained shared-memory-access although it uses release consistency. The reason for this is that the input program of the Shasta compiler is binary one. Of course, neither loop-level optimization nor interprocedural optimization is performed. Therefore, this leads to the frequent fine-grained communication and commodity network hardware is not good for this kind of communication.

2.3 Asymmetric DSM

To reduce the overheads for coherence actions at shared-writes and to optimize shared-writes operations in page-based systems, a page-based software DSM called Asymmetric DSM (ADSM)[60, 62] has been proposed. It combines the page-based approach with the inspector/executor approach. The compiler can use TLB/MMU mechanisms to detect remote-accesses dynamically, thereby eliminating the inspector [48, 75] overheads. On the other hand, the run-time system can execute compiler-inserted routines (like executor) for shared-write-accesses. These routines aggregate communications and consistency-management operations along with LRC.

- Cache-States

Shared data in the ADSM scheme has only two states:

  - `invalid`

    The data is invalid on this processor. If this processor attempts to read the data or write the data, a page fault trap is triggered.

  - `read_and_write`

    The data is valid on this processor. This processor can both read and write the data.

- Shared-read

  A shared-read operation is executed as a load instruction. If a processor attempts to read data that is `invalid`, a page-fault trap is executed. The trap routine invokes the user-level cache-miss handler, which obtains an up-to-date copy of the block by communicating with other processors and sets the state of the cache `read_and_write`.

- Shared-write

  A processor must fetch the cache before it writes the data. This causes a page-fault trap when the cache is `invalid`, and the trap routine invokes the user-level cache-miss handler. The handler obtains an up-to-date copy of the block by communicating with other processors and sets the state of the cache `read_and_write`.

  Furthermore, the codes for managing cache-consistency are required for shared-write operations, and they are executed as user-level codes. They are separated from the corresponding store instructions and explicitly inserted after the corresponding store instructions by the optimizing compiler.

The strategy for handling shared-read operations is different from that for handling shared-write operations. Therefore, this scheme is called "asymmetric" DSM (ADSM). It requires both OS support and user-level cache management. ADSM can be implemented on any distributed system. We have implemented ADSM under a scalable OS SSS–CORE [57, 62, 60] on the SS20 cluster. We call the implemented system ADSM/SSS–CORE system.

2.4 User-level DSM

It is important to reduce both the volume and the number of communications in distributed systems with off-the-shelf network hardware. The fully user-level software-cache scheme (User-level DSM:**UDSM**), where application programs only use user-level codes to maintain software-cache coherence, is better with regard to code optimizations for inter-node communication than are page-based software DSM schemes. In other words, UDSM is more suitable than page-based software DSM schemes in that it offers more opportunities to optimize communication and cache-coherence management codes.

UDSM is hybrid of the segment-based approach and the inspector/executor approach. The fine-grained communication and cache-management overheads found in the Shasta system are avoided by merging user-level cache-coherence-management codes into a coarse-grained/middle-grained one along with LRC. This is done by exploiting the code-generation methods used in the inspector/executor approach [3, 2].

- Cache-States

  Shared data in the UDSM scheme, like that in the ADSM scheme, has two states:

  - `invalid`

    The data is not valid on this processor.
  - `read_and_write`

    The data is valid on this processor.

- Shared-read

  Cache-state checking is required in order to ensure coherence. The optimizing compiler explicitly inserts a checking routine as user-level codes before the corresponding load instruction.

When a cache-miss is detected in the checking routine, the user-level cache-miss handler is invoked. The handler obtains an up-to-date copy of the block by communicating with other processors and sets the state of the cache `read_and_write`.

- Shared-write

  As in ADSM, the optimizing compiler explicitly inserts a consistency-management routine as user-level codes after the corresponding store instruction. Cache-state checking is also required before the corresponding store instruction in order to ensure coherence.

UDSM also can be implemented on any distributed system. We have implemented UDSM under the SSS–CORE on the SS20 cluster. We call the implemented system UDSM/SSS–CORE system.

2.5 Features of Target DSM schemes

Features of target DSM schemes are described here by comparing the target schemes with SVM (the conventional page-based scheme).

- (Software-)cache block-size

  The cache block-size of ADSM is made equal to the size of a memory-page, because cache-state checking is implicitly supported by the virtual memory hardware.

  The UDSM scheme, in contrast, checks the cache-state by using explicitly inserted user-level codes, and the cache block-size is flexible. If it is fine-grained, the overhead for handling the cache state is not negligible and fine-grained communication, which a commodity network is not good at, may occur frequently. But if the block-size is too large, excessive false sharing will occur and make traffic heavy. The block-size, therefore, must be adjusted to improve performance.

Table 2.1: Comparison between SVM and our target schemes.

	SVM	ADSM	UDSM
Cache block-size	vm page	vm page	flexible (user-defined)
Cache states	INVALID R_AND_W R_ONLY	INVALID R_AND_W	INVALID R_AND_W
Miss/Hit check	MMU/TLB	MMU/TLB	explicit user-level codes
Consistency management	MMU/TLB (trap handler)	explicit user-level codes	explicit user-level codes
Protocol	fixed	flexible (user-defined)	flexible (user-defined)
User-level optimization	impossible	write	read write

vm: virtual-memory

R_ONLY: read_only R_AND_W: read_and_write

- Cache-state

  In ADSM and UDSM, there is no `read_only` state like that used to execute consistency-management in SVM because accesses requiring consistency-management (i.e., shared-writes) are detected statically.

- Miss/Hit check

  In SVM and ADSM, the decision of cache-misses/cache-hits is supported by TLB/MMU mechanisms using page-fault traps.

  In UDSM, an explicit user-level routine checks the cache-state. The optimizing compiler remove redundant checking routines and merges multiple checking routines onto contiguous regions in order to reduce the overhead for cache-management. The restriction in batching found in Shasta is not found in UDSM.

- Consistency-management

  In SVM, shared-writes are detected by TLB/MMU mechanisms using write-protection traps. The consistency management is executed in the trap handler.

  In ADSM and UDSM, shared writes are detected by the optimizing compiler, which explicitly inserts user-level routines for cache-consistency management. The optimizing compiler removes redundant consistency-management routines and merges the consistency-management routines onto contiguous regions.

- Protocol

  Various protocols can be used in ADSM and UDSM by modifying the behaviors of the consistency-management routines and the state-checking routines [60, 61].

- User-level Optimization

  SVM does not provide interfaces that enable compilers/programmers to perform optimization. ADSM, on the other hand, offers a programmer/compiler chances to optimize shared-write accesses. UDSM provides a programmer/compiler with opportunities to perform optimize both shared-read accesses and shared-write accesses.

Execution Model

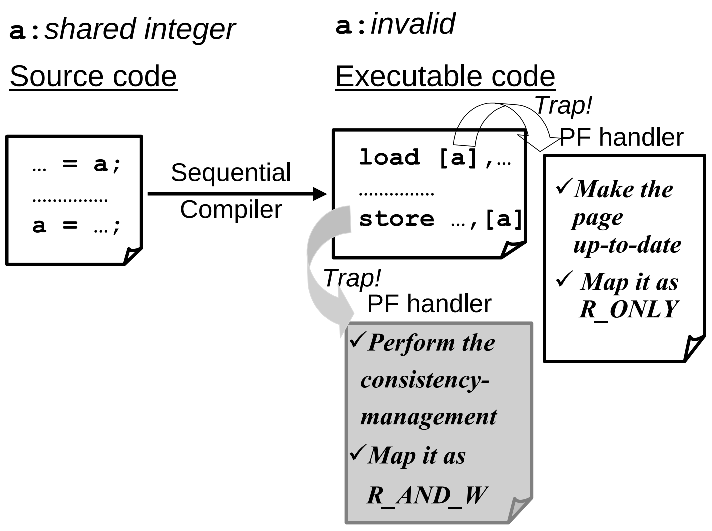


Figure 2.5: SVM execution model.

Figure 2.5 illustrates the SVM execution model. In SVM, the source program is compiled by sequential compiler to generate executable codes. Suppose that a is shared and is invalid when the executable code runs. A cache-miss is detected as a page-fault trap and the handler maps the page as `read_only`. The shared write is detected by a write-protection trap and the handler executes the consistency-management and maps the page as `read_and_write`.

Figure 2.6 illustrates the ADSM execution model. In ADSM, the optimizing compiler directly compiles the source program in order to detect shared-write operations and insert user-level consistency-management routines. A cache-miss is detected as a page-fault trap and the handler maps the page as `read_and_ write`. The consistency-management is executed by an additional user-level routine (denoted by W).

Figure 2.7 illustrates the UDSM execution model. In UDSM, the optimizing compiler directly compiles the source program in order to detect all the shared-accesses and insert cache-management routines. Coherence management is not triggered by traps. Cache-state is checked by an additional user-level routine (denoted by R),

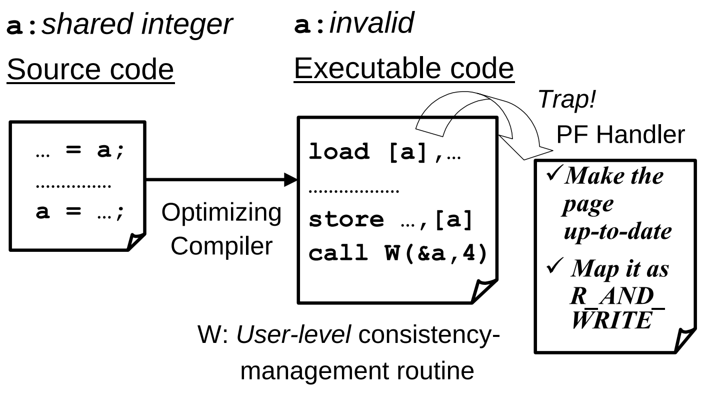


Figure 2.6: ADSM execution model.

and consistency management is executed by an additional user-level routine (denoted by W). The optimizing compiler detects the redundancy of the second occurrence of R(&a,4) and eliminates the second occurrence statically. These detection mechanism and elimination mechanism used in the compiler are described in the following chapter.

2.6 Summary

The UDSM scheme is segment-based and the ADSM scheme is page-based. In both schemes, the optimizing compiler feeds the source program information back to the run-time system. UDSM provides a programmer/compiler with opportunities to optimize both shared-read accesses and shared-write accesses, whereas ADSM offers a programmer/compiler chances to optimize shared-write accesses. Therefore, it depends on an optimizing compiler whether or not ADSM and UDSM produce high performance.

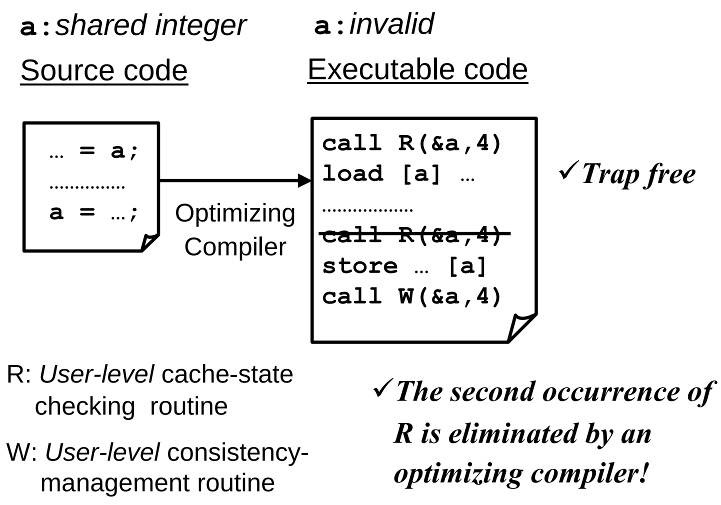


Figure 2.7: UDSM execution model.

Chapter 3

Compiler Optimization for Software Cache

This chapter describes the compiler optimization for Software DSM. The purpose of this optimization is to generate efficient codes reducing both communication overheads and instruction overheads for (software-)cache-coherence management. To achieve this purpose, we have proposed the compiler optimization techniques for explicitly parallel shared-memory programs. To verify our proposed techniques, we have developed the optimizing compiler, called a “Remote Communication Optimizer” (RCOP) [65, 69, 66, 41, 63, 67, 70].

Section 3.1 is the overview of our compiler optimization. Section 3.2 describes the graph terminology useful when explaining compiling techniques. Section 3.3 describes the user-interface of the optimizing compiler, and Section 3.4 describes the scalar dataflow analysis required for successive optimization passes. Next, Section 3.5 describes the detection mechanism of shared-accesses. Finally, Section 3.6 describes the shared-read optimization methods using a interprocedural redundancy elimination framework. It also introduces the concept of a shared-access set and explains optimization using shared-access sets. Section 3.7 describes the shared-write optimization methods by comparing them with the shared-read optimization methods.

3.1 Overview of Optimizing Techniques

The cache-coherence-management routines incur instruction overheads and communication overheads. Using the relaxed coherence mechanism, however, these overheads are reduced at compile-time by exploiting the application's semantics (such as loops and procedure-calls). This is done by the optimizing compiler.

It is required for the optimizing compiler to perform the following operations to reduce these overheads:

- It detects shared-memory accesses that need cache management routines precisely,
- It eliminates redundant cache-management routines, and
- It merges fine-grained cache-management routines into a middle-grained/coarse-grained one.

The followings are required to reduce achieve these operations:

- Parallel constructs recognition

  Our approach deals with explicitly parallel shared-memory programs to accept wider class of applications. Programmers explicitly specify parallel constructs such as shared-memory allocation, task creation and synchronization. Thus, recognition of parallel constructs is required to execute the program in parallel correctly and efficiently.

  Taking program portability and the ease of parallel programming into account, our approach introduces the extension of sequential programming model with directives or macros as explicitly parallel shared-memory programming model.

- Precise points-to information

  This is required for shared-access detection. It is also required when the cache-management routines are moved. The reason for this is that when a statement may modify the parameters of the cache-management routine, this routine cannot be moved across the statement. This information is obtained by interprocedural points-to analysis [28, 82, 84]. This analysis is context-sensitive and computes all the side-effects of the procedure-call.

- Efficient framework for redundancy elimination

  This is required because a control flow graph (CFG) with many cycles (i.e. loops) must be handled when eliminating redundant cache-management routines [21, 64]. An interval analysis approach [15] is used to handle the hierarchical CFG, and this approach requires the interval (i.e., loop) summary to be calculated efficiently. A shared-access set is represented by systems of linear inequalities and used to help calculate the interval summary of the cache-management routines easily and precisely. Shared-access sets are also used in the following optimization techniques.

  - Fusion
  - Coalescing
  - Redundant Index Elimination
  - Fission

  Shared-access sets across procedures are computed efficiently by using the results of points-to analysis.

Our proposed approach has been implemented fully in RCOP. Figure 3.1 shows the overall compilation process. The input program is written in C extended by PARMACS. The RCOP directly analyzes the source program and creates the directed Control Flow Graph (CFG) for each procedure in the program. The RCOP puts each CFG into static single assignment (SSA) form [24].

The RCOP performs an interprocedural points-to analysis to detect shared accesses. After the points-to analysis, the RCOP inserts a cache-management routines for each shared-access. Furthermore, the RCOP performs interprocedural shared-access set calculation to eliminate and merge cache-management routines. The RCOP

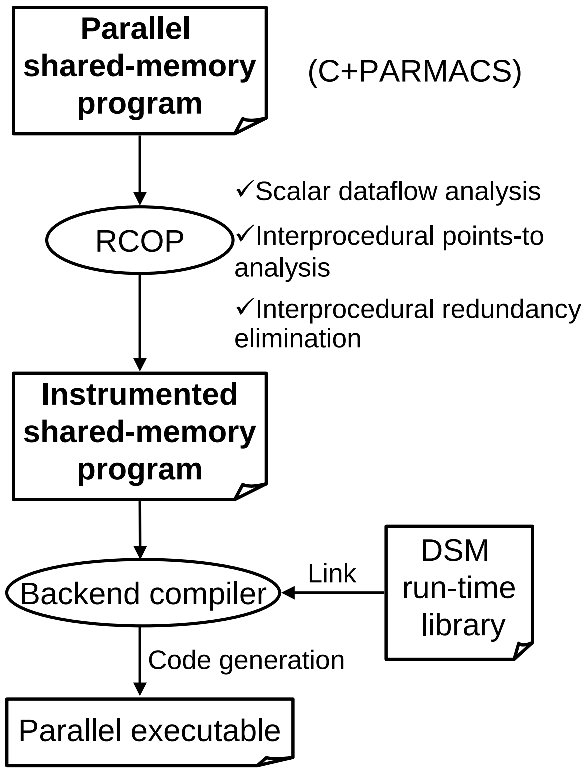
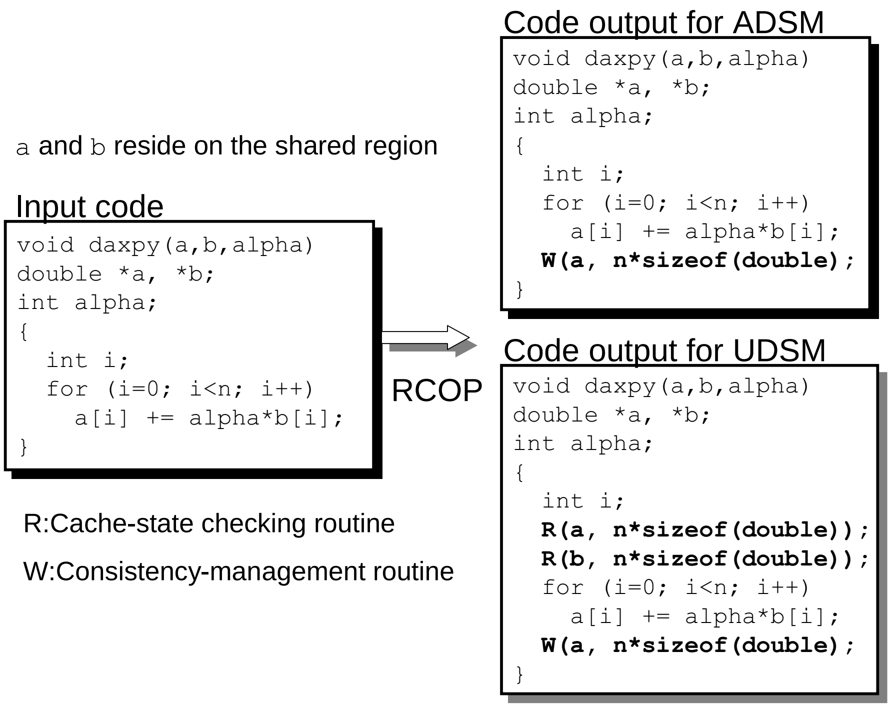

Parallel shared-memory program
(C+PARMACS)
RCOP
✓Scalar dataflow analysis
✓Interprocedural points-to analysis
✓Interprocedural redundancy elimination
Instrumented shared-memory program
Backend compiler
Link
DSM run-time library
Code generation
Parallel executable

Figure 3.1: Overall compilation process.

translates the input program into an instrumented C program containing explicit user-level cache-management routines. The output C program is compiled by a backend compiler, then linked with the run-time library for user-level cache-management to generate executable code.

A consistency-management routine informs the run-time system that a shared-write occurred in a contiguous shared region. The parameters for this routine are the initial address and the size of the region. The routine implicitly requires the written contents and is inserted after the corresponding store instruction. Cache-state checking is executed as a user-level routine (i.e., a cache-state-checking routine) whose parameters are the initial address and the size of the read region. The routine


a and b reside on the shared region
Input code
RCOP
Code output for ADSM
Code output for UDSM
R:Cache-state checking routine
W:Consistency-management routine

Input code

```
void daxpy(a,b,alpha)
double *a, *b;
int alpha;
{
  int i;
  for (i=0; i<n; i++)
    a[i] += alpha*b[i];
}
```

Code output for ADSM

```
void daxpy(a,b,alpha)
double *a, *b;
int alpha;
{
  int i;
  for (i=0; i<n; i++)
    a[i] += alpha*b[i];
  W(a, n*sizeof(double);
}
```

Code output for UDSM

```
void daxpy(a,b,alpha)
double *a, *b;
int alpha;
{
  int i;
  R(a, n*sizeof(double));
  R(b, n*sizeof(double));
  for (i=0; i<n; i++)
    a[i] += alpha*b[i];
  W(a, n*sizeof(double);
}
```

Figure 3.2: Example of generated code.

is inserted as a macro before the corresponding load instruction.

Figure 3.2 shows an example of the code generated for each scheme. "**W**" indicates a routine for managing consistency, and "**R**" indicates a routine for checking a cache-state.

3.2 Graph Terminology

This section introduces graph terminology useful in explaining compiling techniques. A flow multigraph $G = (V, E, r)$ is a finite set V of nodes, a finite multiset E of edges, and a starting node $r \in V$. An edge is an ordered pair (p, q) of nodes: p is the source of the edge, and q is the target of the edge. When edge $(p, q) \in E$, p is a predecessor of q and q is a successor of p. A sequence of edges $(v_1, v_2),\ (v_2, v_3)\ , \dots\ ,\ (v_{n-1}, v_n) \in E$ is a *path from* v_1 *to* v_n. A path is denoted by listing the nodes it contains, like this:

$\langle v_1, v_2, v_3, \ldots, v_{n-1}, v_n\rangle$. If there is a path $\langle v_i, \ldots, v_j\rangle$, it is said that v_i reaches v_j. A path $\langle v_1, \ldots, v_n\rangle$ and $v_1 = v_n$ is called a cycle.

A region R of G is a multigraph whose nodes are in G and such that an edge (m,n) of G is in R if and only if m and n are both in R. If there is an edge (m, n) of G where $m \notin R$ and $n \in R$, the node n is called an entry node of R. A path is internal to a region if all edges of the path belong to the region. A region is strongly connected if every node in the region reaches to every other node in the region.

A depth-first spanning tree (T) of G is generated by a depth-first traversal of G. An edge $e = (m, n)$ is classified as follows:

$$\begin{cases} \text{a tree edge} & \text{if } e \in T \\ \text{a forward edge} & \text{if } e \notin T \text{ and } n \text{ is a descendant of } m \in T \\ \text{a back edge} & \text{if } m \text{ is a descendant of } n \text{ in } T \\ \text{a cross edge} & \text{the other cases} \end{cases}$$

If (l, h) is a back edge, h is a header node and l is a latch node. The interval order of the nodes of G is the order in which they are visited by a reverse postorder traversal. For the j-th node (w) in interval order, $NUMBER(w)$ is defined as j. For all the tree, forward and cross edges (v, w), $NUMBER(v) < NUMBER(w)$ is $true$. If G is acyclic, then interval order is a topological ordering. A path whose sequence of nodes is in interval order is a forward path. A path whose sequence of nodes is in reverse interval order is a backward path.

A Control Flow Graph is a flow graph $G = (V, E, r)$ which represents the procedure structure. A node in V represents the statement in the procedure. An edge in E represents possible transfer of control between such statements.

A Call Graph is a flow graph $G = (V, E, r)$ where each procedure is represented by a simple node and each call site is represented by a unique edge. The starting node r is **`main`**, and the edge (p, q) represents a call site in p that invokes q.

3.3 Application Program Interface

Our approach deals with explicitly parallel shared-memory programs based on the Lazy Release Consistency (LRC) [46] model. In this thesis, the input program is written in C extended by PARMACS [13], a parallel macro construction that provides shared-memory allocation, task creation, and synchronization. It should be noted that our proposed optimization techniques are still valid as long as the parallel constructs are recognized by the compiler. The program follows a conventional shared-memory style, using processes to express parallelism, and using locks and barriers to synchronize. Typically the master process initializes the DSM system by a **`MAIN_INITENV`** operation, allocates and initializes shared memory by a **`G_MALLOC`** operation like **`malloc`**, and starts the slave process on each remote processor by a **`CREATE`** operation like **`fork`**. After initialization is done, each process (not only slaves but also the master) performs a portion of the task, communicating through synchronization operations (**`LOCK`**, **`UNLOCK`**, and **`BARRIER`**). Figure 3.3 shows an program that fills an array in parallel.

Programmers explicitly specify the parallel constructs to execute the program in parallel. Therefore, the programmers are responsible for ensuring that the applications using the parallel constructions run correctly. This indicates that the compilers are not required to solve the problems that result in incorrect execution, such as checks for dependencies, conflicts and race conditions. The optimizing compiler analyzes only the sequential behavior of each process and trace the memory-accesses of an individual process.

3.4 Scalar Dataflow Analysis

Static single assignment form

Our representation uses the SSA form of a procedure because this form is the key to efficient execution of interprocedural points-to analysis. In SSA form, each variable is assigned only at one point in the procedure. To convert a procedure into SSA form, we must do two things:

- Insert ϕ-functions at join nodes in the CFG.

  A ϕ-function for a variable v merges the values of v from distinct incoming control flow paths.

```
    int *A;
    int N = 1000;
    int P = 16;
    main (int argc, char **argv)
    {
        int i;
        char c;
        while ((c = getopt(argc, argv, "N:P:")) != -1) {
            switch(c) {
            case ’N’:
                N = atoi(optarg);
                break;
            case ’P’:
                P = atoi(optarg);
                break;
            }
        }
        MAIN_INITENV();
        A = G_MALLOC(sizeof (sizeof int) * N);

        for (i = 1; i < P; i++)
            CREATE(ZeroVector);

        ZeroVector ();

        WAIT_FOR_END(P - 1);
    }

    void ZeroVector ()
    {
        int j, start, end;
        int id = GET_PID();

        start = id * (N / P);
        end = (id + 1) * (N / P);

        for (j = start; j < end; j++)
            A[j] = 0;
    }
```

Figure 3.3: Sample program written in C extended by PARMACS.

- Rename the variables v so that each assignment has a unique destination variable.

The dominance-frontier approach is used [24]. Intuitively, a ϕ-function for v is placed at the first CFG node n where two distinct definitions of v reach. The ϕ-function itself becomes a new definition of v, so the algorithm iterates. Once ϕ-functions are inserted, each variable generation is renamed to a unique name (using subscripts), and each use of a variable is replaced by the unique reaching SSA definition. An example of a loop and its CFG in SSA form are shown in Figure 3.4.

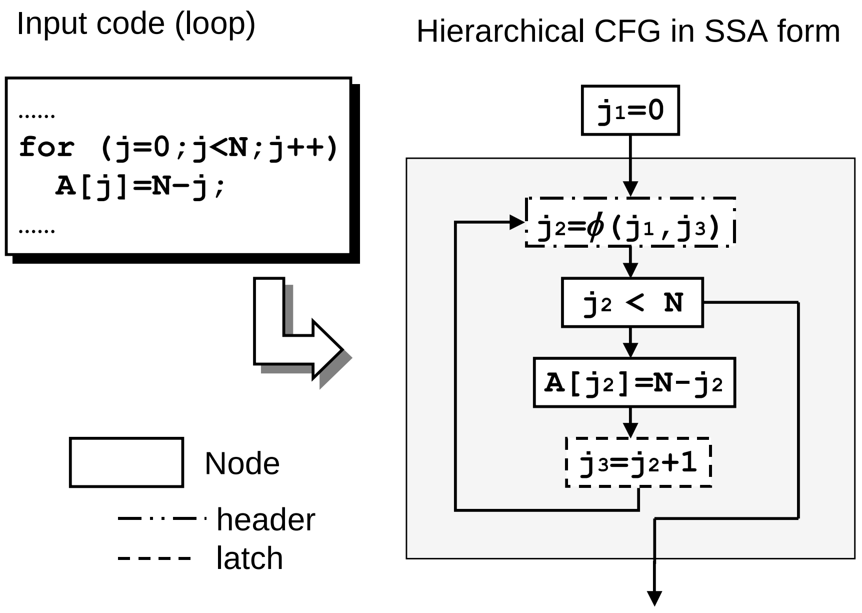


Figure 3.4: Example loop with SSA form.

Induction Variables and Continuous Variables

The following interprocedural shared-access set calculation requires the detection of induction variables and of monotonically increasing or decreasing variables. Induction variables are detected by calculating the following set of expressions $IV(X)$ for each statement X in the program:

$$IV(X) = \text{Assign}(var, sym, \prod_{s \in \text{succ}(X)} IV(s))$$

$\text{Assign}(var, sym, IV)$ represents side-effects of an assignment. When X is an assignment "$var := sym$", it returns a set of expressions:

1. IV with every occurrence of right-hand-side var replaced by sym
2. $var := sym$ (The location var must be loop-invariant)

When X has multiple successor statements s, only expressions that are contained in all $IV(s)$ are propagated to $IV(X)$. After calculating the IV for all statements in the loop, a variable is a basic induction variable if $IV(h)$ (h: the header node of the loop) contains an expression $var := var + c$ and c is loop-invariant [34].

Variables that are incremented or decremented conditionally cannot be recognized as induction variables, but they often show useful properties. They are referred to here as continuous variables, and are defined by the following conditions:

- A loop-invariant location is conditionally increased or decreased by a loop-invariant value (an example is shown in Figure 3.5), or
- A loop-dependent location is increased or decreased by a loop-invariant value (an example is shown in Figure 3.6).

A variable for which the former condition holds is called a continuous scalar variable, and one for which the latter condition holds is called a continuous array variable.

Continuous variables are detected by calculating the following set of expressions $CV(X)$ for each statement X in the program:

$$CV(X) = \text{Assign}(var, sym, \text{Reduce}(=, \sum_{s \in \text{SUCC}(X)} CV(s)))$$

The location var can be loop-dependent. The operator "Reduce" returns expressions that are contained in CV of some successors s and have the same increment (or decrement) value for the same var.

```
for (k = 0, i = 0; i < n; i++) {
    if (A[i] > 0)
        B[k++]=A[i];
}
```

Figure 3.5: Example of continuous scalar variable: **k**.

```
for (i = 0; i < n; i++) {
    j = f(i);
    B[A[j]++] = g(i,j);
}
```

Figure 3.6: Example of continuous array variable: **A[j]**.

3.5 Precise Shared-Accesses Detection

All the shared-memory accesses in a given shared-memory program must be identified if the program is to be executed correctly when cache-coherence management mechanism is explicitly implemented by the compiler-inserted codes. Because the input programs are written in C, a shared address can be contained in a pointer variable and can be passed across procedure calls. When a write through a pointer exists, it may access shared data. When the indirect access always invokes the cache-coherence management mechanism, however, the overheads become large.

Of course, there is an approach to instrument memory accesses to non-private data[1]. This approach analyzes the application binary and inserts cache-management routines for loads and stores whose base registers do not use the stack pointer (SP) or global pointer (GP) register. This binary-instrumentation approach is easy to implement, but too conservative. The cache-management routines are also inserted for memory-accesses to unshared data allocated by `malloc`. This approach may still cause large run-time overheads for cache-management routines.

[1]Private data is all stack and static data.

To detect accesses to shared data, interprocedural points-to analysis [28, 82, 84] is used for the following two reasons.

- Successive optimization passes can move code by using pointer information.

  Because the optimizing compiler handles C programs, optimization using code motion requires precise points-to information.

- Precise shared-pointer information prevents false cache-management routines.

  The more precise the information, the lower the cost of the shared-set calculation and the smaller the run-time overheads for cache-management routines.

3.5.1 Interprocedural Points-to Analysis

This analysis calculates the symbolic locations to which variables may point. Variables and heap locations are represented as a location set (a tuple of a symbolic base address, an offset, and a stride). The points-to function maps each location to the set of locations. For example:

```
double *x, f[10];
x = &f[i];
```

At the end of the above code segment, **`x`** may contain a pointer to **`f[i]`**. Thus, the points-to function is

$$\{(x, 0, 0)\} \rightarrow \{(f, 0, 8)\}.$$

The compiler calculates the points-to functions interprocedurally by using a depth-first traversal of the Call Graph. Within each procedure, iterative dataflow analysis is used to compute points-to functions. The analysis iterates over the statements in the procedure, evaluating each statement and updating the points-to functions. When dereference expressions are evaluated, the pointer values are identified by searching the dominator tree to find the most recent assignment to the corresponding location. Figure 3.7 illustrates this process. The assignment at node **`n1`** defines points-to function $(p_1, 0, 0) \rightarrow (x, 0, 0)$, and the assignment at node **`n3`** defines points-to function

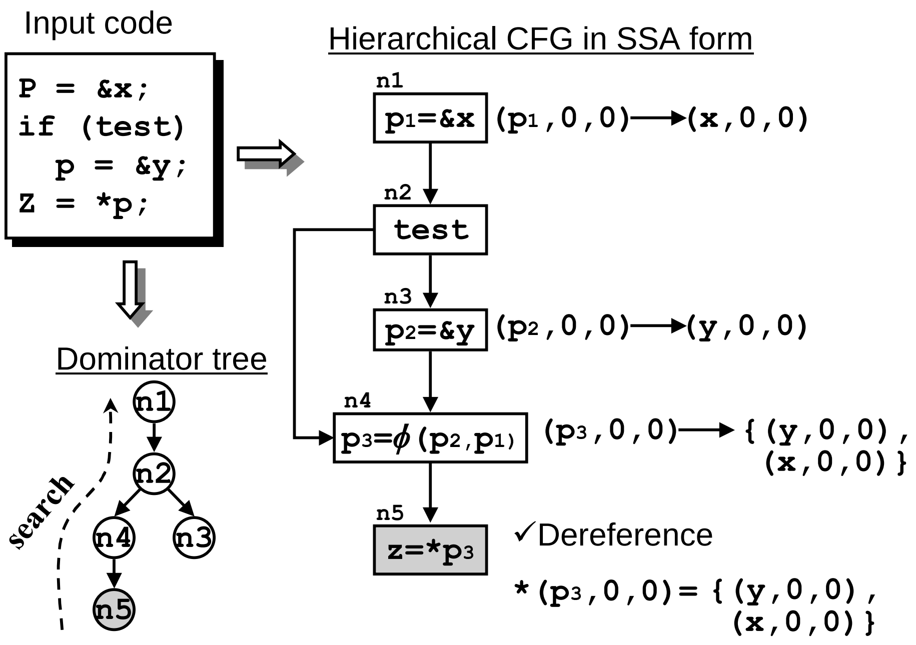


Figure 3.7: Example of dereference-expression evaluation.

$(p_2, 0, 0) \to (y, 0, 0)$. The ϕ-function at node **n4** combines the pointer values from all the incoming edges for the variable (**p**). When the dereference expression ***p** at node **n5** is evaluated, the dominator tree is searched to find the most recent assignment to the location (p_3). Thus, the result of dereference at **n4** is $\{(x, 0, 0), (y, 0, 0)\}$. If in the process of looking up the value of a location, the entry node can be reached without a dominating assignment being found, [2] the location has whatever initial value it had at the entry to the procedure. That initial value is represented by nonlocal blocks. Each nonlocal block represents an equivalence class of the caller's locations and is created lazily when the pointers are dereferenced. For example, consider the following code fragment:

```
int f (int *p, int *q, int *r);
...
f(&x, &x, &y);
```

[2]This happens when the parameters of the procedure are first dereferenced or global variables are first accessed.

Nonlocal blocks are not created until the parameters of **`f`** (i.e., **`p, q ,r`**) are dereferenced at the procedure body:

$$(p, 0, 0) \rightarrow (b\_0, 0, 0)$$

$$(q, 0, 0) \rightarrow (b\_0, 0, 0)$$

$$(r, 0, 0) \rightarrow (b\_1, 0, 0)$$

where b_0 is bound by$\{(x, 0, 0)\}$and b_1 is bound by$\{(y, 0, 0)\}$

For call statements, it is necessary to first find the potential callees and then evaluate the side-effect of the call to the points-to relations, called the partial transfer function (PTF). For each potential callee, the first thing to do is to find the PTF that describes the side-effect of the callee in the current calling context. An existing PTF is checked whether or not it is applied. If the pattern of aliases among the parameters of the procedure in the current calling context is the same as that recorded in the PTF, it is reused. If an existing PTF cannot be used, a new PTF is created by analysing the callee procedure. Once the applicable PTF is found, all the modified points-to information recorded by the PTF (final points-to functions) is translated back to the calling context and used to update points-to functions.

For example, consider the codes shown in the Figure 3.8. The analysis begins by iterating over the statements in the **`main`** procedure. After evaluating three assignments, the analysis comes to the procedure call **`f`** at **`C1`**. This is the first time to evaluate the procedure **`f`**, and a new PTF for **`f`** is created and the **`f`** is analyzed and final points-to functions shown in the Figure 3.9 are produced. These points-to functions are translated back to the caller's name space and the points-to function after the C1 is updated like this: $(x\_0, 0, 0) \rightarrow (y, 0, 0)$, $(y\_0, 0, 0) \rightarrow (x, 0, 0)$. This process is shown in the Figure 3.10. With regard to the call at **`C2`**, the pattern of aliases among the parameters at **`C2`** is the same as at **`C1`**. Therefore, the same PTF is reused. But the bind for the nonlocal block at the call `C2` is different from that at the call **`C1`** (shown in Figure 3.10). The result of applying the PTF at **`C2`** is updated

```
void IndirectSwap (int **, int **);

void main () {
    int x, y, z;
    int *x0, *y0, *z0;

    int test1, test2;
    x0 = &x; y0 = &y; z0 = &z;
    if (test1)
C1:     IndirectSwap(&x0, &y0);
    else if (test2)
C2:     IndirectSwap(&y0, &z0);
}

void IndirectSwap (int **p, int **q)
{
    int *tmp;
    tmp = *p;
    *p  = *q;
    *q  = tmp;
}
```

Figure 3.8: Sample program with two calling contexts.

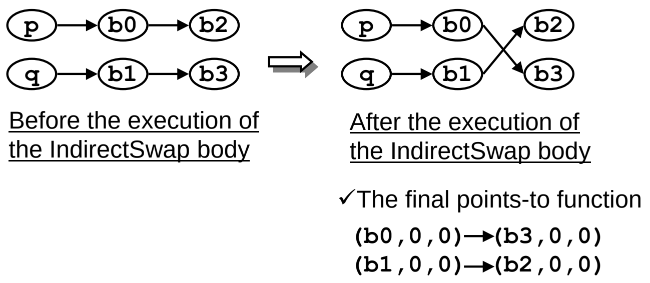

p
b0
b2
q
b1
b3
Before the execution of the IndirectSwap body
After the execution of the IndirectSwap body
✓The final points-to function
(b0,0,0)→(b3,0,0)
(b1,0,0)→(b2,0,0)

Figure 3.9: Graphical representations of points-to functions for **`IndirectSwap`** in Figure 3.8.

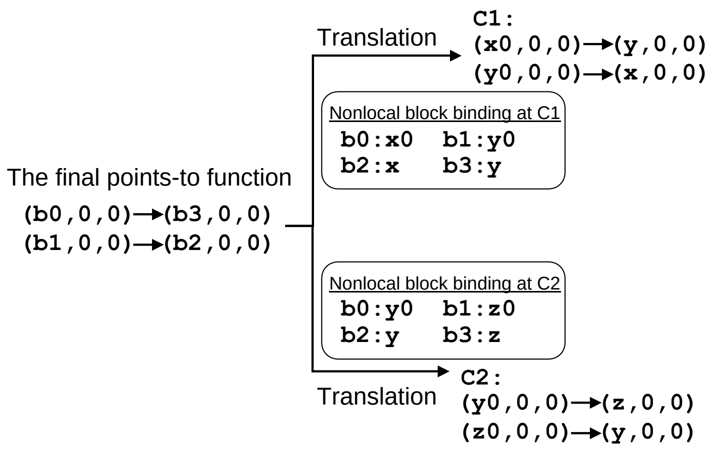


Figure 3.10: Translation of final points-to functions into calling contexts in Figure 3.8.

like this: $(y\_0, 0, 0) \rightarrow (z, 0, 0)$, $(z\_0, 0, 0) \rightarrow (y, 0, 0)$. After the procedure calls are evaluated, the join node at the end of the **`main`** is evaluated. The analysis combines all the possible points-to values for each location. The result is as follows:

$$(x\_0, 0, 0) \rightarrow \{(x, 0, 0),\ (y, 0, 0)\}$$

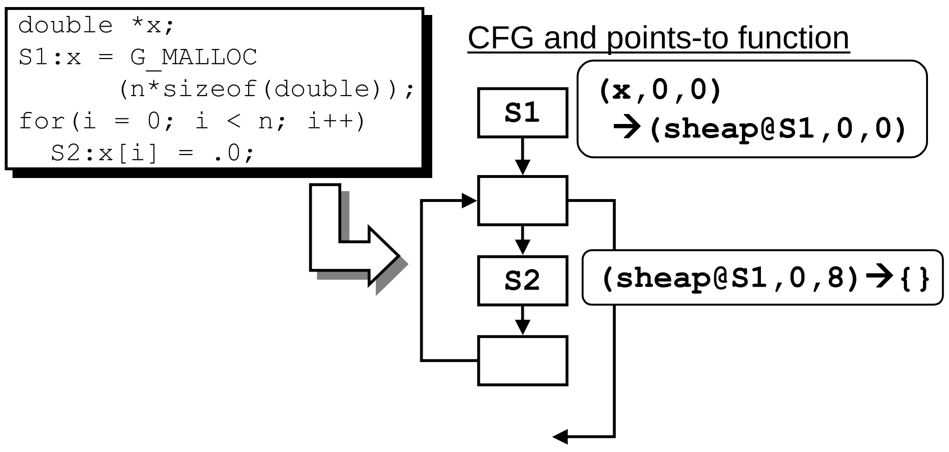
$$(y\_0, 0, 0) \rightarrow \{(x, 0, 0),\ (y, 0, 0),\ (z, 0, 0)\}$$

$$(z\_0, 0, 0) \rightarrow \{(z, 0, 0),\ (y, 0, 0)\}$$

Details of the algorithm and the implementation of the interprocedural points-to analysis are described in Wilson's doctor thesis [82].

3.5.2 Detection Methods

Shared memory is allocated dynamically by the primitive `G_MALLOC`. The optimizing compiler track the return values of the primitive `G_MALLOC`, and must insert cache-management routines. Consider the example shown in Figure 3.11. The



```
double *x;
S1:x = G_MALLOC
       (n*sizeof(double));
for(i = 0; i < n; i++)
  S2:x[i] = .0;
```

Figure 3.11: Example of shared-access detection.

points-to analysis defines the points-to function at the assignment statement **S1** as $(x, 0, 0) \rightarrow (sheap@S1, 0, 8)$. "$sheap@S1$" means the shared heap (i.e., shared memory) allocated by `G_MALLOC` at **S1**. The points-to function at the assignment statement **S2** becomes $(sheap@S1, 0, 8) \rightarrow \{\}$. Each statement is searched for memory-accesses to data whose location set is denoted as $(sheap@*, *, *)$. In this example, the destination location of the assignment at **S2** (i.e., the location of **`x[i]`**) is denoted as $(sheap@S1, 0, 4)$. That is, the write operation at **S2** is a shared one.

For each procedure, each statement is searched for memory-accesses to shared data by using the points-to information in PTFs. When there are multiple PTFs for a procedure, it is possible that the location set for some data is $(sheap@C1, 0, 0)$ in one PTF but the location set for the same data is $(x, 0, 0)$ (i.e., local) in another PTF. Namely, the data is allocated on shared region in one calling context and the data is allocated on private region in another calling context. In this case, the access to the data is considered a shared one and the cache-management routine for the access is inserted. This approach works correctly because at run-time the cache-management routines check whether the data is in the shared-memory range at run-time. If the data is not in the shared-memory range, the cache-management routines do nothing at run-time.

Shared-access detection mechanisms described in this section have been implemented fully in the RCOP. The RCOP inserts cache-coherence management routines according to the target DSM systems:

1. ADSM/SSS–CORE system

   Consistency-management routines are inserted after write-accesses that may use shared-address values (i.e., return values of **G_MALLOC**).

2. UDSM/SSS–CORE system

   Consistency-management routines are inserted as in the ADSM/SSS–CORE system, and checking routines are inserted before read-accesses [3] that may use shared-address values (i.e., return values of **G_MALLOC**).

The RCOP output codes for the code used in the previous example are as follows (They are further optimized in successive optimization passes). The write-access to **x[i]** are considered to include the read-access **x[i]**. Therefore, the corresponding checking routine **R((unsigned)&x[i],sizeof(double))** is inserted.

ADSM Output Codes

```
double *x;
S1:x = G_MALLOC
       (n*sizeof(double));
for(i = 0; i < n; i++) {
  S2:x[i] = .0;
  W((unsigned)&x[i],
     sizeof(double));
}
```

UDSM Output Codes

```
double *x;
S1:x = G_MALLOC
       (n*sizeof(double));
for(i = 0; i < n; i++) {
  R((unsigned)&x[i],
     sizeof(double));
  S2:x[i] = .0;
  W((unsigned)&x[i],
     sizeof(double));
}
```

Figure 3.12: Output codes for first pass.

[3]Write accesses are considered to include read accesses because the cache must be fetched before it is written to.

3.6 Shared-Read Optimization Methods

This section describes shared-read optimization methods using the LRC model. The summary of shared-accesses, called the shared-access set is calculated interprocedurally by using interval analysis [15] to solve redundancy elimination dataflow problems.

3.6.1 Remove Redundant Checking Routines

With the LRC model, consistency is enforced only at an acquire (see Figure 2.3 in Chapter 2). Any shared data accessed after the acquire a_1 and before the following acquire a_2 can be fetched immediately after a_1 because the fetched data is never invalidated before it is actually accessed. The checking routines can be fetched anywhere after a_1 and before the corresponding load instruction. Cache-miss latency is reduced by issuing check codes as far before the corresponding load instruction as possible. This flexibility makes it easy to remove redundant check codes.

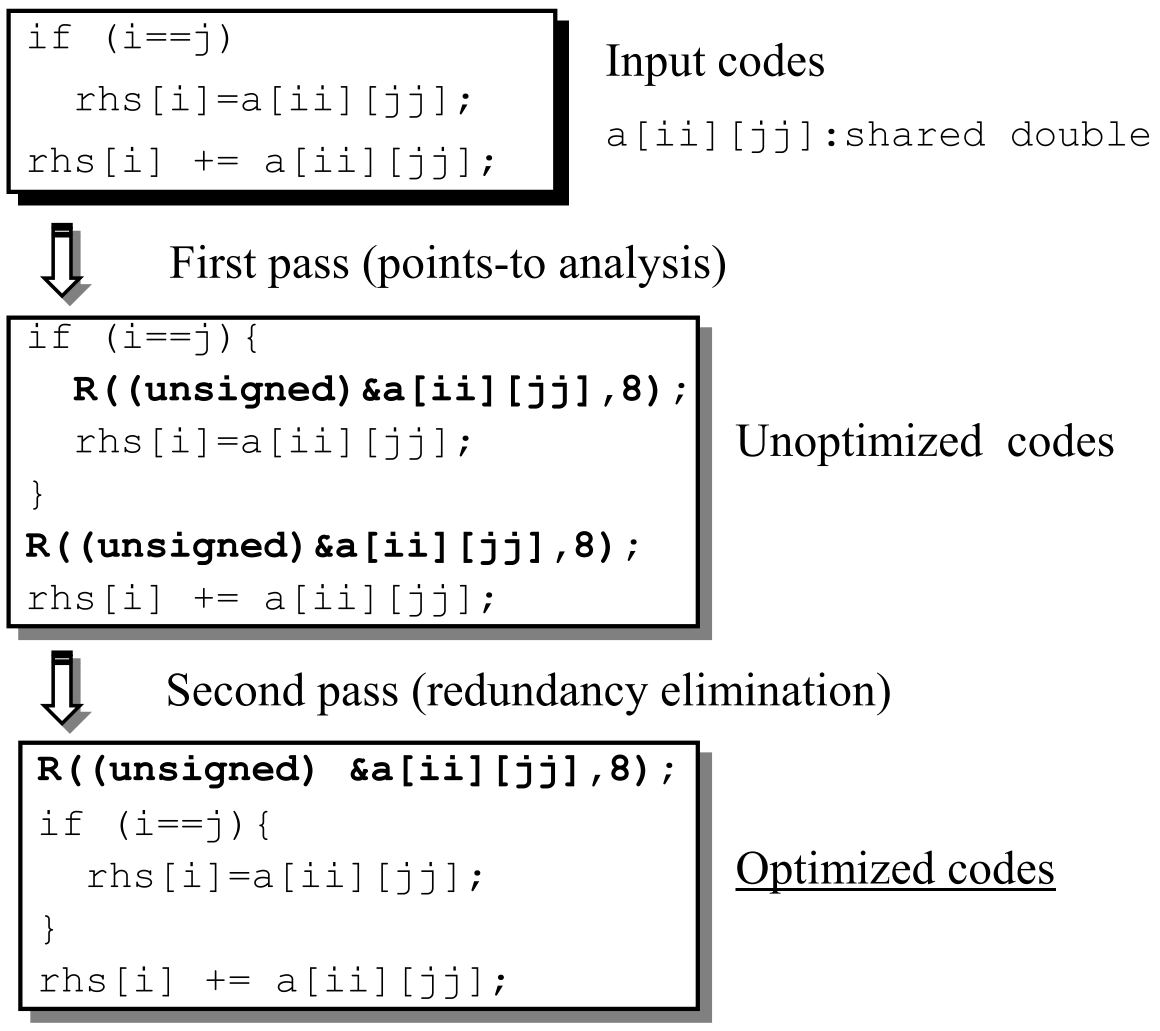
For example, consider the code segment shown in Figure 3.13. Suppose that **`a[ii][jj]`** is shared and its type is **`double`**. Checking routines can be inserted immediately in front of both assignments[4], but the checking routine within the conditional becomes redundant if we place the second checking routine before the conditional. This redundant code can then be eliminated.

Redundancy Elimination Algorithm

This optimization can be formalized as the redundancy elimination [21, 64] for checking routines. Here, a statement in a procedure is represented by i, and i is considered to be a node in a CFG of the procedure. For simplicity, we fix one shared read whose address is a and size is s. From the results of the points-to analysis, the following logical constants are obtained for each i:

COMP(i) statement i issues a shared read whose address is a and whose size is s

[4]A checking routine is denoted by **R**.


Input codes
a[ii][jj]:shared double
First pass (points-to analysis)
Unoptimized codes
Second pass (redundancy elimination)
Optimized codes

```
if (i==j)
  rhs[i]=a[ii][jj];
rhs[i] += a[ii][jj];
```

```
if (i==j){
  R((unsigned)&a[ii][jj],8);
  rhs[i]=a[ii][jj];
}
R((unsigned)&a[ii][jj],8);
rhs[i] += a[ii][jj];
```

```
R((unsigned) &a[ii][jj],8);
if (i==j){
  rhs[i]=a[ii][jj];
}
rhs[i] += a[ii][jj];
```

Figure 3.13: Example of code generation (1).

TRANS(i) statement i propagates information about the shared read above and below

TRANS(i) is false when i is a synchronization primitive or the statement modifies the parameters of the checking routine. The following logical dataflow variables are calculated from these constants:

Availability The shared read is issued in all paths that precede i.

Anticipatability The shared read is issued in all paths that succeed i.

The number of checking routines is minimized by placing them only where

- a shared read is anticipatable,
- a shared read is not anticipatable in one of the preceding paths or the transparency of one is false in one of the preceding paths, and
- a shared read is not available.

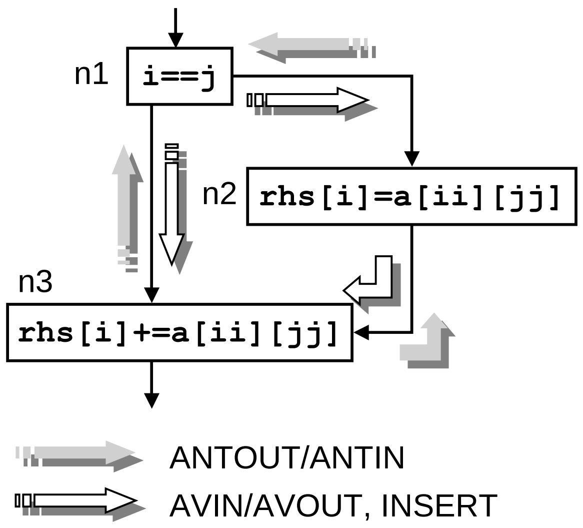
Anticipatability before and after execution of statement i is represented here as ANTIN(i) and ANTOUT(i). Similarly, availability is represented as AVIN(i) and AVOUT(i). INSERT(i) is a variable meaning that the checking routine is actually placed before i. Variables are calculated by using the dataflow equations shown in Figure 3.14, where pred(i) and succ(i) represent sets of predecessors and successors of i. The above constraints about optimal placement simplify the dataflow equations to a combination of uni-directional equations: the calculations of ANTIN and ANTOUT are backward dataflow problems and the calculations of AVIN, AVOUT and INSERT are forward dataflow problems. Each problem is essentially equal to global common subexpression elimination problem, and it is monotone dataflow problem.

Figure 3.15 shows an example of what is obtained when the dataflow equations in Figure 3.14 are applied to the code segment in Figure 3.13. The values of logical dataflow variables and constants are listed in the table in Figure 3.15. When the checking routine is fixed as **`R((unsigned)&a[ii][jj],sizeof(double))`**, the local dataflow properties COMP and TRANS are obtained for each node. All the

$$
\begin{aligned}
\mathrm{ANTOUT}(i) &= \prod_{s \in \mathrm{succ}(i)} \mathrm{ANTIN}(s) \\
\mathrm{ANTIN}(i) &= \mathrm{COMP}(i) + \mathrm{TRANS}(i) \cdot \mathrm{ANTOUT}(i) \\
\mathrm{AVIN}(i) &= \prod_{p \in \mathrm{pred}(i)} \mathrm{AVOUT}(p) \\
\mathrm{AVOUT}(i) &= (\mathrm{COMP}(i) + \mathrm{AVIN}(i)) \cdot \mathrm{TRANS}(i) \\
\mathrm{INSERT}(i) &= \mathrm{ANTIN}(i) \cdot \neg\Big(\prod_{p \in \mathrm{pred}(i)} (\mathrm{TRANS}(p) \cdot \mathrm{ANTIN}(p))\Big) \cdot \neg\mathrm{AVIN}(i) \qquad (3.1)
\end{aligned}
$$

Figure 3.14: Dataflow equations used to remove redundant check routines.

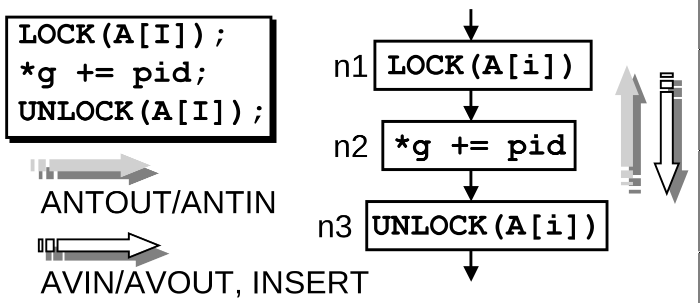
TRANS are *true*. ANTIN, ANTOUT are calculated by a CFG traversal in post order (3→2→1). ANTIN(3) = *true* and ANTIN(2) = *true* yield ANTOUT(1) = *true*. Hence, ANTIN(1) = *true*. AVIN and AVOUT, on the other hand, are calculated by a CFG traversal in reverse post order (i.e., in interval order: 1→2→3). Lastly, INSERT is calculated using the values computed above, and only INSERT(1) is found to be *true*.



Node No. → Property ↓	1	2	3
COMP		T	T
TRANS	T	T	T
ANTIN	T	T	T
ANTOUT	T	T	
AVIN			
AVOUT	T	T	
INSERT	T		

Figure 3.15: Solution of dataflow equations described in Figure 3.14 (1).

The TRANS in $\prod$ term of INSERT equation (eq. 3.1) seems to be redundant, but it is necessary for correct insertion when shared read and synchronization primitive are issued at the same node. The reason is explained here using the example shown



Node No. → Property ↓	1	2	3
COMP	T		T
TRANS		T	
ANTIN	T	T	T
ANTOUT	T	T	
AVIN			
AVOUT			
INSERT	T		T

Figure 3.16: Solution of dataflow equation described in Figure 3.14 (2).

in Figure 3.16. Suppose that **`A[i]`** is shared and its type is **`integer`**. A shared read is issued at both the **`LOCK`** operation and the **`UNLOCK`** operation. The checking routine R((unsigned)&a[i],sizeof(integer)) must be inserted before both synchronization primitives. This is why **`A[i]`** may become invalid after the **`LOCK`** (i.e., acquire) operation and the checking routine must be inserted before using the value **`A[i]`** (the **`UNLOCK`** operation).

Local dataflow properties COMP and TRANS are obtained for each node. TRANS(1) is $false$ and COMP(1) is $true$. TRANS(3) is $false$ and COMP(3) is $true$. ANTIN and ANTOUT are calculated by a CFG traversal in post order, whereas AVIN and AVOUT are calculated by a CFG traversal in interval order. AVOUT(1) and AVOUT(3) become $false$ because of the term TRANS. Hence, all the AVIN are $false$. Lastly, INSERT is calculated using the values computed above, and

TRANS(1) $= false$ is found to make INSERT(2) $= true$ as follows:

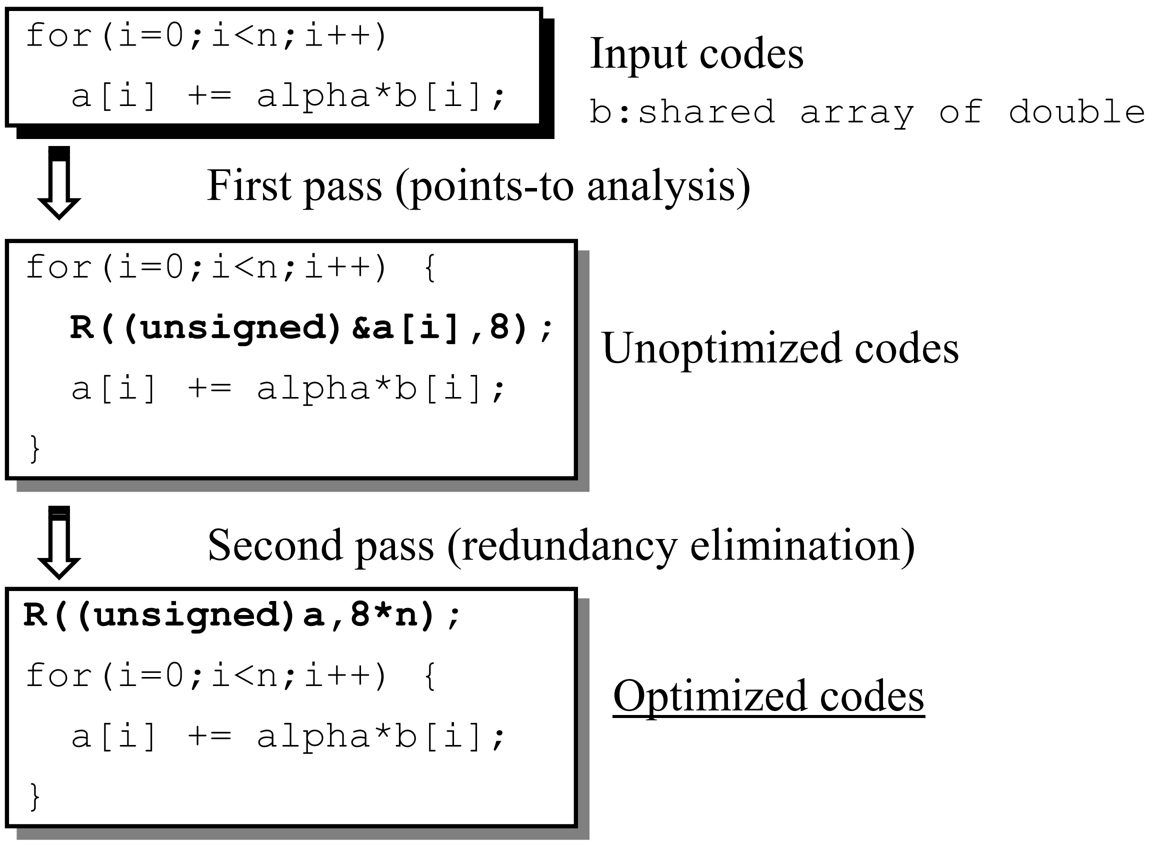
$$\begin{aligned}\text{INSERT}(2) &= \text{ANTIN}(2) \cdot \neg(\text{TRANS}(1) \cdot \text{ANTIN}(1)) \cdot \neg\text{AVIN}(1) \\ &= true \cdot \neg(false \cdot true) \cdot \neg false \\ &= true\end{aligned}$$

These examples are solved by using a single traversal in reverse-topological order and a single traversal in topological order because the CFG is acyclic. In practice, there are many cycles (i.e., loop structures) in the CFG. When there are many cycles in the CFG, The redundancy-elimination dataflow equations must be solved efficiently. The next subsection describes the efficient methods.

3.6.2 Merge Multiple Checking Routines by Using Loop Structures

Up to now, we have fixed one shared read. But in practice, there are many shared reads in the program. Calculating dataflow equations for each shared read would make the compilation time long. From the viewpoint of compilation efficiency, it is appropriate to handle multiple checking routines at the same time. Moreover, handling multiple checking routines results in various optimizations described in this subsection.

An effective way to reduce checking overhead is to batch together the checking routines for multiple shared reads. For instance, consider the code fragments shown in Figure 3.17. Suppose **b** is a shared pointer. It is correct to insert a checking routine into the innermost loop, but by using loop index information, the checking routines can be merged. In this way, loop-index information is important. The optimizing compiler, therefore, associates checking routines with loop-index information. That is, a sequence of checking routines is represented as a shared-access set. This set is a tuple $(f, s, \mathcal{C})$, where f and s are arguments of each checking routine and $\mathcal{C}$ corresponds to a set of inequalities representing the enclosing loops.



```
for(i=0;i<n;i++)
  a[i] += alpha*b[i];
```

Input codes
b:shared array of double
First pass (points-to analysis)

```
for(i=0;i<n;i++) {
  R((unsigned)&a[i],8);
  a[i] += alpha*b[i];
}
```

Unoptimized codes
Second pass (redundancy elimination)

```
R((unsigned)a,8*n);
for(i=0;i<n;i++) {
  a[i] += alpha*b[i];
}
```

Figure 3.17: Example of code generation (2).

Interval Analysis Technique

COMP(i) is considered as the set of shared reads that statement i issues. A dataflow variable, including COMP(i), takes a set of checking routines (i.e., a set of shared-access sets). The logical operations in Figure 3.14 are considered to be set operations. Immediately after the points-to analysis, each shared-access set includes only one check routine. That is, $s =$ `sizeof(X)`$, \mathcal{C} = \emptyset$ where **`X`** is the domain type of f.

This approach, however, encounters difficulties when transparency is computed. This is because transparency cannot be determined before starting the solution of dataflow equations. For instance, there may be a node i that modifies the parameter of the checking routine R_1 but does not modify the parameter of the checking routine R_2. For this reason, the redundancy elimination dataflow equations are redefined using shared-access sets.

$$
\begin{aligned}
\mathrm{ANTOUT}(i) &= \bigcap_{s \in \mathrm{succ}(i)} \mathrm{ANTIN}(s) \\
\mathrm{ANTIN}(i) &= \mathrm{COMP}(i) \bigsqcup \mathrm{TRANS_i}[\,\mathrm{ANTOUT}(i)\,] \\
\mathrm{AVIN}(i) &= \bigcap_{p \in \mathrm{pred}(i)} \mathrm{AVOUT}(p) \\
\mathrm{AVOUT}(i) &= \mathrm{TRANS_i}[\,\mathrm{COMP}(i) \bigsqcup \mathrm{AVIN}(i)\,] \\
\mathrm{INSERT}(i) &= \mathrm{ANTIN}(i) - \big(\bigcap_{p \in \mathrm{pred}(i)} \mathrm{TRANS_p}[\,\mathrm{ANTIN}(p)\,]\big) - \mathrm{AVIN}(i)
\end{aligned}
$$

Figure 3.18: Redundancy elimination dataflow equations using shared-access sets for checking routines.

Logical operations are substituted for set operations. A logical operation $A \cdot \neg B$ is substituted for a difference set operation $A - B$. $A - B$ consists of elements that are in A and not in B. Logical operations $\cdot$ and $\prod$ are substituted for the set operation $\bigcap$ (intersection). Now, transparency is considered as function. $\mathrm{TRANS_i}[X]$ of a node i returns $\emptyset$ if i is a synchronization primitive. Otherwise, it returns the subset of X by eliminating elements of X whose parameters are modified by i. We can define an identity function $\top$ such that $\forall X \; \top[X] = X$, and we can define a zero function $\perp$ such that $\forall X \; \perp[X] = \emptyset$.

We can also replace a logical operation $S_1 + S_2$ with $S_1 \bigsqcup S_2$, the optimized-union operation. The optimized-union operation $S_1 \bigsqcup S_2$ first computes the union S_3 of S_1 and S_2. Then, this operation reduces the union by concatenating its elements if possible. This reduction is called fusion.[5]

Fusion

Condition 3.6.1 (Fusion) When there are elements $(f_1(\boldsymbol{i}), s_1(\boldsymbol{i}), \mathcal{C}_1(\boldsymbol{i}))$, $(f_2(\boldsymbol{j}), s_2(\boldsymbol{j}), \mathcal{C}_2(\boldsymbol{j}))$ in the union S_3 such that

$$\mathcal{C}_1(\boldsymbol{i}) \equiv \mathcal{C}_2(\boldsymbol{j}) \text{ and } f_1(\boldsymbol{i}) < f_2(\boldsymbol{i}) \text{ and } f_2(\boldsymbol{i}) - f_1(\boldsymbol{i}) \le s_1(\boldsymbol{i}),$$

these two elements are combined into one new element:

$$(f_1(\boldsymbol{i}), s_1(\boldsymbol{i}) + s_2(\boldsymbol{i}), \mathcal{C}_1(\boldsymbol{i})).$$

$\mathcal{C}_1(\boldsymbol{i}) \equiv \mathcal{C}_2(\boldsymbol{j})$ indicates that the set represented by $\mathcal{C}_1(\boldsymbol{i})$ is equal to the set represented by $\mathcal{C}_2(\boldsymbol{j})$. This reduction operation means that we can merge checking routines originating in different statements in the program. We represent fusion here by the binary operator "$\circ$".

For example, consider the following code fragments, and compute ANTIN(n1).

```
n1:x_r = x2[2 * j];
n2:x_c = x2[2 * j + 1];
```

Suppose **`x2`** points to a shared address and its domain type is **`double`**. Then COMP($n1$) = $\{R1\}$ and COMP($n2$) = $\{R2\}$, where

$$R1 = (\texttt{(unsigned)\&}x2[2*j], 8, \emptyset), \quad R2 = (\texttt{(unsigned)\&}x2[2*j+1], 8, \emptyset).$$

We find that $\mathrm{TRNAS_{n1}} = \mathrm{TRANS_{n2}} = \top$. Suppose that ANTOUT($n2$) = $\emptyset$.

[5]Its name comes from its similarity to loop transformation.

Then ANTIN(n2) is $\{R_2\}$ because

$$\begin{aligned}\text{ANTIN}(n2) &= \text{COMP}(n2) \bigsqcup \text{TRANS}_{\text{n2}}[\,\text{ANTOUT}(n2)\,]\\ &= \{R_2\} \bigsqcup \top[\,\emptyset\,]\\ &= \{R_2\} \bigsqcup \emptyset\\ &= \{R_2\}.\end{aligned}$$

Therefore, ANTOUT$(n1)$ = ANTIN$(n2)$ = $\{R_2\}$. ANTIN$(n1)$ is computed as follows:

$$\begin{aligned}\text{ANTIN}(n1) &= \text{COMP}(n1) \bigsqcup \text{TRANS}_{\text{n1}}[\,\text{ANTOUT}(n1)\,]\\ &= \{R_1\} \bigsqcup \top[\,\{R2\}\,]\\ &= \{R_1\} \bigsqcup \{R_2\}.\end{aligned}$$

When computing $\{R_1\} \bigsqcup \{R_2\}$, we first create $\{R_1\} \bigcup \{R_2\} = \{R_1, R_2\}$ and we find that

$$\texttt{(unsigned)\&x2[2*j]} < \texttt{(unsigned)\&x2[2*j+1]} \text{ and}$$
$$\texttt{(unsigned)\&x2[2*j+1]} - \texttt{(unsigned)\&x2[2*j]} = 8 \le 8.$$

This indicates that R_1 and R_2 satisfies Condition 3.6.1, and thus can be combined into a new R':

$$R_1 \circ R_2 \to R' = (\texttt{(unsigned)\&}x2[2*j], 16, \emptyset).$$

That is, $\{R_1\} \bigsqcup \{R_2\} = \{R'\}$. Hence, ANTIN(n1) = $\{R'\}$.

Consider the following example. $R1 = (\texttt{(unsigned)\&}x2[2*i], 8, 0 \le i < n)$ and $R2 = (\texttt{(unsigned)\&}x2[2*j+1], 8, 0 \le j < n)$. Let us compute $\{R_1\} \bigsqcup \{R_2\}$.

We find $\{0 \le j < n\} \equiv \{0 \le i < n\}$, i.e., $\mathcal{C}_1(i) \equiv \mathcal{C}_2(j)$. Further, $f_1(i)(= \texttt{(unsigned)\&}x2[2{*}i]) < f_2(i)(= \texttt{(unsigned)\&}x2[2{*}i{+}1])$ and $f_2(i) - f_1(i) = 8 \le s_1(i) = 8$. Therefore we obtain

$$R1 \circ R2 \to (\texttt{(unsigned)\&}x2[2*i], 16, 0 \le i < n).$$

In practice, Condition 3.6.1 is unnecessarily strict: checking routines originating in different statements in the program can be merged more aggressively. All the data in a block is fetched at a cache-miss and is kept coherent as a unit. Therefore, it is important whether or not the multiple shared reads are issued onto data in the same block. Checking routines issued onto neighbor regions can be fused because it is likely that the corresponding data is in the same block. Regions are considered as neighbor when the distance between the regions is smaller than the cache-block size.

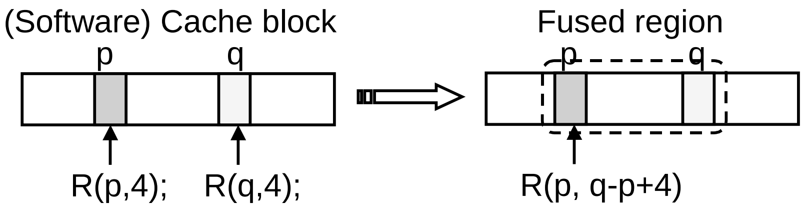


Figure 3.19: Example of fusion operation.

The Condition 3.6.1 is relaxed as follows.

Condition 3.6.2 (Fusion) When there are elements $(f_1(\boldsymbol{i}), s_1(\boldsymbol{i}), \mathcal{C}_1(\boldsymbol{i}))$, $(f_2(\boldsymbol{j}), s_2(\boldsymbol{j}), \mathcal{C}_2(\boldsymbol{j}))$ in the union S_3 such that

$$\mathcal{C}_1 \equiv \mathcal{C}_2 \text{ and } f_1 < f_2 \text{ and}$$

$$f_2 - f_1 \leq s_1 \textit{ or } f_2 - f_1 \leq \text{block\_size} + s_1,$$

these two elements are combined into one new element:

$$(f_1, f_2 - f_1 + s_2, \mathcal{C}_1).$$

Figure 3.19 shows an example of fusion using this relaxed condition.

Even if two shared locations are neighbor, they may not be in the same cache-block. The fused region may cover the multiple caches. The optimizing compiler takes no account of this situation because this situation is detected dynamically and handled properly in the cache-state-checking routine at run-time. It is important to note that the overheads for fused codes are not larger than those for original codes. This is because the fused region does not cover the cache that is not really accessed. Furthermore, the number of cache routines is reduced. In fused codes, state-checks of multiple caches are executed consecutively in one routine. Hence, the locality of references is improved.

There are many conditions that enable fusion. For example,

Condition 3.6.3 (Fusion) When there are elements $(f_1, s_1, \mathcal{C}_1)$, $(f_2, s_2, \mathcal{C}_2)$ in the union S_3 such that
$f_1 = f_2$ and $s_1 = s_2$ and $\mathcal{C}_1 \cup \mathcal{C}_2$ is represented as a set of explicit inequalities, these two elements are combined into one new element:

$$(f_1, s_1, \mathcal{C}_1 \cup \mathcal{C}_2).$$

Explicit inequality for a integer variable v means min $\leq v <$ max. Consider $R_1 = (\texttt{(unsigned)\&}a[n], 8, \emptyset)$ and $R_2 = (\texttt{(unsigned)\&}a[i], 8, 0 \leq i < n)$. Let us compute $\{R_1\} \bigsqcup \{R_2\}$. We consider R_1 as $(\texttt{(unsigned)\&}a[i], 8, i = n)$. Therefore R_1 and R_2 satisfy Condition 3.3 and are combined into R_3.

$$\begin{aligned} R_1 \circ R_2 = R_3 &= (\texttt{(unsigned)\&}a[2*i], 8, 0 \leq i < n \cup i = n) \\ &= (\texttt{(unsigned)\&}a[2*i], 8, 0 \leq i \leq n) \end{aligned}$$

The interval analysis framework [15, 22] is used to calculate the dataflow equations shown in Figure 3.18 efficiently. After COMP is computed locally for each node in the CFG, the terms ANTIN, ANTOUT and TRANS are computed by interval analysis. Then the terms AVIN, AVOUT are computed by interval analysis. Finally, INSERT is computed by using the above values. In interval analysis, the CFG is represented hierarchically with loop structures and the loop summary (i.e., the dataflow effect of the cycle) needs to be computed efficiently.

Interval Terminology

This section introduces interval terminology useful in explaining compiling techniques. For a back edge (m, n) in a flow multigraph G, the nodes and edges belonging to forward paths from n to m form the strongly connected region $STR(m, n)$. The set

$B(h)$ consists of back edges whose target node is h. For each edge (l, h), $STR(l, h)$ is defined. the union of the strongly connected region defined by $B(h)$ is considered to be as an interval region I whose header is h [15]. The set of l is referred to as Latches(I). Assume that G is reducible: that is, that the header h is the only entry node of the interval I [4]. Therefore, h dominates every nodes in the interval I.

Intervals can be nested. An interval I can be a subinterval of interval J. An interval that is not a subinterval of any interval is an outermost interval. An interval that does not contain any subintervals is an innermost interval. The exit edge of an interval I is an edge that is (m, n) such that I contains m but does not contain n. A node m is called as an exit node of I, and the set of all m is Exits(I).

Interval analysis consists of two phases: the elimination phase and the propagation phase. The elimination phase evaluates the dataflow effects of all cycles in a CFG G that originate at header nodes. The propagation phase then solves for each node of G by traversing G in a single course. [6]

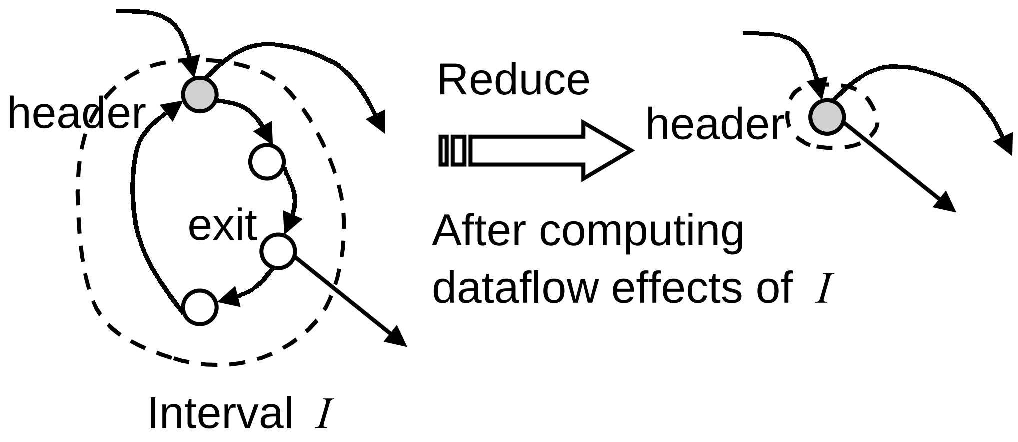


Figure 3.20: Reduction of interval with its header.

1. The elimination phase

   This process is executed from innermost intervals to outermost intervals by traversing G in post order. If the header h of an interval I is encountered, the

[6]Interval order traversal for forward dataflow problems and post order traversal for backward dataflow problems.

dataflow effects of I (i.e., of all paths from h to each exit node) are computed. The effects are referred to here as an interval summary or a loop summary. After the interval summary is computed, I is reduced with h.

First, the dataflow effects of one cycle in the interval I are computed as follows.

Iteration summary

- Anticipatability

  The interval I is traversed in post order of I. That is, anticipatability is computed along the backward path, ignoring the outside effects of I.

  For each latch node l, we can consider $\mathrm{ANTOUT}(l)$ to be $\emptyset$. For each node n (other than l), we can compute $\mathrm{ANTOUT}(n)$ by using the anticipatability of the successor node s ($\mathrm{ANTIN}(s)$). And for each exit edge (in, out), we can ignore the value of $\mathrm{ANTIN}(out)$. That is, we can compute as follows:

$$\mathrm{ANTOUT}(n) = \bigcap_{\substack{s \in \mathrm{succ}(n) \\ s \in I}} \mathrm{ANTIN}(s)$$

  When n modifies induction variables, $\mathrm{TRANS_n}$ is considered as to be $\top$. [7]

  $\mathrm{ANTIN}(I^1)$ is defined as $\mathrm{ANTIN}(h)$ (h is the header). I^1 means one cycle in the interval (i.e., one iteration).

  If there are synchronization primitives in I, we can consider $\mathrm{TRANS_{I^1}}$ to be $\perp$. $\mathrm{TRANS_{I^1}}$ records the list of assigned locations in I.

- Availability

  The interval I is traversed in interval order of I. That is, availability is computed along the forward path, ignoring the outside effects of I.

[7] Shared-read operations cannot be optimized using continuous variables of a loop. The initial-values and final-values of these variables are available only after the loop is actually executed, while checking routines are executed before the loop.

For the first visited node h (i.e., header node), we can consider AVIN(h) to be $\emptyset$. For each node n (other than h), we can compute AVIN(n) by using the availability of the previous node p (AVOUT(p)).

When n modifies induction variables, $\mathrm{TRANS_n}$ is considered as $\top$.

AVOUT(I^1) is defined as $\bigcap_{l \in Latches(I)}$ AVOUT(l).

Interval summary

Figure 3.21 shows an example of computing an interval summary (anticipata-

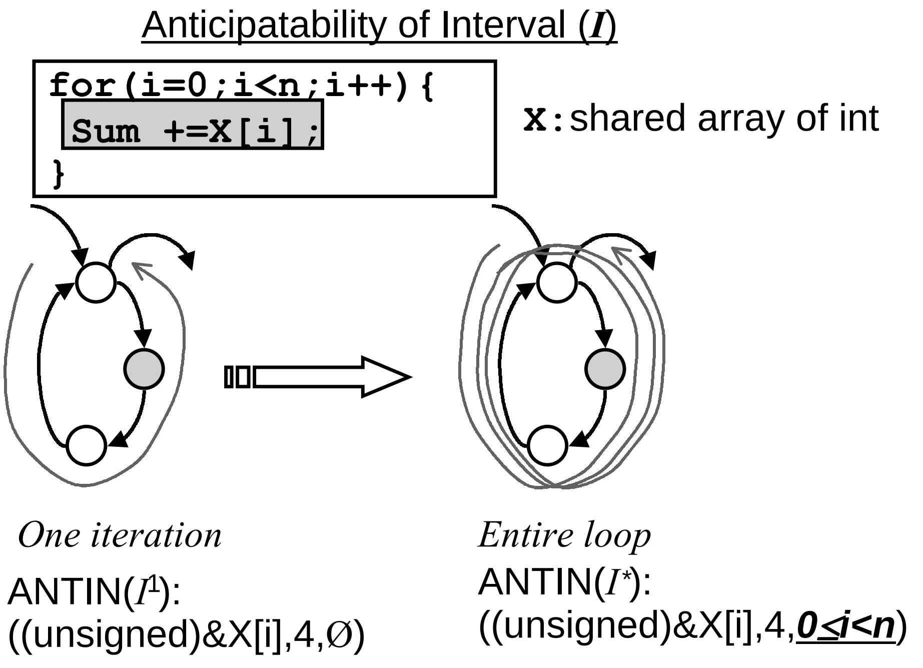


Figure 3.21: Computing interval summary (anticipatability).

bility in this example). The iteration summary (i.e., one cycle summary) of anticipatability ANTIN(I^1) is computed as (`(unsigned)&`$x[i], 4, \emptyset$). The effect of the entire loop is computed by reflecting the loop information to the ANTIN(I^1). This is done by adding the loop index information (i.e., a set of inequalities) to the iteration summary as follows.

$$\{(\texttt{(unsigned)\&}x[i], 4, \emptyset)\} \multimap \{0 \leq i < n\}$$
$$= \{(\texttt{(unsigned)\&}x[i], 4, \{0 \leq i < n\})\}$$

$\{(\texttt{(unsigned)\&}x[i], 4, \{0 \leq i < n\})\}$ is considered to be the interval (i.e., loop) summary of anticipatability.

In this way are computed the dataflow effects of n cycles in the interval I, where n is the number of iterations. n may be zero or not determined explicitly. In general, n depends on the exit nodes. For each exit node e are computed the effects of all the paths within I from h to e. This is done by adding $\prod_{e \in \text{Exits(I)}} \mathcal{C}_{h \mapsto e}$ to shared-access sets in the iteration summary. $\mathcal{C}_{h \mapsto e}$ corresponds to a set of inequalities representing the set of all paths within I from the header h to the e. When the set of all the paths within I from the header h to the e cannot be expressed as a set of inequalities, the summary cannot be propagated outward. In this case, $\mathcal{C}_{h \mapsto e}$ is considered as $\emptyset$. Hence, $\prod_{e \in \text{Exits(I)}} \mathcal{C}_{h \mapsto e} = \emptyset$, and the interval summary is considered to be $\emptyset$.

The process of adding a set of inequalities ($\mathcal{C}$) to a set of shared-access sets ($\mathcal{S}$) is denoted here as $\mathcal{S} \multimap \mathcal{C}$. This process returns the $\emptyset$ when $\mathcal{C}$ is $\emptyset$. Otherwise it returns the $\mathcal{S}$ whose elements include $\mathcal{C}$.

The reflective, transitive summaries of anticipatability and availability in I are referred to here as ANTIN(I^*) and AVOUT(I^*) and are defined as follows:

$$\text{ANTIN}(I^*) = \text{ANTIN}(I^1) \multimap (\prod_{e \in \text{Exits(I)}} \mathcal{C}_{h \mapsto e})$$
$$\text{AVOUT}(I^*) = \text{AVOUT}(I^1) \multimap (\prod_{e \in \text{Exits(I)}} \mathcal{C}_{h \mapsto e})$$

$\text{TRANS}_{\text{I}^*}$ is defined as $\text{TRANS}_{\text{I}^1}$. It is the reflective, transitive summary of the transparency in I.

Consider the example shown in Figure 3.22. Suppose that **`u`** is a shared array of **`double`**. The interval (loop) J has an induction variable j. ANTIN(J^1) and

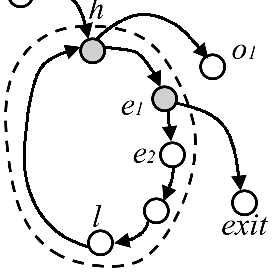
```
    ...
    n1 = 1<<q;
    base = n1-1;
    for (j=0; j<n1; j = j + 1) {
      if (base+j > rootN-1) {
        return;
      }
      u[2*(base+j)] = cos(2.0*PI*j/(2*n1));
      u[2*(base+j)+1] = -sin(2.0*PI*j/(2*n1));
    }
    ...
```

Figure 3.22: Sample loop with two exits.

AVOUT(J^1) are computed as follows:

$$\begin{aligned}\text{ANTIN}(J^1) = \text{AVOUT}(J^1) &= \{R1\} \bigsqcup \{R2\} \\ &= \{R3\}\end{aligned}$$

$$\begin{aligned}\text{where } R1 &= (\texttt{(unsigned)\&}u[2*(base+j)], 8, \emptyset) \\ R2 &= (\texttt{(unsigned)\&}u[2*(base+j)+1], 8, \emptyset) \\ R3 &= R1 \circ R2 = (\texttt{(unsigned)\&}u[2*(base+j)], 16, \emptyset)\end{aligned}$$



Interval ***J***

l:`j=j+1`
h:`(j<n)?e1:o1`
e1:`(base+j<=rootN-1)?e2:exit`

Figure 3.23: CFG of sample loop shown in Figure 3.22.

The interval J has two exit nodes e_1 and h. $\mathcal{C}_{h\mapsto h} = \{0 \le j < n1\}$ and $\mathcal{C}_{h\mapsto e_1} = \{0 \le j \text{ and } base + j \le rootN - 1\}$.

$$\begin{aligned}\mathcal{C}_{h\mapsto e_1} \bigcap \mathcal{C}_{h\mapsto h} &= \{0 \le j < n1\} \bigcap \{0 \le j \text{ and } base + j \le rootN - 1\}\\ &= \{0 \le j < n1\} \bigcap \{0 \le j < rootN - 1 - base + 1\}\\ &= \{0 \le j < \mathrm{Min}(n1, rootN - base)\}\end{aligned}$$

Hence,

$$\begin{aligned}\mathrm{ANTIN}(J^*) = \mathrm{ANTIN}(J^1) &\multimap (\prod_{e\in\mathrm{Exits(J)}} \mathcal{C}_{h\mapsto e})\\ &= \{R3\} \multimap \{0 \le j < \mathrm{Min}(n1, rootN - base)\}\\ &= \{((\texttt{unsigned})\&u[2 * (base + j)], 16, \{0 \le j < \mathrm{Min}(n1, rootN - base)\}\\ &= \{R3^{'}\}.\end{aligned}$$

Similarly $\mathrm{AVOUT}(J^*) = \{R3^{'}\}$.

Two optimizing methods are used to optimize shared-access sets when inequalities representing the loop are added.

Coalescing This is applicable when checking routines onto contiguous locations are issued in a loop. The name "coalescing" comes from its similarity to the loop transformation. Suppose the shared-access set $R = (f(\boldsymbol{j}), s, \mathcal{C}(\boldsymbol{j}))$ and suppose that the index variable vector $\boldsymbol{j}$ of the loop has the increment (or decrement) value vector $\boldsymbol{c}$.

Condition 3.6.4 (Coalescing) When $f(\boldsymbol{j} + \boldsymbol{c}) - f(\boldsymbol{j})(= \delta) \le s$, R can be replaced by

$$R' = (f(\boldsymbol{j_0}),\ \delta\cdot(n-1) + s,\ \mathcal{C}(\boldsymbol{j}) - \mathcal{I}(\boldsymbol{j}))$$

such that n is the number of iterations and $\boldsymbol{j_0}$ consists of the minimum values of $\boldsymbol{j}$ and $\mathcal{I}(\boldsymbol{j})$ is a set of inequalities with $\boldsymbol{j}$.

For the example shown in Figure 3.17,

$$R = (\texttt{(unsigned)\&}b[i], 1, \{0 \le i < n\}) \rightarrow R' = (\texttt{(unsigned)}b, n, \emptyset).$$

One benefit of this code-generation method is that the instruction overheads of the checking routines are reduced because the check is elevated from the loop. Another is that the run-time system can use the size information for message vectorization. The coalesced region may cover multiple caches. This case is detected dynamically at run-time and handled properly in the cache-state checking routine at run-time.

In practice, the above Condition 3.6.4 is tight. It is relaxed as follows.

Condition 3.6.5 (Coalescing) When $f(\boldsymbol{j}+\boldsymbol{c}) - f(\boldsymbol{j}) (= \delta) \le s$ *or* $\delta \le$ *block_size* $+ s$, R can be replaced by

$$R' = (f(\boldsymbol{j_0}),\ \delta \cdot (n-1) + s, \mathcal{C}(\boldsymbol{j}) - \mathcal{I}(\boldsymbol{j}))$$

such that n is the number of iterations and $\boldsymbol{j_0}$ consists of the minimum values of $\boldsymbol{j}$ and $\mathcal{I}(\boldsymbol{j})$ is a set of inequalities with $\boldsymbol{j}$.

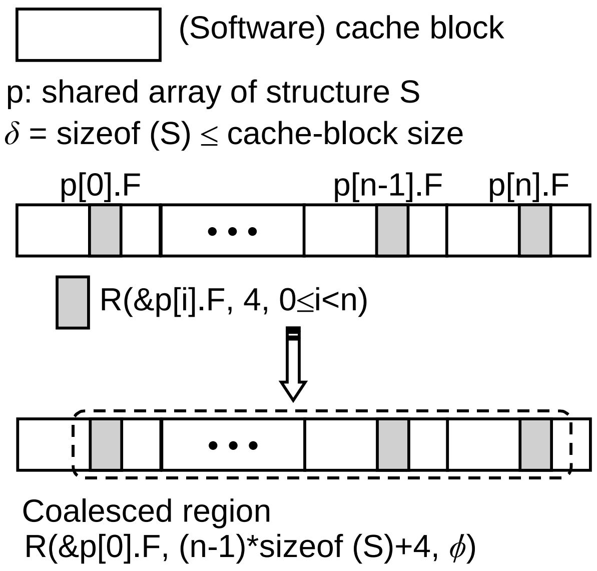

(Software) cache block
p: shared array of structure S
δ = sizeof (S) ≤ cache-block size
p[0].F
p[n-1].F
p[n].F
R(&p[i].F, 4, 0≤i<n)
Coalesced region
R(&p[0].F, (n-1)*sizeof (S)+4, φ)

Figure 3.24: Example of coalescing operation.

Consider the following example.

```
typedef struct mol_dummy
  {
    double VM[3];
    double F[7 + 2][3][3];
  }
molecule_type;
```

Suppose that **`VAR`** is a shared-array whose element type is **`molecule_type`** and the block_size of the software-cache is 1024 bytes. When an interval summary yields $R = (\texttt{(unsigned)}VAR[mol].VM, 24, 0 \leq mol < MOL)$, the computation is as follows.

$$\begin{aligned}\delta =& \texttt{(unsigned)}VAR[mol+1].VM - \texttt{(unsigned)}VAR[mol].VM\\ =& \texttt{sizeof (molecule\_type)}\\ =& 672 < \text{block\_size} - 24 = 1000.\end{aligned}$$

Therefore, we find that the Condition 3.6.5 is satisfied, and R can be replaced by

$$R' = (\texttt{(unsigned)}VAR, 672{\cdot}MOL - 648, \emptyset).$$

Redundant index elimination Suppose the shared-access set $R = (f(\boldsymbol{j}), s(\boldsymbol{j}), \mathcal{C}(\boldsymbol{j}))$ and suppose that $\boldsymbol{j}$ is the index vector of a loop.

Condition 3.6.6 (Redundant index elimination) When $\forall \boldsymbol{j}\ \exists \boldsymbol{j_0} \in \mathcal{C}(\boldsymbol{j})$

$$f(\boldsymbol{j_0}) \leq f(\boldsymbol{j}) + s(\boldsymbol{j}) \leq f(\boldsymbol{j_0}) + s(\boldsymbol{j_0}),$$

the index variable vector $\boldsymbol{j}$ becomes redundant. Therefore, inequalities with $\boldsymbol{j}$ ($\mathcal{I}(\boldsymbol{j})$) are removed from $\mathcal{C}(\boldsymbol{j})$.

$$R \to R' = (f(\boldsymbol{j_0}), s(\boldsymbol{j_0}), \mathcal{C}(\boldsymbol{j}) - \mathcal{I}(\boldsymbol{j}))$$

Consider the following example.

```
...
for(j=0; j<N; j++){
        c[j] = a[i]*b[j];
}
...
```

Suppose that **a** is in the shared-region and its element type is **int**. The loop summary generates $\{R\}$ where

$$R = ((\texttt{unsigned})\&a[i], 4, 0 \leq j < N)$$

(unsigned)&a[i] and 4 are loop-invariant. Therefore, the above Condition 3.6.6 is satisfied. As a result, the loop summary is $\{R'\}$ where

$$R' = ((\texttt{unsigned})\&a[i], 4, \emptyset).$$

Consider another example. Suppose that an array **c** is in the shared-region and that its element type is **int**.

$$\left.\begin{array}{l}\texttt{for(i=0;i<N;i++)}\\ \left.\begin{array}{l}\quad\texttt{for(j=0;j<N-i;j++)}\\ \qquad\texttt{...\ \ = c[i+j]}\end{array}\right\} = J\end{array}\right\} = I$$

When the summary of J is computed, $R1$ is coalesced into $R1'$:

$$R1 = ((\texttt{unsigned})\&c[i+j],\ 4,\ 0 \leq j < N - i)$$

$$R1' = ((\texttt{unsigned})\&c[i],\ 4\cdot\ (N - i),\ \emptyset)$$

Calculation of the summary of I produces {R2} such that

$$R2 = ((\texttt{unsigned})\&c[i],\ 4\cdot\ (N - i),\ 0 \leq i < N).$$

We find

$$(\texttt{unsigned})\&c[i] + 4\cdot(N - i) = (\texttt{unsigned})c + 4\cdot N,$$

since $(\texttt{unsigned})\&c[i] = (\texttt{unsigned})c + 4 * i$.

Therefore, we find that the Condition 3.6.6 is satisfied because

$$\begin{aligned} f(0) = \texttt{(unsigned)}c &\leq f(i) + s(i) \\ &= \texttt{(unsigned)}c + 4{\cdot}N = f(0) + s(0). \end{aligned}$$

Hence, we obtain

$$R2 \rightarrow R2' = (\texttt{(unsigned)}c, 4{\cdot}N, \emptyset).$$

This elimination is also applicable to nonlinear but monotonous expressions. For example, in a shared-read set

$$R = (x, 2 * 2^q * (N/2^q), \{1 \leq q \leq M\}),$$

we can eliminate q by using the monotonicity of 2^q and $Q{*}(N/Q)$ to obtain

$$R' = (x, 4 * (N/2), \emptyset).$$

Even if these optimization methods cannot be performed, there is still some value in computing the interval summary. The summary is propagated outward containing inequalities representing index variables of the loop. This process corresponds to the fission of the loop. That is, two loops are generated. One is the loop checking cache-states and the other is the loop reading the shared data actually and computing. Of course, fission does not reduce the number of cache-state checking routines. But it does improve the locality of reference.

Reduction

When an interval I is reduced to the header node h, each exit edge (i, e) is substituted for (h, e). The interval I is handled as a single CFG node h when the interval summary of the outer interval containing the interval I is computed. On that occasion, the anticipatability of the h is computed as follows:

$$\begin{aligned} \mathrm{ANTOUT}(h) &= \bigcap_{e \in \mathrm{Exits}(I)} \mathrm{ANTIN}(e) \\ \mathrm{ANTIN}(h) &= \mathrm{ANTIN}(I^*) + \mathrm{TRANS}_{\mathrm{I}^*}[\mathrm{ANTOUT}(h)] \end{aligned}$$

ANTIN(h) is recorded as ANTIN$^*(h)$. The availability of the h is computed as follows:

$$\text{AVIN}(h) = \bigcap_{\substack{s \in \text{pred}(h) \\ s \notin I}} \text{AVOUT}(s)$$

$$\text{AVOUT}(h) = \text{AVOUT}(I^*) + \text{TRANS}_{\text{I}^*}[\text{AVIN}(h)]$$

AVOUT(h) is recorded as AVOUT$^*(h)$.

2. The propagation phase

   For the original CFG G (i.e., not a reduced one), anticipatability and transparency are computed in the course of a single post order traversal. Immediately after the node n of interval I is evaluated, we redefine ANTIN(n) as ANTIN$(n) \bigsqcup$ ANTIN$^*(h)$. Availability is computed in the course of a single interval order for the original CFG G. Immediately after the header node h of an interval I is evaluated, we redefine AVOUT(h) as AVOUT$(h) \bigsqcup$ AVOUT$^*(h)$.

Interprocedural Approach

Up to now, procedure-calls have not been taken into account here. The procedure call, however, is also important for the efficient code generation. Our approach to interprocedural calculation of redundancy elimination dataflow equations is as follows.

- The transparency of the callee must be known to the call site in the caller procedure.

  - Synchronization information

    The optimizing compiler must inform the caller of whether does or does not the callee contain synchronization primitives by analyzing the callee procedure.

  - Modification information

    Whether or not the callee modifies the parameters of the shared-access sets can be determined by using the result of points-to analysis. This is done by checking the PTF of the callee. There is no need to analyze the callee procedure for this purpose.

- ANTIN at the entrance of the callee procedure is considered to be the COMP at the call site in the caller procedure, and there is no need to map ANTOUT at the call site in the caller to the ANTOUT at the exit of the callee.
- Availability is used only to determine whether the placement of the checking routine is safe. A safe placement is one for which, along any path, the number of checking routines is not increased by optimization. AVIN and AVOUT are therefore computed intraprocedurally.

When we take this approach, there is no need to make calling-context information known to the callee procedure, and the procedure summaries (TRANS and ANTIN at the entrance) are computed only once.

If we want to compute dataflow equations interprocedurally, we must handle a procedure that is called recursively or called through pointers. We refer to such a procedure an open procedure [20]. An open procedure does not return precise information.

The methods for solving dataflow equations interprocedurally are as follows.

1. For each statement i of each procedure, Whether i does or does not issue a shared read (i.e., COMP(i)) is calculated by using the results of points-to analysis, and COMP(call_site) is considered as $\emptyset$. Whether i is or is not a synchronization primitive is computed, and $\mathrm{TRANS_i}$ becomes $\perp$ when i is a synchronization primitive.
2. The CFG is extended with the Call Graph. The Call Graph is decomposed into strongly connected regions. It can be, therefore, considered to be acyclic. The optimizing compiler makes a bottom-up traversal of the Call Graph. For each region (r) of the Call Graph, it performs the following process.
    - When r is a singleton, The caller of r must be informed of the synchronization information. If there is a node i in the CFG of r such that $\mathrm{TRANS_i}$ is $\perp$, we set $\mathrm{TRANS_{call\_site}}$ in the caller as $\perp$.

Then, the following evaluation procedure EvalProc(r) is performed.

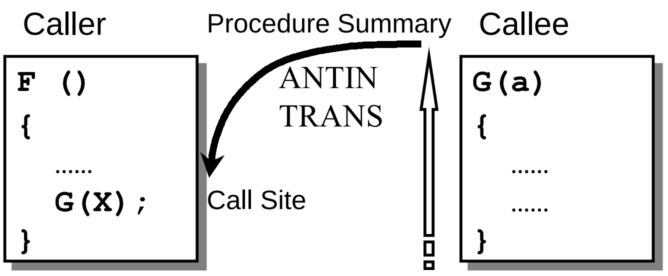


Figure 3.25: EvalProc process.

[EvalProc(r)]

Anticipatability and transparency are first computed by using the interval analysis framework. A call site (cs) that invokes s may be encountered. Because the Call Graph is traversed in the bottom-up manner, EvalProc(s) has already been performed. That is, $\mathrm{TRANS}_{\mathrm{cs}}$ and COMP(cs) have already been computed. When $\mathrm{TRANS}_{\mathrm{cs}}$[ANTOUT(cs)] is computed, the optimizing compiler investigates only the final points-to functions recorded in PTFs of the callee s and checks whether or not the parameters of elements in ANTOUT(cs) are modified in the s.
The availability is then computed by using the interval analysis framework. $\mathrm{TRANS}_{\mathrm{cs}}$[AVIN(cs)] is calculated as mentioned above. Finally, optimal placement of the checking routine (INSERT) is computed according to the equation shown in Figure 3.18. INSERT at entrance node e of the CFG of r cannot be computed without using the ANTIN value at a predecessor of e (denoted here as p).

$$\mathrm{ANTIN(p)} = \begin{cases} \emptyset & \text{if } r \text{ is an open procedure} \\ \mathrm{ANTIN}(e) & \text{else} \end{cases}$$

The anticipatability information is then made known to the call site. If r is not an open procedure (i.e., r is not called through a function pointer and r is not called by r itself), ANTIN(e) is reflected to the COMP(call_site) of the caller. At this time, each element of ANTIN(e) is translated back to the caller's name space. Although the lower bounds and the upper bounds of index variables in shared-access sets are translated, the index variables are the same.

- When r is not a singleton, the procedures in the region r are mutually recursive.

  First, we make the synchronization information known to the callers not only in the region r but also out the region of r. For each node s in r, the optimizing compiler checks for a node i in the CFG of s such that $\mathrm{TRANS_i}$ is $\perp$. If such a node is found, $\mathrm{TRANS_{call\_site}}$ is set to $\perp$ for all the call sites invoking the procedures in r.

  Second, we perform EvalProc (s) for each node s in r.

  In this case, the anticipatability information cannot be propagated to the caller.

Consider the example shown in Figure 3.26. COMP and TRANS are first computed for each CFG. For the CFG of **`f`**, $\mathrm{COMP}[s1] = \{R1, R2\}$[8].

$$R1 = (\texttt{(unsigned)\&}a[i], 8, \emptyset)$$
$$R2 = (\texttt{(unsigned)\&}b[i], 8, \emptyset)$$

The call graph is acyclic. Therefore, EvalProc(**`f`**) is computed first.
EvalProc(**`f`**)

Redundancy elimination dataflow equations are solved by using the interval analysis. First, anticipatability and transparency are computed. The loop with index variable i (interval I) is summarized. At the elimination phase, the summary is obtained as follows. $\mathrm{ANTIN}[I^*] = \{R1', R2'\}$ where

$$(\texttt{(unsigned)\&}a[i], 8, 0 \leq i < n) \to R1' = (\texttt{(unsigned)}a, 8 * n, \emptyset)$$
$$(\texttt{(unsigned)\&}b[i], 8, 0 \leq i < n) \to R2' = (\texttt{(unsigned)}b, 8 * n, \emptyset)$$

Hence, at the propagation phase, $\mathrm{ANTIN}[\mathrm{entrance}] = \{R1', R2'\}$ is obtained. Because **`f`** is not an open procedure, $\mathrm{ANTIN}[p] = \{R1', R2'\}$. Second, availability are

[8] **`sizeof(double)`** is considered here to be 8 bytes

```
double *A,*B,*C;
int N=100;
double alpha = 0.3;
main ()
{
   a1:A = G_MALLOC(100*sizeof(double));
   a2:B = G_MALLOC(100*sizeof(double));
   a3:C = malloc(100*sizeof(double));
   c1:f(A, B, C, N);
}
void f(double *a, double *b, double *c,
       int n)
{
   for (i=0;i<n;i++)
      s1:c[i] = a[i]+alpha*b[i];
}
```

Figure 3.26: Sample code segment.

computed like anticipatability computation. Lastly, optimal placement is obtained from these computed values. It is found that for each node x, INSERT$(x) = \emptyset$.

Because r is not an open procedure, ANTIN[entrance] must be reflected to COMP[`f(A,B,C,N);`] of the caller (i.e., **`main`**); that is, to COMP[`c1`] of **`main`**. $\{R1', R2'\}$ is translated back to the caller's name space (Figure 3.27). As a result, COMP[`c1`] $= \{R3, R4\}$

$$R3 = (\texttt{(unsigned)}A, 8*N, \emptyset) \;\; R4 = (\texttt{(unsigned)}B, 8*N, \emptyset)$$

EvalProc(**`main`**)

First, anticipatability and transparency are computed. The call site **`f(A, B, C, N)`** is encountered. COMP[`f(A,B,C,N);`] has already been computed. TRANS$_{\texttt{f(A,B,C,N);}}$ is considered as $\top$ because the final points-to functions

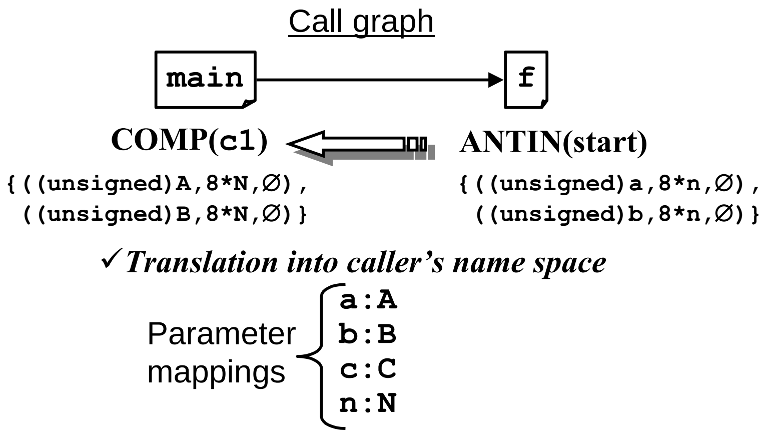


Figure 3.27: Apply summary of callee to call_site.

are empty in the procedure `f`. It is found that $\text{TRANS}_{\text{a2}}[\{R3, R4\}] = \{R3\}$ and $\text{TRANS}_{\text{a1}}[\{R3\}] = \emptyset$.

When availability and optimal placement are then computed, it is found that

$$\begin{aligned}\text{INSERT}(\texttt{a3}) &= \text{ANTIN}(\texttt{a3}) - \text{TRANS}_{\texttt{a2}}[\text{ANTIN}(\texttt{a2})] - \text{AVIN}(\texttt{a2})\\ &= \{R3, R4\} - \{R3\} - \emptyset\\ &= \{R4\}.\end{aligned}$$

Similarly, $\text{INSERT}(\texttt{a2}) = \{R3\}$.

These proposed optimization methods described in Section 3.6 are fully implemented in the RCOP, which optimizes shared-read operations in the UDSM/SSS–CORE system.

3.7 Shared-Write Optimization Methods

This section describes shared-write optimization methods by using the LRC model. They are similar to shared-read optimization methods. That is,

- Redundant routines for managing cache-consistency are removed.
- Multiple routines for managing cache-consistency are merged by using loop structures and procedure calls.

```
double *A,*B,*C;
int N=100;
double alpha = 0.3;
main ()
{
   a1:A = G_MALLOC(100*sizeof(double));
   R(A, N * 8);
   a2:B = G_MALLOC(100*sizeof(double));
   R(B, N * 8);
   a3:C = malloc(100*sizeof(double));
   c1:f(A, B, C, N);
}
void f(double *a, double *b, double *c,
      int n)
{
   for (i=0;i<n;i++)
      s1:c[i] = a[i]+alpha*b[i];
}
```

Figure 3.28: Output code of Figure 3.26.

These methods also calculate shared-access sets interprocedurally by using interval analysis [15] to solve redundancy elimination dataflow equations. The basic approach is derived from Inagaki's work [41], but is substantially extended by several techniques using continuous variables and by having all the functions implemented. Shared-write optimization methods are explained here being compared with shared-read optimization methods.

3.7.1 Remove Redundant Consistency-Management Routines

With the LRC model, consistency is enforced only at an acquire (see Figure 2.3 in Chapter 2). Shared-write operations are not visible to all other processors until a synchronization variable released in a subsequent operation is acquired by another processor. A consistency-management routine can, therefore, be placed between the corresponding store instruction to the subsequent synchronization primitive. This flexibility makes it easy to remove redundant consistency-management codes.

Redundancy Elimination Algorithm

The redundancy elimination algorithm for consistency-management routines is similar to that for cache-state checking routines.

COMP(i) indicates that i issues a shared-write. Hence, availability and anticipatability are defined as follows.

Availability The shared write is issued in all paths that precede i.

Anticipatability The shared write is issued in all paths that succeed i.

INSERT(i) is a variable meaning that the consistency-management routine is actually placed after i.

The number of consistency-management routines is minimized by placing them only where,

- a shared-write is available,
- a shared-write is not available in one of the succeeding paths, and
- a shared-write is not anticipatable.

Shared-write and synchronization primitive are assumed to not be issued at the same node. The redundancy elimination dataflow equations using shared-access sets are defined as follows. These dataflow equations are also resolved into uni-directional equations. That is, AVIN and AVOUT are first calculated as forward dataflow problems. Then, ANTIN, ANTOUT, and INSERT are calculated as backward dataflow problems.

$$\begin{aligned}
\mathrm{ANTOUT}(i) &= \bigcap_{s \in \mathrm{succ}(i)} \mathrm{ANTIN}(s) \\
\mathrm{ANTIN}(i) &= \mathrm{COMP}(i) \bigsqcup \mathrm{TRANS_i}[\,\mathrm{ANTOUT}(i)\,] \\
\mathrm{AVIN}(i) &= \bigcap_{p \in \mathrm{pred}(i)} \mathrm{AVOUT}(p) \\
\mathrm{AVOUT}(i) &= \mathrm{TRANS_i}[\,\mathrm{COMP}(i) \bigsqcup \mathrm{AVIN}(i)\,] \\
\mathrm{INSERT}(i) &= \mathrm{AVOUT}(i) - \bigcap_{p \in \mathrm{pred}(i)} \mathrm{AVOUT}(p) - \mathrm{ANTOUT}(i)
\end{aligned}$$

Figure 3.29: Redundancy elimination dataflow equations using shared access sets for consistency-management routines.

In contrast to what is done in the shared-read optimization, availability is mainly used and anticipatability is computed only to determine whether the placement of the consistency-management routine is safe.

3.7.2 Merge Multiple Consistency-Management Routines by Using Loop Structures

An Interval analysis framework is also used to solve the above dataflow equations efficiently. It is almost the same as the one used in the shared-read optimization.

However, several optimization methods are different from those in the shared-read optimization:

- Fusion
- Coalescing

These two optimization methods for shared-reads use the relaxed conditions such as Condition 3.6.2 and Condition 3.6.5. This is because the corresponding checking routines are merged when multiple shared reads are issued onto neighbor locations. On the other hand, only when multiple shared writes are issued onto contiguous locations, can the corresponding consistency-management routines be merged. For that reason, the strict Condition 3.6.2 is used when performing fusion optimization and the strict Condition 3.6.5 is used when performing coalescing optimization.

The point in shared-write optimization methods is to utilize continuous variables (see Section 3.4). The final value of a continuous variable in a loop is flow-sensitive. That is, how many times the continuous variable is incremented (or decremented) is not known until the loop is actually executed. Suppose that shared writes onto contiguous locations are issued using continuous variables in a loop. Their consistency-management routines can be coalesced by using continuous variables. The number of iterations (i.e., n in coalescing Condition 3.6.4) is computed using the final value of the continuous variable. The coalesced routine is placed after the loop. When the coalesced routine is executed, the final value has already been computed and the routine works correctly. In shared-read optimization methods, however, this process does not work correctly, because the merged checking routine is placed before the loop.

When shared-write optimizations using continuous variables are taken into account, the computation of the iteration summary of availability is a little more complicated.

- When the CFG node n modifies the continuous scalar variable v, $\mathrm{TRANS_n}$ eliminates the shared-access sets that depend on v and are not issued in the same basic block b that contains n. That is, shared-access sets s dependent on v and issued in the b are propagated to the successor of n. At the subsequent join node, s is further propagated to the successor node.

- When n modifies the continuous array variable and the location of the variable can be determined, $\mathrm{TRANS_n}$ is considered as $\top$.

Coalescing optimization using continuous scalar variables is defined as follows.

Coalescing This is applicable when checking routines onto contiguous locations are issued in a loop L. Suppose shared-access set $W = (f(j), s, \mathcal{C}(j, \boldsymbol{i}))$ and that the continuous variable in L, j, has the increment (or decrement) value c and that $\boldsymbol{i}$ is index variables of L.

Condition 3.7.1 (Coalescing) When $f(j+c) - f(j)(= \delta) \leq s$ and the assignment to the location of j dominates or post-dominates the corresponding shared write, W can be replaced with

$$W' = (f(j_0),\, \delta\cdot(n-1)+s,\, \mathcal{C}(j,\boldsymbol{i}) - \mathcal{I}(j,\boldsymbol{i}))$$

such that j_0 is the minimum values of j used in the shared write, j_n is the maximum values of j used in the shared write, n is the number of iterations $(= \frac{j_n - j_0 + 1}{c})$, and $\mathcal{I}(j,\boldsymbol{i})$ is a set of inequalities representing the loop L.

Consider the following loop I.

```
for (k = 0, i = 0; i < n; i++) {
    if (A[i] > 0) {
        B[k] = A[i];
        k++;
    }
}
```

Suppose **B** points to a shared address and that its domain type is **double**. k is a continuous scalar variable. When computing the interval summaries, one cycle summaries are obtained as follows.

$$\begin{aligned}&\text{ANTIN}(I^1) = \emptyset\\&\text{AVOUT}(I^1) = (\texttt{(unsigned)\&}B[k], 8, \emptyset)\end{aligned}$$

As a result, the interval summaries are obtained as follows.

$$\begin{aligned}&\text{ANTIN}(I^*) = \emptyset\\&\text{AVOUT}(I^*) = (\texttt{(unsigned)\&}B[k], 8, \{0 \leq k,\ 0 \leq i < n\})\end{aligned}$$

AVOUT(I^*) is further optimized by coalescing. It is found that

- the location of k is loop-invariant,
- $f(k+1, i) - f(k, i) = 8 \leq 8$

- assignment to the location of k post-dominates the shared write $B[k]$, and
- the minimum value is 0 and the maximum value is $k - 1$.

Therefore, it is coalesced as follows.

$$(\texttt{(unsigned)\&}B[k], 8, \{0 \le k,\ 0 \le i < n\}) \rightarrow (\texttt{(unsigned)}B, 8 * k, \emptyset)$$

If the target is ADSM/SSS–CORE system, the output code is as follows.

```
for (k = 0, i = 0; i < n; i++) {
    if (A[i] > 0) {
       B[k] = A[i];
       k++;
    }
}
W(B, 8*k);
```

Figure 3.30: Output code for ADSM/SSS–CORE system.

This optimization is not performed to shared-reads. Therefore, if the target is UDSM/SSS–CORE system, the output code is obtained as follows.

```
for (k = 0, i = 0; i < n; i++) {
    if (A[i] > 0) {
       R(&B[k], 8);
       B[k] = A[i];
       k++;
    }
}
W(B, 8*k);
```

Figure 3.31: Output code for UDSM/SSS–CORE system.

Consider coalescing optimization using the continuous array variable (shown in Figure 3.5). The continuous array variable cv is expressed as $g(i)$ where i

represents an index variable of the loop L. Suppose shared-access set $W = (f(cv), s, \mathcal{C}(i))$ and that the cv has the increment (or decrement) value c.

Condition 3.7.2 (Coalescing)
When

- the location of a continuous variable cv is loop-dependent (i.e., denoted as $g(i)$),
- $f(cv + c) - f(cv)(= \delta) \leq s$,
- the assignment to the location of cv dominates or post-dominates the corresponding shared-write, and
- max and min such that $g(min) \leq cv(= g(i)) \leq g(max)$ can be detected,

W can be replaced with

$$W' = (f(m(j)),\, \delta\cdot(n(j) - 1) + s,\, \mathcal{C}(i) - \mathcal{I}(i) + \{min \leq j \leq max\}\})$$

such that

- $m(j)$ is the minimum value of $g(j)$, $M(j)$ is the maximum value of $g(j)$,
- $n(j)$ is the number of iterations $(= \frac{M(j)-m(j)+1}{c})$, and
- $I(i)$ is a set of inequalities representing the loop L.

This indicates that the value of $g(j)$ must be stored before the loop-execution if $m(j)$ and $M(j)$ are to be obtained after the loop-execution.

Consider the following example. Suppose that **`B`** is a shared array whose element type is **`double`** and that **`A`** is an array whose element type is **`integer`**.

```
for (i = 0; i < n; i++) {
   j = (i >> 24) & 255;
   B[A[j]++] = z(i,j);
}
```

The execution model is shown in Figure 3.32. Interval summaries are obtained

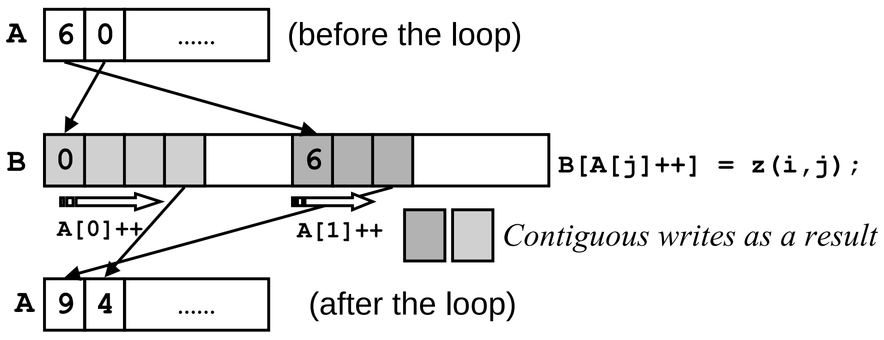


Figure 3.32: Execution model of memory access.

as follows.

$$\text{AVOUT}(I^1) = \{W\}$$
$$\text{ANTIN}(I^1) = \emptyset$$
$$\text{such that } W = (\texttt{(unsigned)}\&B[A[j]], 8, \{0 \le i < n\})$$

We find the following things.

- **`A[j]`** is a continuous variable that has the increment value 1.
- **`(unsigned)&B[A[j]+1]-(unsigned)&B[A[j]]`** ≤ 4.
- $\&A[0] \le \&A[j] \le \&A[255]$. This is because $0 \le (i >> 24)\ \&\ 255 \le 255$ holds for any i.

Coalescing optimization is performed as follows.

$$W \rightarrow ((\text{unsigned})\&B[I[k]], 8 * (A[k] - I[k]), \{0 \le k \le 255\})$$
such that $I[k]$ is the minimum value of $A[k]$.
i.e., $I[k]$ represents the value of $A[k]$ before the loop-execution.

For each $k \in \{0 \le k \le 255\}$, the value of **`A[k]`** is stored into **`I[k]`** before the loop-execution. Therefore, output code for the ADSM/SSS–CORE system is as follows.

```
for (k = 0; k < 255; k++)
    I[k] = A[k];
for (i = 0; i < n; i++)  {
     j = (i >> 24) & 255;
     B[A[j]++] = z(i,j);
 }
 for (k=0; k < 255; k++)
     W(&B[I[k]], 8*(A[k]-I[k]));
```

The output code for UDSM is as follows.

```
for (k = 0; k < 255; k++)
    I[k] = A[k];
for (i = 0; i < n; i++)  {
     j = (i >> 24) & 255;
     R(&B[A[j]],8);
     B[A[j]++] = z(i,j);
 }
 for (k=0; k < 255; k++)
     W(&B[I[k]], 8*(A[k]-I[k]));
```

These proposed optimization methods described in Section 3.7 are fully implemented in the RCOP, which optimizes shared-write operations both in the UDSM/SSS–CORE system and ADSM/SSS–CORE system.

3.8 Summary

The purpose of compiler optimization is to generate codes reducing the communication and instruction overheads of software cache-coherence management. This can be done by exploiting the application's semantics (such as loops and procedure calls) as much as possible by using the relaxed coherence model, interprocedural alias information and interprocedural redundancy elimination framework based on interval analysis.

An optimizing compiler called a "Remote Communication Optimizer" (RCOP) has thus been developed for shared-memory parallel programs. The RCOP performs

- interprocedural points-to analysis and
- interprocedural shared-access set calculation using interval analysis to solve redundancy elimination equations.

As a result, the RCOP implements the following optimizations:

- It detects all the shared-accesses precisely.
- It removes redundant cache-coherence management routines
- It merges multiple redundant cache-coherence management routines by utilizing loop structures and procedure calls.

The RCOP performs shared-write optimizations using continuous variables.

Chapter 4

Run-Time Optimization for Software Caching

This chapter describes the run-time optimization for Software DSM. The purpose of the run-time optimization is to manage cache-coherence efficiently in cache-management routines and synchronization primitives. The run-time system must reduce communication overheads and instruction overheads by using the relaxed coherence mechanism.

How many times the communication is invoked and the volume of the communication depend on the cache-coherence protocol. The effects of cache-coherence protocols that follow LRC model are therefore, first, studied by implementing and experimenting with three protocols described in Section 4.1.

Then, the communication overheads and instruction overheads incurred under the best protocol are further to be reduced. Because the run-time system needs to handle both fine-grained communications and coarse-grained communications issued through compiler-inserted interfaces efficiently, the bulk transfer mechanism provided by the platform is used for the coarse-grained communications , while as many as possible of the fine-grained communications whose destination processors are the same are combined. Remote requests are handled quickly with low overheads by utilizing the remote invocation mechanism of the user-specified program. This run-time system is described in Section 4.4.

Although the bulk data transfer mechanism and the remote invocation mechanism of the user-specified program are used in this thesis, its general applicability is not lost because they can be implemented with commodity hardware [59].

4.1 Cache-Coherence Protocol

The performance of parallel shared-memory applications written on the same memory model is considered to be affected by the cache-coherence protocol. Although Adve et al. [1] compare LRC implementations, little is known about protocol effects when fully optimized codes run. Three new cache-coherence protocols that implement LRC and support multiple writers for the codes fully optimized by the compiler are therefore evaluated [65, 68, 69].

The three protocols –called history-based lazy release consistency (HLRC), software emulation of AURC (SAURC), and a hybrid of HLRC and SAURC (HYBRID)– are described here.

First, the write history of shared-writes is defined as the tuple of the initial address and the size of the written region. These are the parameters of the consistency-management routines.

- History-based LRC protocol (HLRC):
  This is an implementation of the lazy invalidate protocol [46]. Invalidations are delayed until copy holders issue acquire operations. HLRC differs from the TreadMarks implementation [46], in not using a twin/diff mechanism because of the their large overheads.

  1. Write detection:
     A compiler-generated consistency-management routine saves the write history of the shared-write (i.e., the parameters of the routines).
  2. Write collection:
     By using the write history information, the processor computes the local write result efficiently when it is necessary (Figure 4.1). The time to collect the write result is proportional to the amount of the created write

history. The created write history cannot be discarded without explicit synchronization, and this means that a large amount of memory is required.

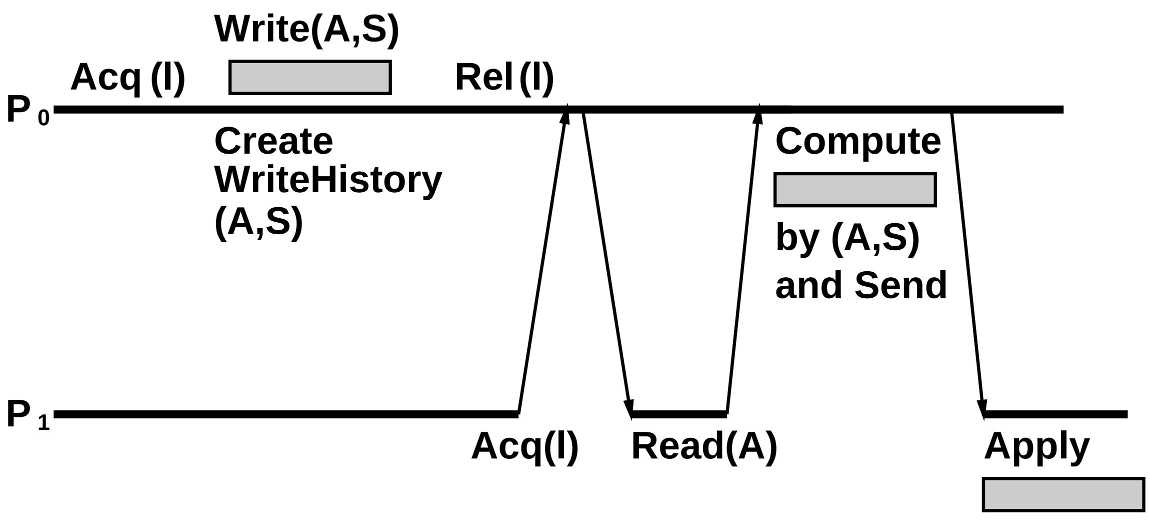


Figure 4.1: Example of HLRC protocol.

- Software emulation of AURC protocol [39] (SAURC):

  The AURC protocol can be implemented without special hardware support. It has also been implemented as Home-based LRC [88] which uses twin/diff mechanism. The implementation in SAURC, however, uses the write-history mechanism.

  1. Write detection:
     A compiler-generated consistency-management routine records the write history of the shared-write. At the release point, the local write result is propagated to the block-home processor by using the write history information (Figure 4.2). After sending the local write result to the block-home processor, the processor discards the created write history. The block-home processor discards the received write results after modifying the caches by using them.

2. Write collection:
   The home-block is always kept up-to-date. The consistency of other copies is managed as in the lazy invalidate protocol. When the cache-miss occurs, fetching the whole cache-block from the block-home processor is required.

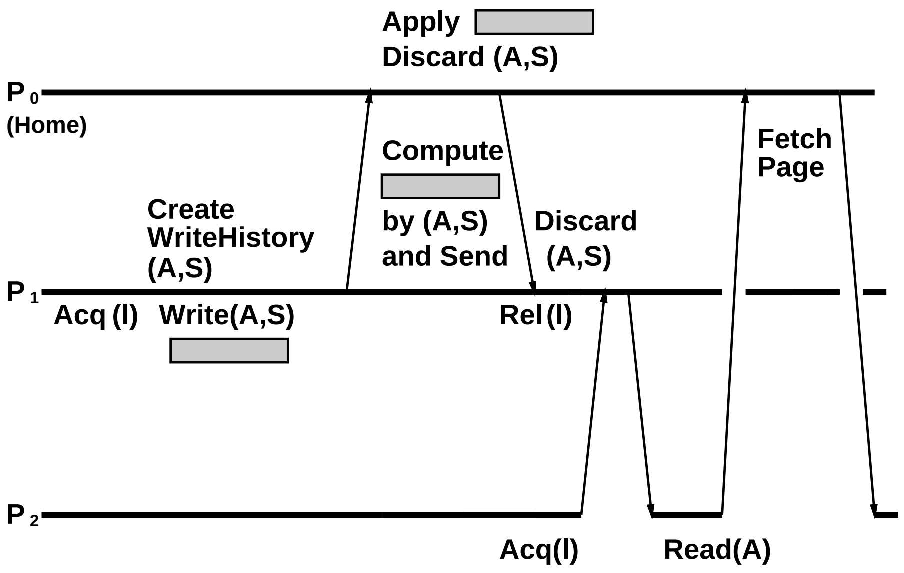


Figure 4.2: Example of SAURC protocol.

- HYBRID protocol:
  In the HLRC protocol, more than one remote processor may have to be visited in order to obtain write results at cache-misses. In the SAURC protocol, the processor has only to visit the block-home processor when there is a cache-miss, but the whole cache-block must be fetched from the block-home processor. Thus, in the SAURC protocol, the communication traffic is expected to be heavier than it is if only the modifications are sent. A hybrid protocol combining aspects of the HLRC and SAURC protocols is therefore developed.

  In HYBRID protocol, when there is a cache-miss, the processor visits the block-home processor and obtains only the modified data rather than the whole cache-block.

1. Write detection:
   A compiler-generated consistency-management routine saves the write history of the shared-write. At the release point the local write result is propagated to the block-home processor by using the write history information. After sending the local write result, the processor discards the write history. The block-home processor records the write history included in the received write results (Figure 4.3).

2. Write collection:
   The block-home processor is always kept up-to-date. The consistency of other copies is managed as in the lazy invalidate protocol. When a cache-miss occurs, the processor detecting the cache-miss sends its timestamp to the block-home processor. The block-home processor then sends only the modified data by using the received timestamp and the recorded write history information.

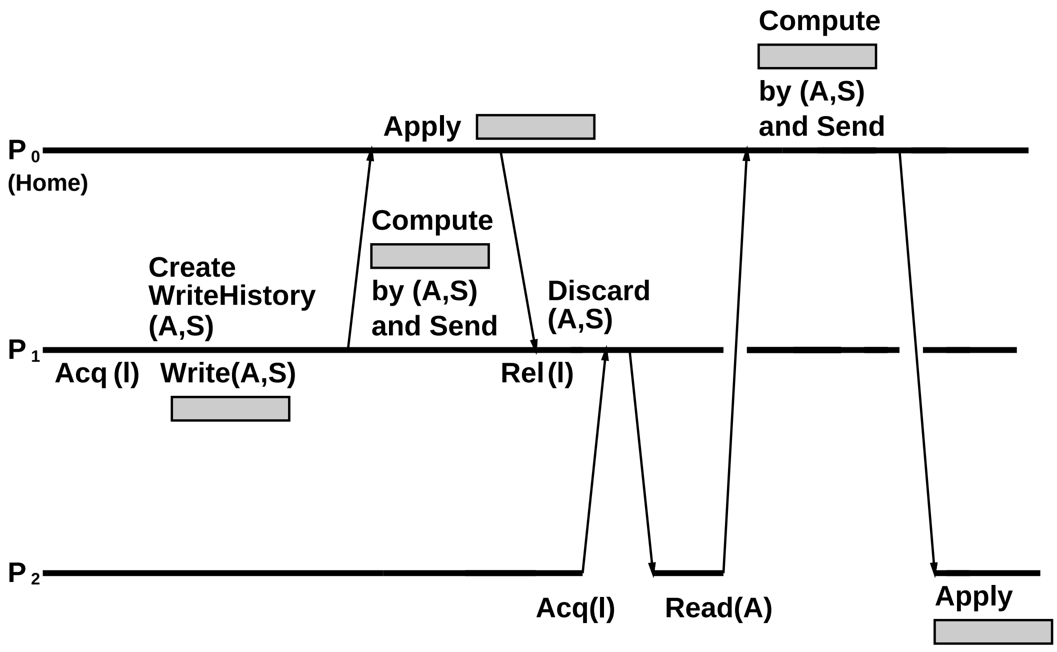


Figure 4.3: Example of HYBRID Protocol

4.2 Implementation Issues

These three protocols have been evaluated on the run-time system called “RS1”, and the construction of the RS1 is similar to that of the TreadMarks except for the twin/diff mechanism. The run-time system must prevent access to outdated versions of shared data, and TreadMarks does this by using a complicated timestamp mechanism called vector timestamp [46]. The following paragraph describes the vector timestamp mechanism that was also used in the RS1.

Vector timestamp mechanism

The execution of each process is divided into intervals, [1] each denoted by an interval index. Every time processor executes synchronization primitive, an interval index is incremented so that a new interval starts. Intervals of different processes are partially-ordered according to the acquire/release relationship.

- Intervals on a single processor are totally ordered by program order.
- An interval on processor p precedes an interval on processor q when the interval of q begins with the acquire corresponding to the release that concluded the interval of q.

The partial-order is represented by the vector timestamp assigned to each interval. The vector timestamp is a vector of timestamps and has each entry for each processor. The vector timestamp for interval i of processor p is denoted v_p^i, which is defined as follows. If $q = p$, then $v_p^i[q] = i$. Otherwise, $v_p^i[q]$ is equal to the most recent interval index of processor q that precedes the current interval i of processor p according to the partial-order. When a cache-block is modified by a local processor, the write-notice is created associated with the current vector timestamp. A write-notice is an indication that a cache has been modified during a certain interval.

When processor p acquires a lock from processor q, it sends its current vector timestamp to q. q then computes write-notices for all intervals included in q’s current vector timestamp but not in the vector timestamp received from p. After that, q

[1]They have no relation to the interval analysis described in Chapter 3.

returns to p not only the lock grant message but also its current vector timestamp and the computed write notices. When processor p receives the lock-grant message, it computes its new vector as maximum of its previous vector and the vector received from q and invalidates caches according to the write-notices.

At a cache-miss, the processor detecting the miss puts the associated vector timestamp in the cache-miss messages. The processors responding to the request send the required data and the vector timestamp associated with the cache. The processor that detected the miss then updates both the cache-block and the associated vector timestamp.

4.3 Empirical Evaluation of Protocol

The prototypes of the compiler and the above run-time system R1 of UDSM scheme were implemented on a multicomputer Fujitsu AP1000+[37, 71]. Each node consists of 50MHz Super SPARC with 20 KB I-cache, 16 KB D-cache and 16MB memory. The processors are interconnected by a 2-D torus network whose bandwidth is 25 Mbytes/sec per link. Because CellOS on the AP1000+ does not provide signals to users, request messages from remote processors are serviced through polling mechanism. Polls are inserted at every loop backedge and every function call[79]. The performance of each protocol on three kernels (LU-Contig, Radix, FFT) of SPLASH-2 [86] is evaluated using 16 nodes. LU-Contig performs blocked LU factorization of a dense matrix. The problem size is a 256×256 matrix with 16×16 blocks. Radix performs an integer radix sort. The problem size is 256K keys to be sorted and a radix of 1024. FFT performs a complex, 1D-FFT computation. The problem size is 16384 complex data points.

Note that in all cases the intraprocedural optimization described in Chapter 3 were performed. The results are shown in Figure 4.4, 4.5, 4.6. L shows the results when the HLRC protocol is selected. A shows the results when the SAURC protocol is selected. H shows the results when the HYBRID protocol is selected. The portion of each bar that is labeled “task” shows not only the computing time but also the cache-state checking time. The portion labeled “msg” shows the time spent in handling remote requests by polling. The portion labeled by “PF” shows the cache-miss time. The portion labeled by “CM” shows the consistency managing time at shared writes. The portion labeled by “sync”shows the synchronization time such as lock/unlock and barrier times. And the portion labeled “GC” shows the time spent in garbage collection.

Since on the AP1000+ the cache writes through, all modifications that the processor makes to cached data are reflected externally [37, 71]. This makes the cost of storage relatively high. Store instructions are executed most frequently in the “CM” time. Therefore, the “CM” time is relatively high even though the optimization reduces overheads for consistency-management.

In this implementation RS1/AP1000+, SAURC is best consistency protocol. The results for HYBRID are almost same as those for SAURC. These two protocols show better results than did the HLRC protocol. Although the network speed is relatively high in this platform, it seems important to maintain a home for each cache-block to which all updates are propagated and from which all copies are derived.

In three applications, HLRC causes large GC overheads. However, it should be noted that lazy invalidate protocols that implement LRC incur this memory-management overheads. As Yuanyuan et al. point out [88], the protocol memory requirements can be even larger than the application memory. In the diff-based system, the major memory consumptions are diffs and write-notices, and they must be kept until the garbage collection. Also in the history-based RS1, the created write histories are not discarded locally and occupies a large amount of memory. This causes garbage collection. The cache-miss time in the HLRC protocol is larger than the cache-miss times in other protocols because more than one remote processor may have to be visited in order to obtain the updates and there are non-negligible overheads for computing updates using vector timestamp mechanism.

In SAURC, when there are only two processors that have copies, update scheme is used. That is, not only the home-block but also the copy is updated. This causes a large amount of network traffic in SAURC, therefore, the overheads for consistency-management in SAURC become comparatively large. It should be noted that write histories are discarded almost immediately after they are created and applied, and

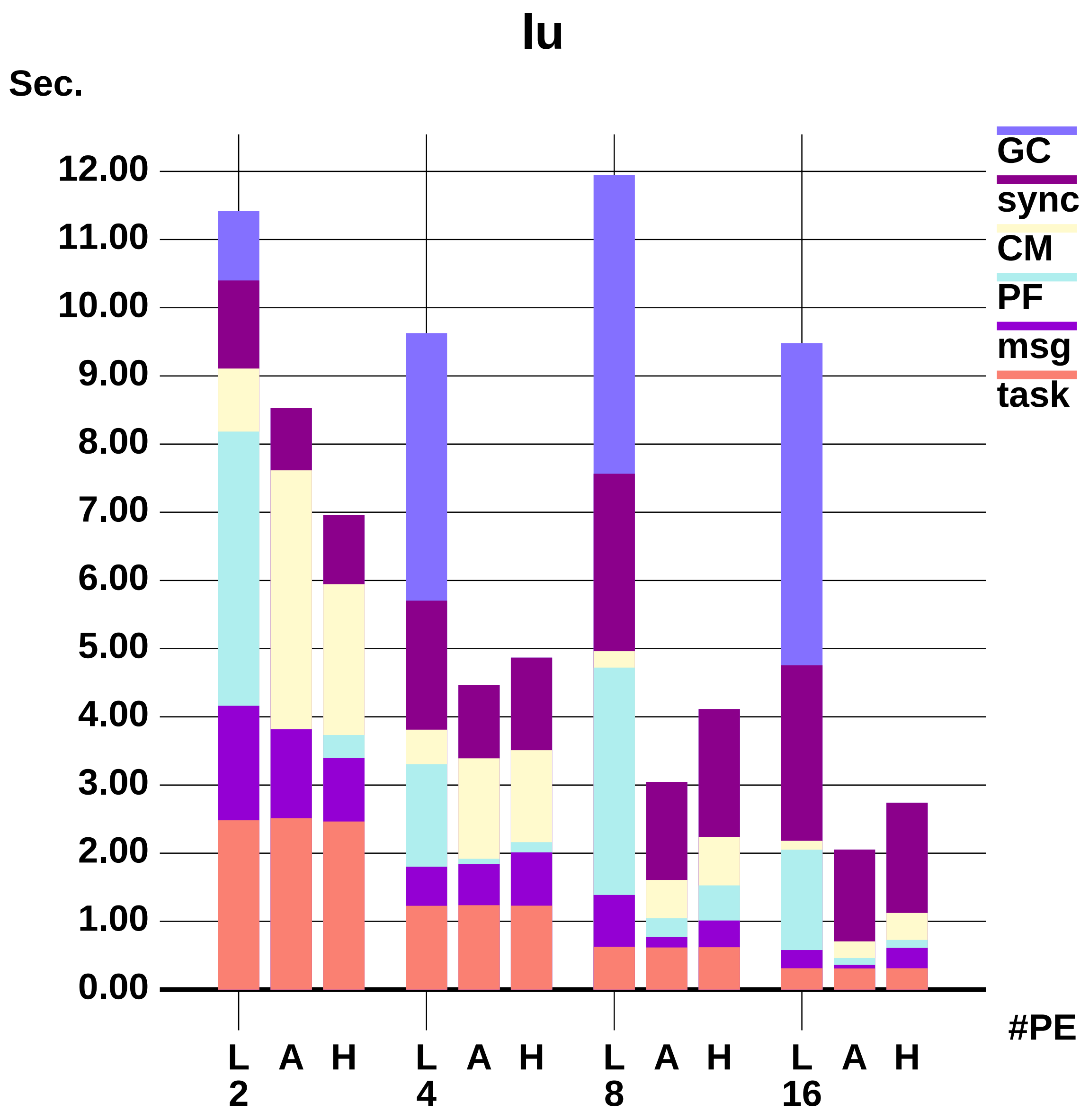


Figure 4.4: Cache-coherence protocol effects on LU-Contig.

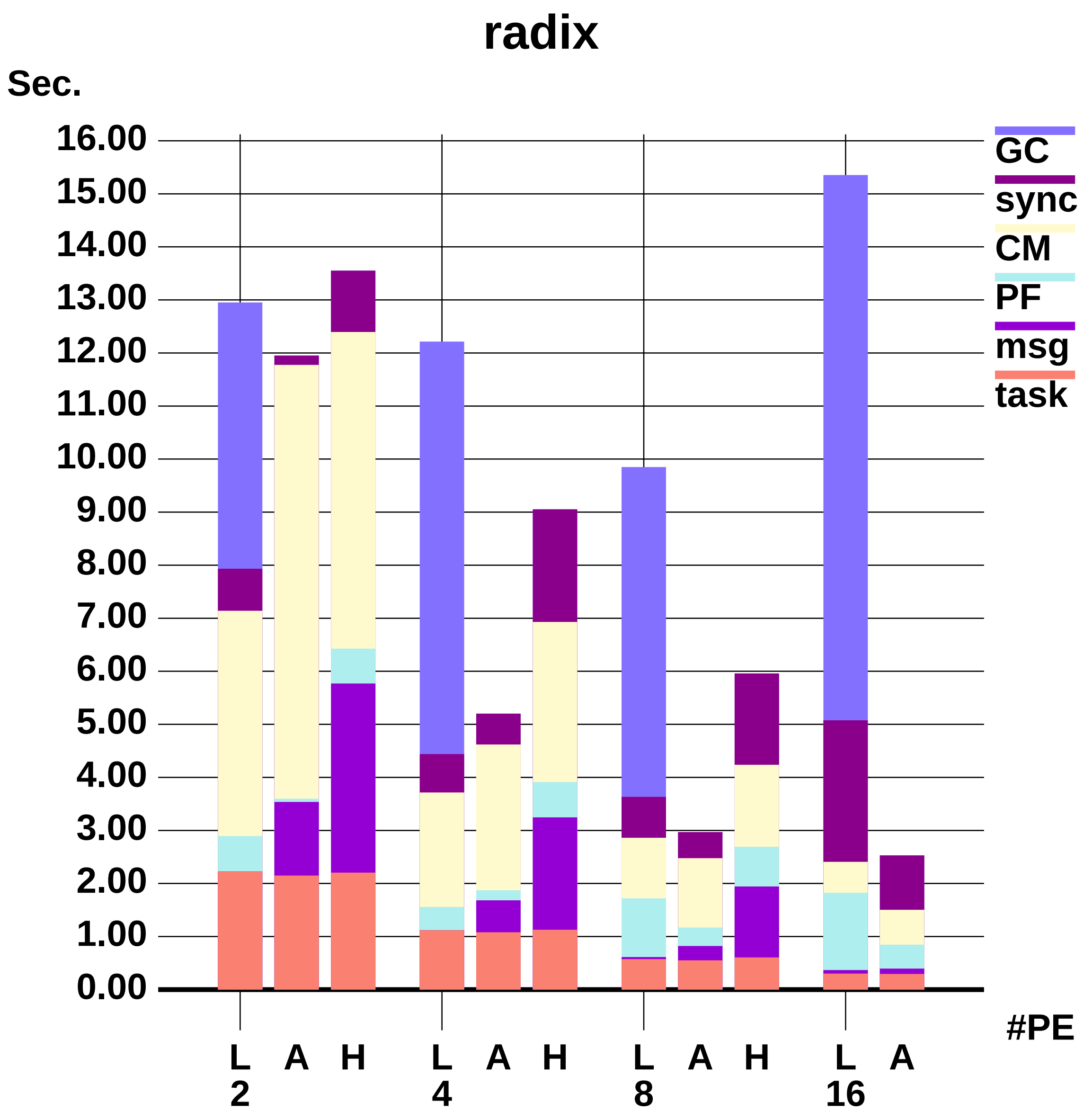


Figure 4.5: Cache-coherence protocol effects on Radix.

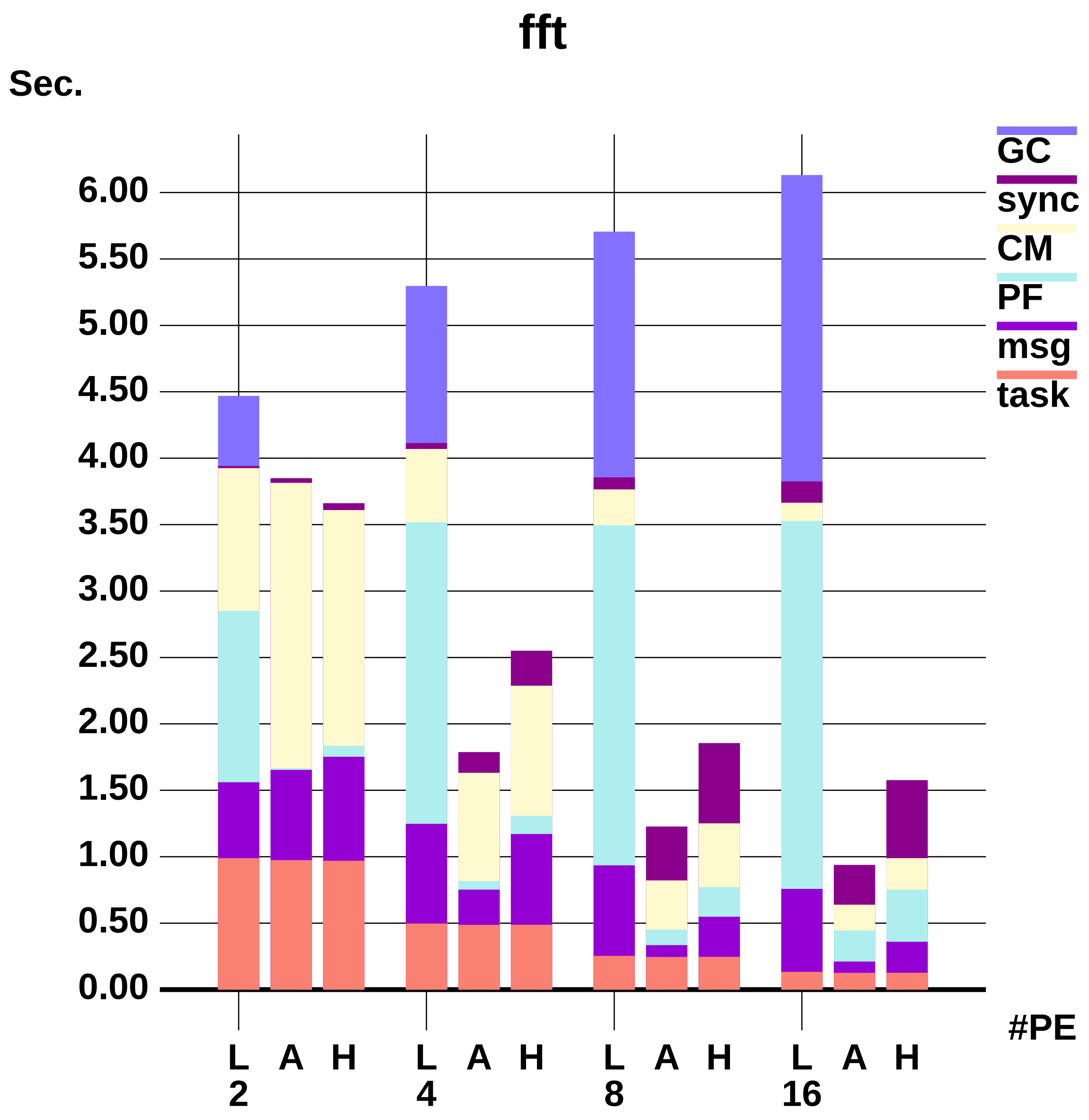


Figure 4.6: Cache-coherence protocol effects on FFT.

there are no need for garbage collections

In HYBRID, we cannot perform the Radix experiment because write histories are collected at the home-processor and the system at the home-processor becomes out of memory.

Of course, the vector timestamp mechanism preserves the partial-ordering among write-notices strictly, but the experimental results show that in all protocols the synchronization and cache-miss overheads increase as the number of processors increases. This indicates that the vector timestamp mechanisms can cause large overheads when the number of processors increases. It, therefore, seems that a lower-overhead timestamp mechanism is needed if the synchronization and cache-miss overheads are to be reduced.

4.4 Basic Design of a Lightweight Run-time System

Although the RS1 is a prototype system and its implementation was naive, data collected with the prototype system make us to design the new lightweight run-time system, called “RS2”. SAURC was used as the cache-coherence protocol in the lightweight run-time system RS2. To avoid excessive network communication, however, RS2 does not use bidirectional update scheme when the number of copy holders is two.

Furthermore, the home-usage makes it possible to preserve the partial-ordering among write-notices with low overheads. For each cache, one-bit write-notice indicating whether it has been modified since the last barrier operation [41] is maintained. The acquiring processor gets the one-bit write-notice table (i.e., bit vector) from the releasing processor and invalidates the caches according to the received bit vector. Note that the home-block is always kept up-to-date and never invalidated. The objection will no doubt be raised that unnecessary information is transfered when processor acquires a lock. Consider the following case. When a processor acquires a lock, modifies a cache, releases the lock, and acquires the same lock again, a cache may be invalidated even when the cache is not modified after the processor itself modifies the cache. However, the experimental results clearly show that the tracing of precise modification information in all cases incur large overheads. This low-overhead mechanism is therefore used to reduce the synchronization overheads and cache-miss overheads in many cases.

When the processor accesses the invalidate cache, the cache-miss occurs and the cache is kept up-to-date on demand by fetching the cache-block from the home-processor.

The lightweight runtime system RS2 has been implemented on CellOS/AP1000+ [41]. Its implementation is called RS2/AP1000+. Furthermore, the RS2 is extended for distributed-memory computers with commodity networks, called “RS3”. The RS3 also utilizes bulk data transfer mechanism provided by the platform.

The RS3 is different from the RS2 in the following aspects:

- Avoiding fine-grained communication

  The AP1000+ has dedicated hardware which executes remote block transfer operation(put/get interface [37]) and the communication network is fast. Written contents are, therefore, always sent to the home processor in a consistency-management routine, even if the data is fine-grained.

  When the compiler cannot fully optimize, however, fine-grained communications are frequently issued. Because a commodity network is not good for fine-grained communication, information is saved as write history in a consistency-management routine, and fine-grained communication packets whose destination processors (i.e., home) are the same are combined into a large packet by using write-history information at run-time. This optimization is called packet combining.

- Reducing the communication traffic at synchronizations

  When only a few blocks have been modified since the last barrier operation, transmitting the write-notice bit table for a large shared address space seems excessive. The RS3, therefore, records an updated block-number in a list apart from the write-notice bit table. When the number of updated blocks exceeds the size of the write-notice bit vector, the updated block-number is no longer recorded. At the release or barrier-arrival operation, if the size of the list is smaller than that of the write-notice bit table, the list is transmitted. Otherwise, the dirty bit table is transmitted.

- Handling remote requests quickly with low overheads

  The run-time system on AP1000+ detects remote requests by using a polling mechanism because CellOS on AP1000+ does not provide signals to users. A reasonable response time is ensured by inserting polls at each backedge in the CFG and procedure calls.

  For applications with coarse-grained synchronization patterns, however, these polling overheads are non-negligible. Therefore, it would seem useful to detect remote requests by using an interrupt mechanism. This is because the overheads for this kind of mechanism are reduced by the progressive techniques of recent operating systems such as RPC.

The details of the lightweight run-time system for distributed-memory computers with commodity networks RS3 are as follows.

4.4.1 Primitive Data Structure

A home processor is associated with each block in the shared region and the user can specify a block-home processor for each block. Each processor maintains the state information for each block in two bit tables.

Valid bit table has one-bit entries indicating whether or not the corresponding block is valid or invalid.

Dirty bit table has one-bit entries indicating whether or not the corresponding block has been modified since the last barrier operation.

Each processor maintains an updated block list recording the block-number modified since the last barrier operation. The size of the updated block list dose not exceed that of dirty bit table.

Each processor also manages a bit table with the size of the number of processors.

Acknowledge table indicates whether the processor has written into the block of the corresponding block-home processor.

Figure 4.7 shows an example of how to use these primitive structures. Shared data `X` resides on the cache `p` and `h` is the block-home processor of this processor. The system has `N` blocks and `n` processors. Suppose `X` is first invalid. Mechanisms of the cache-state checking routine (**`R`**) and the consistency-management routines (**`W`**) are described in the figure.

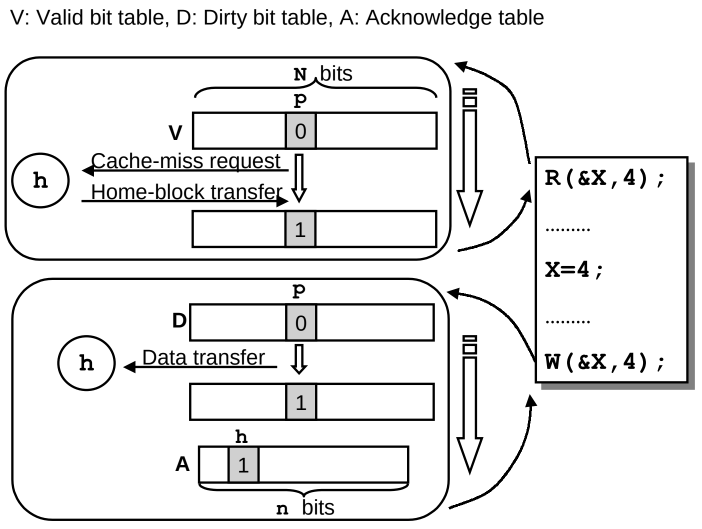


Figure 4.7: Mechanisms of cache-management routines.

The synchronization tags for locks are handled by specified synchronization-home (i.e., lock-home) processors. Each lock has its own dirty bit table, and its own updated block list.

Each processor uses $n-1$ combiningbuffers (n is the number of processors in the system). The processor associates each combiningbuffer with each processor except for the processor itself. The combiningbuffer associated with processor q (denoted as $CB(q)$) saves the fined-grained packets to q. The size of the combiningbuffer is the capacity of one packet.[2] The behavior of the run-time system for each primitive is

[2]In the work dealt with here, it is 1.4 KB.

described below.

4.4.2 Consistency-Management Routine

The outline of algorithm for the consistency-management routine is shown in Figure 4.4.2.

- The run-time system RS3 checks to see if the issued region resides on the shared-region.

  This is because the consistency-management routines may be issued to may-shared accesses.

- The RS3 checks if it is executed within the parallel task.

  The reason for this is that the consistency-management routines may be issued before parallel task starts. In other words, the consistency-management routines may be issued to shared-accesses before **CREATE** macro.

- The RS3 check to see if the size of the issued shared-write region is less than zero.

  The main reason is that the size may be expressed in the symbol, such as loop boundaries when loop optimization such as coalescing is performed.

- For simplicity, assume that the written region does not cover the multiple cache-blocks at first. Let h be the block-home processor of the written region and p be the writing processor (i.e., the processor issuing this routine). The RS3 reflects information of the written block into the dirty bit table and updated block list, and the information is also reflected into the dirty bit table and updated block list of the synchronization tags that p has acquired. If the number of updated blocks is larger than that of dirty bit vector, the updated block-number is no longer recorded in the list.

  If h is the same as p, the procedure does nothing. Otherwise, if the size of the written region is fine-grained, the write history (i.e., parameters of this routine) and the written contents are saved to the combiningbuffer for h, i.e., $CB(h)$.

```
inline void SAURCConsistencyManageCode(Address Addr,int BNumber)
{
    extern int SystemInitialized;
    Address StartAddr,PageStart,PageEnd,EndAddr;
    Address Start, End;

    Pid Home;
    int StartPN, EndPN;
    int PNumber;

    if (!SystemInitialized || BNumber <= 0)
        return;
    StartPN = GET_PAGE_NUMBER(Addr);
    EndPN   = GET_PAGE_NUMBER(Addr+BNumber-1);

    if (StartPN < 0 || StartPN >= TotalPageNumber)
        return;

    PageStart = GET_PAGE(Addr);
    PageEnd   = PageStart + PAGE_SIZE - 1;

    StartAddr = Addr;
    EndAddr   = (Address)(Addr + BNumber - 1);

    for(PNumber=StartPN;PNumber<=EndPN;PNumber++) {
        Home = PageHome[PNumber];

        Start = PageStart < StartAddr ? StartAddr : PageStart;
        End   = EndAddr < PageEnd ? EndAddr : PageEnd;

        SET_BIT_TABLE(AckTable,Home);
        if (!TEST_BIT_TABLE(DirtyBit,PNumber)) {
            SET_BIT_TABLE(DirtyBit,PNumber);
        }
        if (Home != MyProcNumber) {
            WriteForHome(Home,PNumber,Start,End-Start+1);
        }
        if (CEILING(UpdatedBlockNumber+1, sizeof(short))
            <
           CEILING(MAX_USE_PAGE_NUMBER, 32)) {
             UpdatedBlockList[UpdatedBlockNumber+1] =
             (unsigned short)PNumber;
             UpdatedBlockNumber++;
        }
        for (i = 0; i < LockPointer; i++) {
             SynchData *L = LockList[i];
             if (!TEST_BIT_TABLE(L->DirtyBitTable,PNumber)) {
                 SET_BIT_TABLE(L->DirtyBitTable,PNumber);
              if (CEILING(L->UpdatedBlockNumber+1,sizeof(short)) <
                  CEILING(MAX_USE_PAGE_NUMBER,32))
                L->UpdatedBlockList[L->UpdatedBlockNumber+1] =
                (unsigned short)PNumber;
             L->UpdatedBlockNumber++;
           }
        }
        if (Home != MyProcNumber) {
            WriteForHome(Home,PNumber,Start,End-Start+1);
        }
        PageStart += PAGE_SIZE;
        PageEnd   += PAGE_SIZE;
    }
    return;
}
```

Figure 4.8: Algorithm for the consistency-management routine.

```
inline void WriteForHome(Pid Home,int PNumber,
                         Address StartAddr,unsigned Size)
{
    int AlignSize = ALIGN(Size,sizeof(int));
    int Old = CombiningBufferCounter[Home] + 2*sizeof(int);
    int New = Old + 3*sizeof(int) + AlignSize;
    int *Ptr;

    if (New >= BUFFERSIZE) {
        TransferCombiningBuffer(Home);
        Old = CombiningBufferCounter[Home] + 2*sizeof(int);
        New = Old + 3*sizeof(int) + AlignSize;
        if (New >= BUFFERSIZE) {
            /* bulk data transfer */
            Put(Home,StartAddr,Size,NULL,NULL);
            return;
        }
    }
    /* create packet and combine */
    Ptr =(int *)(CombiningBuffer[Home] + Old);
    *Ptr++ = WRITE;
    *Ptr++ = StartAddr;
    *Ptr++ = Size;
    memcpy((char *)Ptr, (char *)StartAddr, Size);
    CombiningBufferCounter[Home] += (AlignSize + 3 *sizeof(int));
}
```

Figure 4.9: Algorithm for combining used in the consistency-management routine.

- When the $CB(h)$ becomes full, the writing processor p sends the contents of the $CB(h)$ to the block-home processor h asynchronously. The writing processor p does not wait for the completion of the communication and continues the execution. The interrupt routine at the block-home processor h receives the packet and executes write operations according to the contents of the packet.

- When the size of the written region is large, there is no need for combining and the write history is not saved.

  First, the contents of $CB(h)$ are sent to the processor h as mentioned above. Second, the writing processor p directly sends the written contents asynchronously by utilizing the bulk data transfer mechanism. The number of data copy is reduced by one from what it is in the fine-grained case. The writing processor p does not wait for the completion of the communication and continues the execution. The outline of algorithm for combining is shown in Figure 4.4.2.

- The block-home processor h is recorded in the acknowledge table.

Now consider the case in which the written region covers multiple cache-blocks. In this case, the written region is divided at the block boundary, and the process described above is applied to the separate written regions.

An advantage of this approach is that coherence management becomes more refined by adopting the cache-coherence protocol to the application's semantics. Multiple consistency protocols can be used to reduce further the number and volume of communications.

Modifying the behaviors of the consistency-management routines and the checking routines makes it possible to use other protocols such as the home-only protocol [41].

- Home-only protocol

  This is used to reduce cache-miss traffic at fetch-on-write. The writer updates the home-block without maintaining coherence.

  It should be noted that the contents of the caches written by this home-only protocol are inconsistent (i.e., the part actually written by the processor itself is certainly up-to-data but the other parts may be out-of-date) until the subsequent synchronization. Therefore, when a cache-miss is detected in the inconsistent cache before the subsequent synchronization, the home-block is not fetched from the block-home processor until the updates of the processor detecting the miss are reflected to the home-block.

  This is made possible by maintaining home-only acknowledge table [41]. This table indicates whether the processor has performed home-only shared-writes into the block of the corresponding block-home processor. The cache-miss handler first checks this table. If the block-home is recorded in this table, the processor detecting the cache-miss sends confirmation message to the block-home processor and confirms that its updates have been reflected. Then, the cache-miss handler fetches the home-block from the block-home processor. If the block-home processor is not recorded in this table, the processor detecting the cache-miss fetches the home-block from the block-home processor as usual.

Figure 4.10 shows example of home-only protocol. Suppose **a** is first invalid on the processor **p** and **h** is the block-home of cache **c** and the system has **n** processors.

In the UDSM scheme, the effects of home-only protocol is obtained by not checking the cache-state of the corresponding blocks to be written. This is done automatically by the RCOP.

In the ADSM scheme, the system call that validates the corresponding block (i.e, page) is required. Furthermore, after the home-only shared-write, it is necessary to restore the page-state. The procedures for validating/restoring the page-state are manually inserted.

4.4.3 Cache-State Checking Routine and Cache-Miss Handler

In a checking code in the UDSM scheme, the shared access range is first checked as in a consistency-management routine. Then, the processor checks the corresponding entry in the valid bit table. If the checking code is a merged one, multiple blocks within the whole read region are checked consequently. Otherwise, the single block is checked. The algorithm is shown in Figure 4.11. The home-block is always kept up-to-date and the state is always valid. When the processor accesses a cache that the other processors have modified–that is, a cache whose state is invalid–, a cache-miss occurs. When a cache-miss is detected, the block contents are copied from the block-home processor by utilizing the bulk data transfer mechanism. Of course, if the block-home is recorded in the home-only acknowledge table, the processor detecting the miss must confirm that updates of the processor detecting the miss itself have been reflected before the fetch.

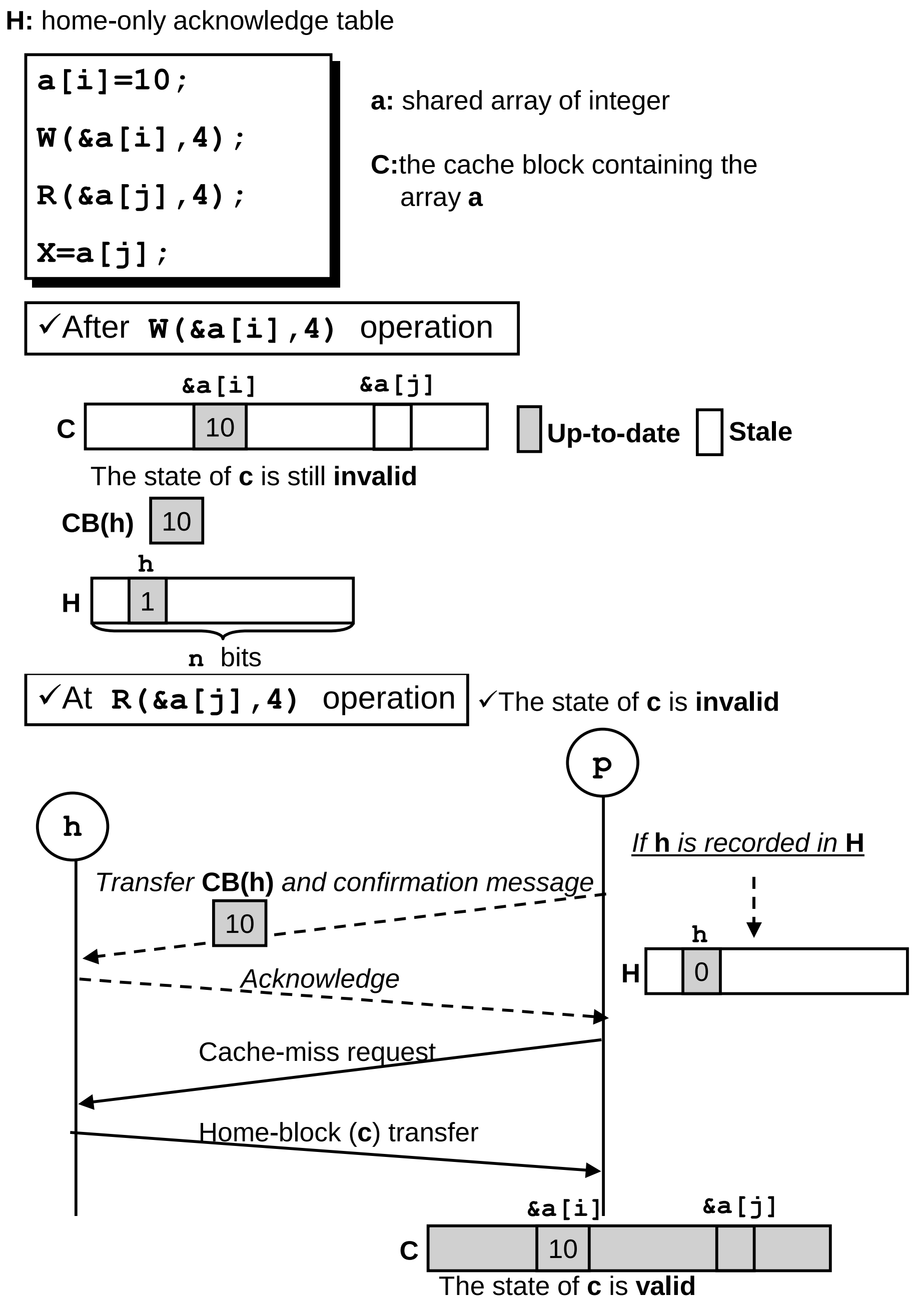


Figure 4.10: Example of home-only protocol.

```
    #define     CheckSharedPageTable(Address)                      \
({                                                                 \
    extern void                 PageFaultHandler (void *);         \
    extern unsigned long        PageValid[];                       \
    register unsigned long      _vpn;                              \
    register unsigned long      _s, _f;                            \
                                                                   \
    _vpn = GET_SHARED_PAGE_NUMBER((unsigned long) (Address));      \
    if (0 <= _vpn && _vpn < TotalPageNumber) {                     \
        _s = ((unsigned long) (_vpn)) >> 5;                        \
        _f = ((unsigned long) (_vpn)) & 31u;                       \
        if (!(PageValid[_s] & (1u << _f)))                         \
            PageFaultHandler ((void *) (Address));                 \
    }                                                              \
})
#define CheckChunkSharedPageTable(Address, Length)                 \
({                                                                 \
    extern void                 PageFaultHandler(void *);          \
    extern unsigned long        PageValid[];                       \
    register unsigned long      _fvpn, _tvpn, _vpn;                \
    register unsigned long      _s, _f;                            \
                                                                   \
    _fvpn = GET_SHARED_PAGE_NUMBER ((unsigned long) Address);      \
    if (0 <= _fvpn && _fvpn < TotalPageNumber) {                   \
        _tvpn = GET_SHARED_PAGE_NUMBER                             \
                ((unsigned long) Address + (Length) - 1);          \
        for (_vpn = _fvpn; _vpn <= _tvpn; _vpn++) {                \
            _s = ((unsigned long) (_vpn)) >> 5;                    \
            _f = ((unsigned long) (_vpn)) & 31u;                   \
            if (!(PageValid[_s] & (1u << _f)))                     \
                PageFaultHandler ((void *) GET_PAGEADDR (_vpn)); \
        }                                                          \
    }                                                              \
})
```

Figure 4.11: Inline-codes for cache-state checking routine.

```
#define SizeOfDT            ((MAX_USE_PAGE_NUMBER) / 32)
void SAURCLockAcquire   (unsigned LID)
{
    SynchData *L;
    int Msg[3];
    Pid Home;
    unsigned SizeofLT;
    L = &LockInfo[LID];
    SizeOfLT = CEILING(UpdatedBlockNumber+1, sizeof(short));
    Home = L->Home;
    /* Flush CombiningBuffer */
    WriteFlush();
    if (L->Home == MyProcNumber) {
        /** I am lock-home **/
        if (L->Acquired == TRUE) {
            /** Nobody has lock **/
            if (SizeOfLT >= SizeOfDT) {
                /* dirty bit table */
                ApplyDirtyBit(L->DirtyBitTable);
                MergeDirtyBit(L->DirtyBitTable);
            } else {
                /* updated block list */
                L->UpdatedBlockList[0] =
                (unsigned short)L->UpdatedBlockNumber;
                DecodeUpdatedBlcokListForLock(L);
            }
            L->Acquired = FALSE;
            /** Nobody gets lock **/
        } else {
            /** Someone has lock **/
            L->WaitQueue[L->Tail++] = (Pid)MyProcNumber;
            L->Tail %= MAX_PROC_NUMBER;
            while(!L->Acquired);
            SizeOfLT=CEILING(L->UpdatedBlockNumber+1,sizeof(short));
            if (SizeOfLT >= SizeOfDT) {
                /* dirty bit table */
                ApplyDirtyBit(L->DirtyBitTable);
                MergeDirtyBit(L->DirtyBitTable);
            } else {
                /* updated block list */
                L->UpdatedBlockList[0] =
                (unsigned short)L->UpdatedBlockNumber;
                DecodeUpdatedBlcokListForLock(L);
            }
            L->Acquired = FALSE;
            /** Nobody gets lock **/
        }
    } else {
        /* Send lock-grant message to home */
        L->Acquired = FALSE;
        Send (L->Home, Msg, 3*sizeof (int));
        while (!L->Acquired);
        SizeOfLT=CEILING(L->UpdatedBlockNumber+1,sizeof(short));
        if (SizeOfLT >= SizeOfDT) {
            /* dirty bit table */
            ApplyDirtyBit(L->DirtyBitTable);
            MergeDirtyBit(L->DirtyBitTable);
        } else {
            /* updated block table */
            L->UpdatedBlockList[0] =
            (unsigned short)L->UpdatedBlockNumber;
            DecodeUPList(L);
        }
    }
}
```

Figure 4.12: The pseudo code for acquire operation

```
#define SizeOfDT            ((MAX_USE_PAGE_NUMBER) / 32)
void    SAURCLockAcquireHandler     (unsigned *Msg)
{
    unsigned    LID             = Msg[0];
    Pid         ProcID          = Msg[1];
    SynchData   *L              = &LockInfo[LID];
    unsigned    SizeOfLT;
    SizeOfLT   = CEILING(L->UpdatedBlockNumber+1,sizeof(double));

    if (L->Acquired) {
        /** lock grant **/
        if (SizeOfLT >= SizeOfDT)
            /* dirty bit table */
            Put(ProcID,L->DirtyBitTable,
                SizeOfDT*sizeof(BitVector), L->DirtyBitTable,
                NULL,NULL);
        else {
            Put(ProcID,L->UpdatedBlockList,
                SizeOfLT*sizeof(unsigned long),L->UpdatedBlockList,
                NULL,NULL);
        }
        Put(ProcID, &L->UpdatedBlockNumber, sizeof(unsigned long),
            &L->UpdatedBlockNumber, NULL,&L->Acquired);
        L->Acquired = FALSE;
        /** No one can gain the lock **/
    } else {
        /** enqueue the message in WaitQueue **/
        L->WaitQueue[L->Tail++] = (unsigned char) ProcID;
        L->Tail %= MAX_PROC_NUMBER;
    }
}
```

Figure 4.13: The pseudo code for acquire handler operation

4.4.4 Acquire (Lock) Operation

The explicit lock-acquire message is always sent to the lock-home processor. Therefore, the lock-acquire message is serialized. At an acquire operation, the processor retrieves the updated block list of the lock from the lock-home processor if the size of the list is smaller than that of dirty bit table. Otherwise, the processor retrieves the dirty bit table. In any case, the valid bit table is updated according to the received information. In the ADSM scheme, a system call that invalidates the corresponding block (i.e, page) is also issued. The state of the cache that has been modified since the last barrier operation becomes invalid everywhere except in the home-block. For each cache, only a one-bit write-notice is required. It should be noted that the size of the synchronization messages is limited at most by that of the dirty bit table. The pseudo code of the acquire operation is shown in Figure 4.4.4 and the pseudo code of the acquire request handler operation is shown in Figure 4.4.4.

4.4.5 Release (Unlock) Operation

Figure 4.14 shows how the shared-write and release operations work.

In a release operation, the releasing processor first sends contents of each combiningbuffer if they are not empty. Then, the processor sends confirmation messages to the processors recorded in the acknowledge table and confirms that all sent messages have arrived at their destinations. These processes are executed in the procedure `WriteFlush` shown in Figure 4.4.5.

The explicit lock-release message is always sent to the lock-home processor. Therefore, the lock-release message is serialized.

- If the size of the updated block list of the lock held by the releasing processor is smaller than that of the dirty bit table of the lock, the releasing processor sends the updated block list of the lock back to the lock-home processor. The lock-home processor merges the sent updated block list into the updated block list of the lock.

- Otherwise, the dirty bit table of the lock is sent back to the lock-home processor.

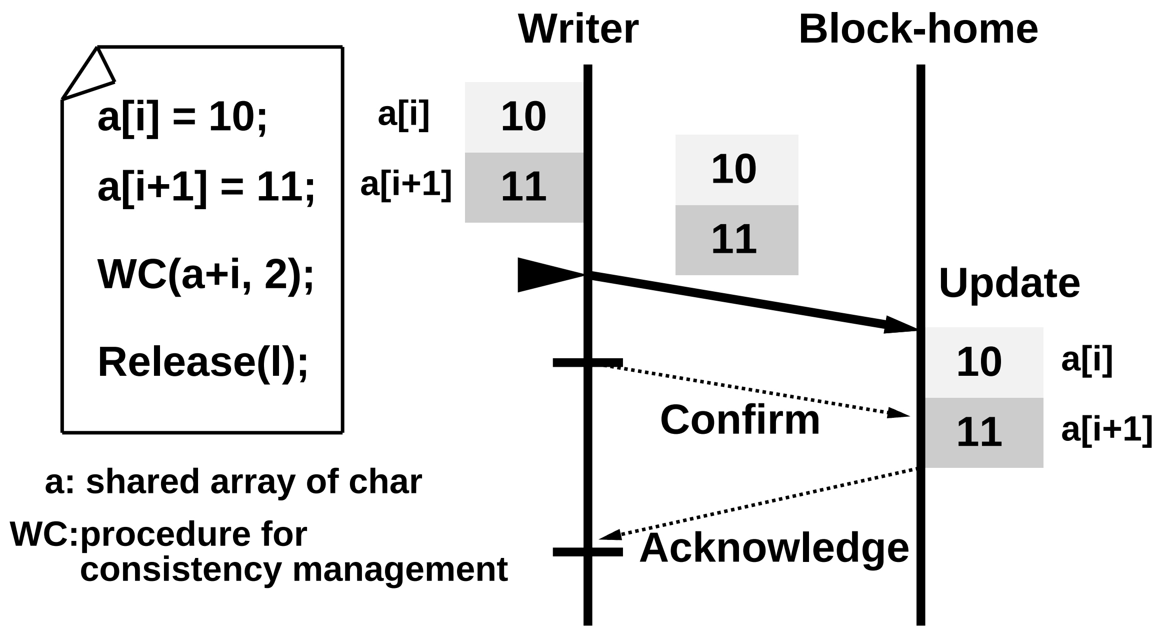


Figure 4.14: Behavior of shared-write and release operations.

The lock-home processor merges the sent dirty bit table into the dirty bit table of the lock.

The pseudo code of the release operation is shown in Figure 4.4.5, and the pseudo code of the release handler operation is shown in Figure 4.4.5.

4.4.6 Barrier Operation

At each barrier operation, the following steps are executed:

1. Each processor checks whether all the preceding block-home updates have been completed (in a procedure `WriteFlush()`).

2. Each processor sends its updated block list to the master processor if the size of the list is smaller than that of dirty bit vector. Otherwise, its dirty bit table is transmitted.

```
void SAURCLockRelease(unsigned LID)
{
    SynchData *L;
    int Msg[3];
    Pid Home;
    unsigned SizeOfLT;
    L = &LockInfo[LID];
    Home = L->Home;
    /* Flush CombiningBuffer */
    WriteFlush();
    if (L->Home == MyProcNumber) {
        /** I am home **/
        L->Acquired = TRUE;
        /** if acquire exists **/
        if (L->Head != L->Tail) {
            volatile int Flag = 0;
            Pid ProcID = L->WaitQueue[L->Head++];
            L->Head %= MAX_PROC_NUMBER;
            SizeOfLT = CEILING(L->UpdatedBlockNumber+1,sizeof(short));
            /** grant lock **/
            if (SizeOfLT >= SizeOfDT) {
                /* transfer dirty bit table */
                Put(ProcID, L->DirtyBitTable,SizeOfDT*sizeof(BitVector),
                    L->DirtyBitTable,NULL,NULL);
            } else {
                /* transfer updated block list */
                Put(ProcID,L->UpdatedBlockList,SizeOfLT*sizeof(BitVector),
                    L->UpdatedBlockList,NULL,NULL);
            }
            Put(ProcID, &L->UpdatedBlockNumber, sizeof(unsigned long),
               &L->UpdatedBlockNumber, NULL, &L->Acquired);
            L->Acquired = FALSE;
        }
    } else {
        /** send home my dirty bit table **/
        SizeOfLT = CEILING(L->UpdatedBlockNumber+1,sizeof(short));
        if (SizeOfLT >= SizeOfDT) {
            /* transfer dirty bit table */
            Put(L->Home, L->DirtyBitTable,SizeOfDT*sizeof(BitVector),
                L->DirtyBitTable,NULL,NULL);
        } else {
            /* transfer updated block list  */
            Put(L->Home,L->UpdatedBlockList,SizeOfLT*sizeof(BitVector),
                L->UpdatedBlockList,NULL,NULL);
        }
        Put(L->Home, &L->UpdatedBlockNumber, sizeof(unsigned long),
            &L->UpdatedBlockNumber, NULL,NULL);

        /** return lock to the home **/
        Msg[0] = (int)LOCK_RELEASE_REQUEST;
        Msg[1] = (int)LID;
        Msg[2] = (int)MyProcNumber;
        Send(L->Home,Msg,3*sizeof(int));
    }
}
```

Figure 4.15: The pseudo code of release operation.

```
void    SAURCLockReleaseHandler     (unsigned *Msg)
{
    unsigned            LID             = Msg[0];
    Pid                 ProcID;
    SynchData           *L              = &LockInfo[LID];
    unsigned            SizeOfLT;
    SizeOfLT = (L->UpdatedPageNumber + 2) >> 1;

    /** WaitQueue is not empty **/
    if (L->Tail != L->Head) {
        L->Acquired = TRUE;
        ProcID = L->WaitQueue[L->Head++];
        L->Head %= MAX_PROC_NUMBER;
        if (ProcID != MyProcNumber) {
            /* grant lock*/
            if (SizeOfLT >= SizeOfDT)
                Put(ProcID, L->DirtyBitTable,
                    SizeOfDT*sizeof(BitVector),
                    L->DirtyBitTable,NULL,NULL);
            else {
                Put(ProcID,L->UpdatedPageList,
                    SizeOfLT*sizeof(BitVector),
                    L->UpdatedPageList,NULL,NULL);
            }
            Put(ProcID, &L->UpdatedPageNumber, sizeof(unsigned long),
                &L->UpdatedPageNumber,NULL,&L->Acquired);
            /* Nobody can gain the lock */
            L->Acquired = FALSE;
        }
    } else {
        /** WaitQueue is empty **/
        L->Acquired = TRUE;
    }
}
```

Figure 4.16: The pseudo code of release handler operation.

3. The master processor merges the sent dirty bit tables and updated block lists. When the size of merged update block list is smaller than that of the dirty bit vector, and when none of dirty bit tables are sent to the master processor, the master processor broadcasts the merged updated block list. Otherwise, the master processor reflects all the received updated block lists into the merged dirty bit vector, and broadcasts the merged dirty bit vector.

4. All processors invalidate their copies by using the sent dirty bit table or updated block list.

5. Each processor clears its dirty bit table and updated block list, and it also clears the dirty bit table of synchronization tags it manages.

The pseudo code of the barrier operation is shown as follows.

```
BitVector *DirtyBitTmp[MAX_PROC_NUMBER];
volatile int BAFlag=0;
volatile int USBAFlag=0;
volatile int BDFlag[MAX_PROC_NUMBER]={0};
volatile int USBDFlag[MAX_PROC_NUMBER]={0};
void SAURCBarrier()
{
    int i,j;
    unsigned NumOfUP = CEILING(UpdatedBlockNumber+1, sizeof(short));
    int DirtyBitFlag = FALSE;


    if ((SystemInitialized == 0) || (TotalProcNumber == 1))
        return;

    WriteFlush();

    if (MyProcNumber != TotalProcNumber - 1) {
        /* I am not a master */
        if (NumOfUP >= SizeOfDT) {
            /* Send dirty bit vector*/
            Put(TotalProcNumber-1,DirtyBit,
                sizeof(BitVector)*(CEILING(MAX_USE_PAGE_NUMBER, 32),
                DirtyBitTmp[MyProcNumber],
                NULL,&BDFlag[MyProcNumber]);
        } else {
            /* Send updated-page list*/
            UpdatedBlockList[0] = (unsigned short)UpdatedBlockNumber;
            Put(TotalProcNumber-1,UpdatedBlockList,
                sizeof(unsigned long)*NumOfUP,
                DirtyBitTmp[MyProcNumber],NULL,&USBDFlag[MyProcNumber]);
        }
        /** Wait for ack from master **/
        UpdatedBlockNumber = 0;
        while(BAFlag + USBAFlag != 1){;}
        if (BAFlag == 1)
            /* Dirty bit vector */
            ApplyDirtyBit(DirtyBit);
        else
            /**  Updated page list **/
            DecodeUpdatedBlockNumberList((unsigned *)DirtyBit);

        BAFlag = USBAFlag = 0;
        for(j=0;j<ADSMLockCount;j++) {
            SynchData *SY = &LockInfo[j];
            if (SY->Home == MyProcNumber)
                ClearLockInfo(SY);
        }
        ClearBitVector(DirtyBit);
    } else {
        /* I am a manager */
        int Number = TotalProcNumber-1;
        /* reflect my dirty bit table/updated page list */
        if (NumOfUP >= SizeOfDT) {
            /* dirty bit table */
            CopyBitVector(DirtyBit,DirtyBitTmp[MyProcNumber]);
            DirtyBitFlag = TRUE;
        } else {
            /* update page list */
            UpdatedBlockList[0] = (short)UpdatedBlockNumber;
            DecodeUpdatedBlockNumberList((unsigned *)UpdatedBlockList);
            /* Translate updated page list 'UpdatedBlockList' to dirty bit
       table 'DirtyBitTmp[MyProcNumber]' */
            TranslateUpdatePageListToBitVector(UpdatedBlockList,
       DirtyBitTmp[MyProcNumber]);

        }
        /* receive dirty bit table/updated page list
           from each other processor */
        for(;;) {
            for(i=0;i<TotalProcNumber-1;i++) {
                if (BDFlag[i] + USBDFlag[i] == 1) {
                    if (BDFlag[i] == 1) {
                        /* dirty bit table has arrived */
                        CopyBitVector(DirtyBitTmp[i],
                                      DirtyBitTmp[MyProcNumber]);
                        DirtyBitFlag = TRUE;
                    BDFlag[i] = 0;
                } else {
                    /* updated page list has arrived */
                    DecodeUpdatedBlockList((unsigned *)DirtyBitTmp[i]);
                    /* gather update page list ’DirtyBitTmp[i]’
                     into ’UpdatePage’ */
                    GatherUpdatedBlockList((unsigned *)DirtyBitTmp[i],
                                          UpdatedBlockList);
                    /* translate update page list ’DirtyBitTmp[i]’
                     to dirty bit vector ’DirtyBitTmp[MyProcNumber]’*/
                    TranslateUpdatePageListToBitVector
                        (DirtyBitTmp[i],
                         DirtyBitTmp[MyProcNumber]);
                    USBDFlag[i] = 0;
                }
                Number--;
                if (Number == 0)
                    goto SendLoop;
            }
        }
    }
SendLoop:
    if ((DirtyBitFlag == TRUE)
        ||
       (CEILING(UpdatedBlockNumber+1, sizeof(short))
         <
        CEILING(MAX_USE_PAGE_NUMBER, 32)) {
        /* Send dirty bit vector to others */
        for(i=0;i<TotalProcNumber-1;i++) {
            Put(i,(void *)DirtyBitTmp[MyProcNumber],
                sizeof(BitVector)*(CEILING(MAX_USE_PAGE_NUMBER, 32)),
                (void *)DirtyBit,
                NULL,(volatile int *)&BAFlag);
        }
    } else {
        /* Send all the update page lists to others */
        UpdatedBlockList[0] = (unsigned short)UpdatedBlockListSize;
        for(i=0;i<TotalProcNumber-1;i++) {
            Put(i,(void *)UpdatedBlockList,
                sizeof(unsigned long)
                * CEILING(UpdatedBlockNumber+1,sizeof(short)),
                (void *)DirtyBit,
                NULL,(volatile int *)&USBAFlag);
        }
    }
    ApplyDirtyBit(DirtyBitTmp[MyProcNumber]);
    ClearBitVector(DirtyBit);
    UpdatedBlockNumber = 0;
```

```
        for(j=0;j<ADSMLockCount;j++) {
            SynchData *SY = &LockInfo[j];
                ClearLockInfo(SY);
        }
        ClearBitVector(DirtyBitTmp[MyProcNumber]);
        }
    }
}
```

4.5 Summary

In this chapter, the effects of cache-coherence protocols has been studied by developing and implementing the cache-coherence protocols that follow LRC model.

1. History-based LRC (HLRC)
2. Software emulation of AURC (SAURC)
3. Hybrid of HLRC and SAURC

Data collected with these prototype implementations shows that it is important to maintain a home for each cache-block and that the timestamp mechanism strictly preserving the partial ordering among write notices incurs relatively high synchronization costs and cache-miss costs.

The lightweight run-time system, called RS3, for cache-coherence based on SAURC has therefore been constructed to execute applications more efficiently under the distributed-memory system with off-the-shelf hardware.

- It maintains a one-bit write-notice for each cache indicating whether the cache has been modified since the last barrier operation. This write-notice reduces the synchronization costs and memory requirements. Furthermore, the updated block list mechanism is used to reduce the data transfer at synchronization operations.

- It utilizes the bulk data transfer mechanism to efficiently execute coarse-grained communication through the consistency-management routines issued by the optimizing compiler.

- It performs the fine-grained communications efficiently by combining those whose destination processors are the same and transferring as many as possible of them at once.
- It utilize the remote invocation mechanism of the user-specified program to handle remote requests quickly with low overheads.

The implementation issues and performance evaluation are described in the next chapter.

Chapter 5

Performance Evaluation

This chapter presents the results of evaluating performance of the optimizing methods developed in this thesis. After the experimental environment is explained in Section 5.1, SPLASH-2 benchmark suite [86] is explained in Section 5.2. For SPLASH-2 applications, overheads for cache-coherence management routines are described in Section 5.3, which confirms the effects of compiler optimization. The parallel performance and effectiveness of these optimizing techniques are presented in Sections 5.4 – 5.9. Finally, Section 5.10 compares the results with the results obtained using another user-level segment-based scheme, Shasta [79]. Section 5.10 also summarizes this chapter.

5.1 Environment

The lightweight run-time system for cache-coherence mechanism described in Chapter 4, called RS3, has been implemented under a scalable OS, SSS–CORE [57, 62, 60], on an SS20 workstation connected with the Fast Ethernet (100BASE-TX). The RS3 is almost the same for ADSM/SSS–CORE and UDSM/SSS–CORE. The SSS–CORE provides a protected and virtualized high-speed user-level communication and synchronization scheme called "Memory-Based Communication Facilities (MBCF)" [61, 59]. MBCF/100BASE-TX guarantees that packets always arrive and that they

Table 5.1: Peak bandwidthes of the MBCF/100BASE-TX (from Ref.[59]).

data size (byte)	4	16	64	256	1024	1408
100BASE-TX (Mbytes/s)	0.29	1.06	4.03	8.28	10.86	11.24

arrive in FIFO order. For programmers and compilers, the MBCF provides methods for the direct remote memory accesses in user task spaces. Middle-grained or coarse-grained data can be handled in one MBCF operation and multiple MBCF operations can be merged into one communication packet by combining optimizations. The command variation of the MBCF includes remote-memory-accesses (read and write), remote-memory-accesses with flag operations, atomic operations (swap, test&set, compare&swap), memory-based fifo, and memory-based signals [58, 59].

The MBCF supports remote invocation of the user-specified program in the privilege of the target task. The invocation mechanism (MBCF_SIGNAL) is similar to that of UNIX's signals, and invoked programs are executed only in the scheduling periods of the target task. Atomicity of the invoked program is ensured. The latency of this invocation mechanism is low.

The version of SSS–CORE used for performance evaluation is Ver. 1.1a. It is running on 16 nodes of the Sun SPARCstation 20 (85 MHz SuperSPARC $\times$ 1) as shown in Figure 5.1. Each node is equipped with Fast Ethernet SBus Adapter2.0, and is connected by Fast Ethernet with 3Com Super Stack II Switch 3900 (switching Hub). The SSS–CORE implements the MBCF on the 100BASE-TX (MBCF/100BASE-TX). Table 5.1 shows peak bandwidthes of the MBCF/100BASE-TX. Table 5.2 shows one-way round-trip latencies of the MBCF/100BASE-TX.

Table 5.2: One-way latency of the MBCF/100BASE-TX (from Ref. [59]).

data size (byte) command	4	16	64	256	1024
MBCF_WRITE (μs)	24.5	27.5	34	60.5	172
MBCF_FIFO (μs)	32	32	40.5	73	210.5
MBCF_SIGNAL (μs)	49	52.5	60.5	93	227.5

Figure 5.1: SSS–CORE/an SS20 workstation cluster connected with a 100BASE-TX Ethernet.

5.2 Applications

The effects of the optimization techniques developed in this work are evaluated by running the following nine benchmarks from Stanford ParaLlel Applications for Shared memory(SPLASH)-2 [86] on the ADSM/SSS–CORE system and the UDSM/SSS–CORE system: LU decomposition (LU-Contig), Radix, FFT, Barnes, Raytrace, Water-Nsquared (Water-NS), Water-Spatial (Water-SP) and Ocean (Ocean-RW) and Volume rendering (Volrend). For each program, a block-home and a lock-home processor are specified according to the optimization hints described in the source codes.

The source codes were modified as follows.

- LU-Contig

  A contiguous version was used. The owner of block (i, j) was exchanged with that of block (j, i).

- FFT

  The matrix transposition of the original FFT program is written so that a receiver reads the ports of the array. Severe false sharing, however, occurs because the receiver is not always the block-home processor of the read blocks. The program was therefore rewritten in such a way that a sender writes to the blocks whose block-home processors are the receivers.

- Raytrace

  In the original Raytrace program, a lock operation is used to increment the global counter in order to identify each ray uniquely. Because this creates a bottleneck in the execution and the counter is not used for actual computation, the lock operation was removed.

- Ocean-RW

  The noncontiguous Ocean program was modified to partition the grid rowwise [43].

- Barnes

The program was modified to build the octree of the particles sequentially [27].

For these measurements, the macros used in SPLASH-2 codes are converted by `m4` to procedure-calls that the RCOP can recognize. Then, the RCOP automatically transforms application programs into instrumented C programs explicitly containing user-level cache-coherence management routines. The compiling techniques described in Chapter.3 were fully implemented in the RCOP. The output C program is compiled by gcc 2.7.2 (optimizing level of "-O2"[1]) as the backend compiler, and then linked with the lightweight run-time library for user-level cache-coherence management to generate executable code.

Classifications of sharing patterns and synchronization granularity in the SPLASH-2 applications are listed in Table 5.3. The numbers of barriers and locks are actually measured in executing 16-processor execution, and the values are averages for 16 processors. [2]

Table 5.3: Classifications of sharing patterns and synchronization granularities in SPLASH-2 Applications.

Application	Concurrent Writer per Cache-block	Spatial Access Granularity	Number of Barriers	Number of locks	Temporal Synch. Granularity
LU-Contig	single	coarse	256	0	coarse
Radix	multiple	coarse	359	5	middle
FFT	multiple	coarse	10	0	coarse
Barnes	multiple	fine	2	10	coarse
Raytrace	multiple	fine	0	131	coarse
Water-NS	multiple	fine	10	4620	fine
Water-SP	multiple	fine	10	10	coarse
Ocean-RW	single	coarse	359	5	middle
Volrend	multiple	fine	4	0	coarse

[1]except for Ocean-RW

[2]Each problem size is described in Table 5.4.

5.3 Overheads for Cache-Management Routines

Table 5.4: Problem size and sequential execution time (sec) and overheads for cache-coherence management.

Program	Problem size	Sequential (s)	ADSM parallel 1PE (s)	Overhead (%)	UDSM parallel 1PE (s)	Overhead (%)
LU-Contig	2048^2 doubles	435.62	436.34	0.16	464.38	6.6
Radix	4M integer keys	6.49	6.53	0.61	6.85	5.5
FFT	1M complex doubles	19.14	20.86	8.9	19.79	3.3
Barnes	32K bodies	55.71	57.20	2.6	66.51	19
Raytrace	balls4, 128^2 pixels	171.41	171.44	0.017	175.80	2.5
Water-NS	4096 molecules	479.63	487.49	1.6	498.49	3.9
Water-SP	4096 molecules	53.23	54.92	3.1	58.79	10
Ocean-RW	258^2 ocean	20.76	21.67	4.3	24.47	18
Volrend	head	4.11	4.125	0.43	4.983	21

For each program, the problem size, execution times of the sequential program and of the parallel program generated for ADSM/SSS–CORE and UDSM/SSS–CORE on a single processor, and the percentage increase over the sequential time are listed in Table 5.4.

It should be noted that all the data used in this experiment are on the memory of one processor. It should also be noted that all accesses are classified as either local or must-shared by interprocedural points-to analysis. It is found that there are no may-shared accesses. This fact shows that the interprocedural points-to analysis we have adopted is precise. The sequential programs were generated by applying the NULL macro to the original shared-memory programs. The parallel programs were generated by the RCOP with our proposed optimizing techniques.

For Raytrace, Water-NS and Water-SP, the block size of the UDSM/SSS–CORE system was 1 KB. For the other programs, the block size was 4 KB. The block size of the ADSM/SSS–CORE system was 4 KB (the underlying page size).

The overheads of the parallel programs executed on a single processor are due to cache-coherence management codes (hence, with no cache-misses). The overheads for ADSM/SSS–CORE ranged from 0.017% to 8.9%. The overheads for UDSM/SSS–CORE ranged from 2.5% to 21%. The instruction overheads for user-level cache management were thus reduced by using the proposed optimizing techniques. Furthermore, the overheads decrease when the number of processors is increased.

The overhead for ADSM/SSS–CORE includes the shared-write operations needed when using user-level consistency-management codes, while that for UDSM/SSS–CORE includes the shared-read operations needed when user-level checking codes. In the FFT program, the overhead for ADSM/SSS–CORE is much larger than that for UDSM/SSS–CORE because to implement the home-only protocol, ADSM/SSS–CORE issues system calls that validate pages. UDSM/SSS–CORE, in contrast, omits checking codes automatically.

Barnes and Volrend contain fine-grained shared-read accesses for which loop-level optimization is not performed. Therefore, the overhead for cache-coherence management on UDSM/SSS–CORE becomes about 20% of the sequential execution time. Ocean-RW contains near-neighbor memory accesses and memory accesses at block boundary are not contiguous. Coalescing optimizations are also not performed to these accesses. Furthermore, in Ocean-RW, the number of shared-read operations is very large and instrumented program size becomes quite large. Therefore, gcc cannot compile Ocean-RW with optimization level "-O2". Hence, the shared-read overhead of Ocean-RW on UDSM/SSS–CORE is 18% of the sequential execution time.

5.4 Optimization Effects on Parallel Execution

The total effects of all the optimizations for ADSM/SSS–CORE executed on 16 processors are listed in Table 5.5, where "WCs" is the average number of consistency-management routines. "Msgs" is the total number of messages and "MsgSz" is the total message size. "NO" indicates the data without optimization and "YES" indicates the data with full optimization.[3] It should be noted that optimization reduces the execution times for all applications.

[3]Note that precise shared-access detection is performed to both.

Table 5.5: Effects of optimization for ADSM/SSS–CORE (executed on 16 processors).

Application	Optimize	Time(s)	WCs	Msgs(K)	MsgSz(MB)
LU-Contig	NO	222.91	178111129	128.74	173.54
	YES	34.03	44204	128.74	173.54
Radix	NO	11.90	540927	7996.92	447.68
	YES	1.91	5970	34.84	20.27
FFT	NO	183.30	7192576	17990.07	1172.62
	YES	4.93	6528	92.99	99.46
Barnes	NO	43.33	176410	2692.86	268.10
	YES	8.82	51948	32.90	22.63
Raytrace	NO	12.81	6518	159.99	87.69
	YES	11.76	2306	64.15	75.52
WaterNS	NO	45.60	112902	2263.73	543.94
	YES	43.56	21260	648.44	441.39
WaterSP	NO	15.28	253549	4041.59	573.55
	YES	6.19	64558	140.28	146.59
OceanRW	NO	4.89	217414	635.42	111.21
	YES	3.89	204734	80.32	79.24
Volrend	NO	0.90	10419	167.87	22.89
	YES	0.49	1400	9.13	11.76

In LU-Contig, the number and volume of communications are the same with and without optimization. The reason for this is that LU-Contig does not contain remote-writes, That is, data are transferred only when there are cache-misses. Of course, interprocedural compiler optimization methods reduce the number of consistency-management routines by 99.8 % because LU-Contig has regular shared-memory-accesses in the loops and procedure calls.

In Radix, FFT, Barnes and Volrend applications, both the number and volume of communications are considerably reduced. This is because they have shared-memory-accesses in the loops. Inserting a consistency management routine in a loop causes a large overhead of procedure calls and reduces memory access locality. Optimization is, therefore, quite effective in these applications. The number and volume of communications in Radix and FFT are especially reduced because the home-only protocol optimization prevents the severe false sharing at fetch-on-write.

Raytrace, Water-NS, and Ocean-RW have fine-grained synchronization granularities. Our optimization can change fine-grained memory accesses into coarse-grained ones while keeping the meaning of parallel programs. Fine-grained synchronization granularities, therefore, cannot be changed by our optimization. Nonetheless, the number and volume of communications are quite reduced. Furthermore, Raytrace and Water-NS originally have high task ratios and communications are not the bottlenecks in the parallel executions. As a whole, it follows from these results, that the optimizing methods developed in this work are also quite effective in the parallel execution.

5.5 Effects of Shared-Write Optimization

Figure 5.2 and 5.3 respectively show the effects of RCOP optimization for shared-writes in 16-processor execution under ADSM/SSS–CORE and UDSM/SSS–CORE. For each program, the left bar is the execution time without shared-write optimizations (base time), and the right bar is the execution time with the optimization. The other optimizations (protocl and run-time) are performed to both. Execution time is normalized by the base time. “Sync” is the waiting time for synchronization. “CM” is the time for consistency-management routines. “Miss” is the waiting time for cache misses. “Msg” is the message-handling time for synchronization, and “Task” is the time for the original computation in ADSM. “Task” in UDSM includes time for the inline cache-state checking. These notations are used throughout this chapter.

Shared-write optimization under the SAURC protocol leads to communication optimization because the home-update messages are issued in shared-write operations. Furthermore, the MBCF on SSS–CORE provides direct remote-block transfer operations. Therefore, if the RCOP detects the coarse-grained / middle-grained shared-accesses, the performance is improved substantially. This is clearly shown in Figure 5.2 and Figure 5.3.

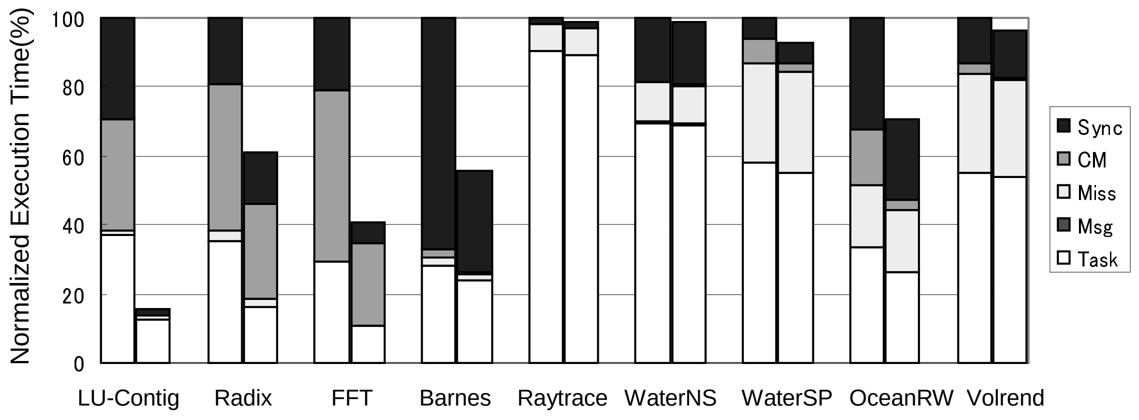


Figure 5.2: Effects of shared-write optimization for ADSM/SSS–CORE (executed on 16 processors).

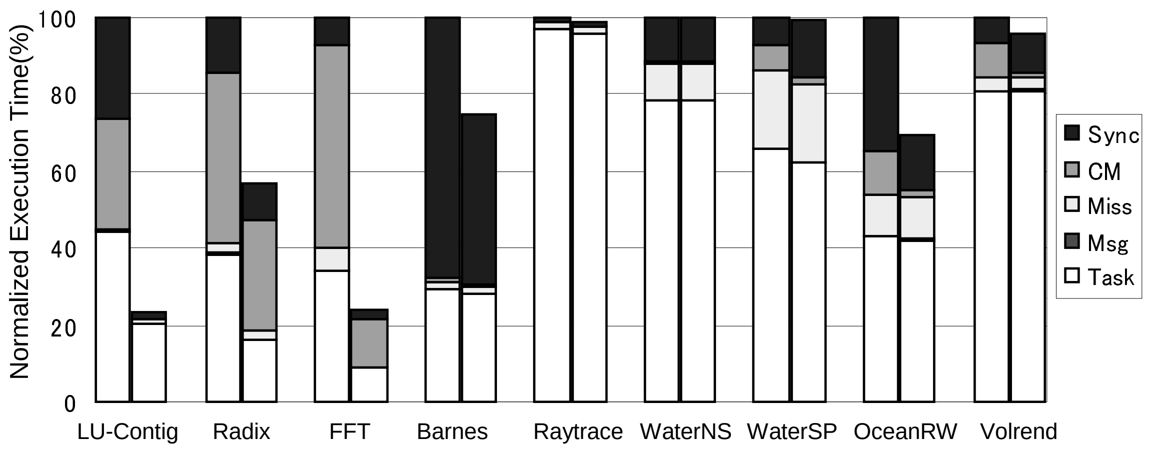


Figure 5.3: Effects of shared-write optimization for UDSM/SSS–CORE (executed on 16 processors).

Inserting a consistency-management routine in a loop causes a large overhead of procedure call and reduces memory access locality. Therefore, this optimization reduces Task time. By this optimization, not only the CM time but also the Sync time is reduced. This is because the synchronization overheads are reduced when the network traffic is alleviated.

5.6 Effects of Protocol Optimization

In Radix, a series of integer keys are sorted in ascending order. In the permutation phase of the sorting, shared-read accesses are coarse-grained but shared-write accesses are fine-grained and scattered. This situation is shown in Figure 5.4. Therefore, severe false sharing occurs at shared writes in this phase.

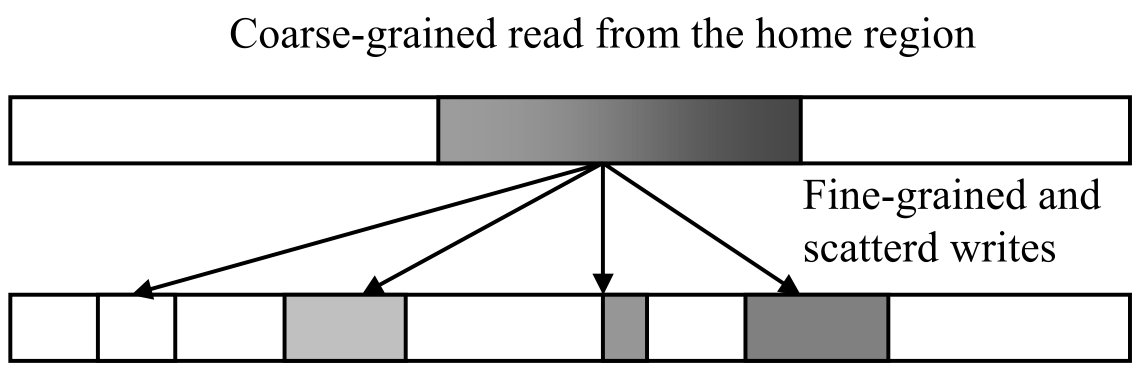

Coarse-grained read from the home region
Fine-grained and scatterd writes

Figure 5.4: Permutation phase.

The codes of this phase is shown in Figure 5.5. `key_from` and `key_to` reside on the shared region. Accesses to `key_from` are coarse-grained, and accesses to `key_to` are fine grained and scattered. These scattered write-accesses are utilized by the RCOP. `rank_ff_mynum[this_key]` is recognized as the continuous array variable. The RCOP also finds that unsigned int `this_key` is less than `bb(=radix-1)` and the shared-write dominates the modification to the `rank_ff_mynum[this_key]`. Therefore, the shared-writes can be coalesced by using the initial values and final values of `rank_ff_mynum[this_key]`.

```
      /* put it in order according to this digit */

      for (i = key_start; i < key_stop; i = i + 1) {
        this_key = key_from[i] & bb;
        this_key = this_key >> shiftnum;
        tmp = rank_ff_mynum[this_key];
        key_to[tmp] = key_from[i];
        rank_ff_mynum[this_key] += 1;
      }   /*  i */
```

Figure 5.5: Code for the permutation phase.

Furthermore, the home-only protocol reduces cache-misses at these shared-writes. In UDSM, checking codes for accesses to `key_to` are omitted automatically. The optimized output code-segment for UDSM/SSS–CORE is shown in Figure 5.6. For ADSM/SSS–CORE, system calls that validate pages containing `key_to` are inserted manually.

The home-protocol effects on Radix under ADSM/SSS–CORE are shown in Figure 5.7. The computation of speed-up ratios is based on the times for the sequential programs (not parallel 1PE). "w/o HO" means that RCOP does not perform home-only protocol optimization. "w/o CO" means that RCOP does not perform optimization such as coalescing, fusion and redundant index elimination. "Opt" means that RCOP performs all the optimization.

In "w/o CO" the overheads for consistency-management routines are comparatively large, but they decrease when the number of processors is increased. Therefore, scalability is obtained. In "w/o HO", in contrast, the performance is not improved even when the number of processors is increased. The reason for this is that fetch-on-write incurs the severe false sharing and this causes the large amount of communication traffic. As Jiang *et al.* point out [43], Radix is a difficult application on SVM because of the large amount of communication traffic and contention. The run-time support such as twin/diff mechanism does not solve this problem. The compiler support only solves it.

In FFT, source and destination matrices are distributed among processors so that each processor is the home-processor of contiguous set of n/p rows, where n is the

```
      CheckChunkSharedPageTable(key_from[i],
                                (key_stop-key_start)*sizeof(int));
      {
         int _c1;
         for (_c1 = 0; _c1 < radix; _c1 += 1)
           init_rank_ff_mynum[_c1] = rank_ff_mynum[_c1];
       }
      for (i = key_start; i < key_stop; i = i + 1) {
        this_key = key_from[i] & bb;
        this_key = this_key >> shiftnum;
        tmp = rank_ff_mynum[this_key];
        key_to[tmp] = key_from[i];
        rank_ff_mynum[this_key] += 1;
      }
      {
         int _c1;
         for (_c1 = 0; _c1 < radix; _c1 += 1)
           CONSISTENCYMANAGECODE
           (&((int *)key[to])[init_rank_ff_mynum[_c1]],
            (rank_ff_mynum[_c1] - init_rank_ff_mynum[_c1])
            * sizeof (int));
       }
```

Figure 5.6: Output code for permutation.

size of the matrix and p is the number of processors. In the modified transpose, each processor reads from its local set of rows and writes an $\frac{\sqrt{n}}{p}$ by $\frac{\sqrt{n}}{p}$ sub-matrix to each of the other processors. This situation is shown in Figure 5.8. Therefore, the heavy cache-misses occur at the shared-writes. These overheads, however, are eliminated by the home-only protocol.

Figure 5.9 shows the codes for this phase. `src` and `dst` reside on the shared region. Accesses to `src` are coarse-grained, and Accesses to `dst` are fine grained and scattered. However, this scattered write-accesses are coalesced by the RCOP. In UDSM, checking codes for accesses to `dst` are omitted automatically. The output code-segment for UDSM/SSS–CORE system is shown in Figure 5.10. For ADSM/SSS–CORE system, system calls that validate pages containing `dst` are inserted manually.

The home-protocol effects on FFT under ADSM/SSS–CORE are shown in Figure 5.11. Note that the computation of the speed-up ratios is based on the time for the sequential program (not parallel 1PE). The meanings of “w/o HO”, “w/o CO” and

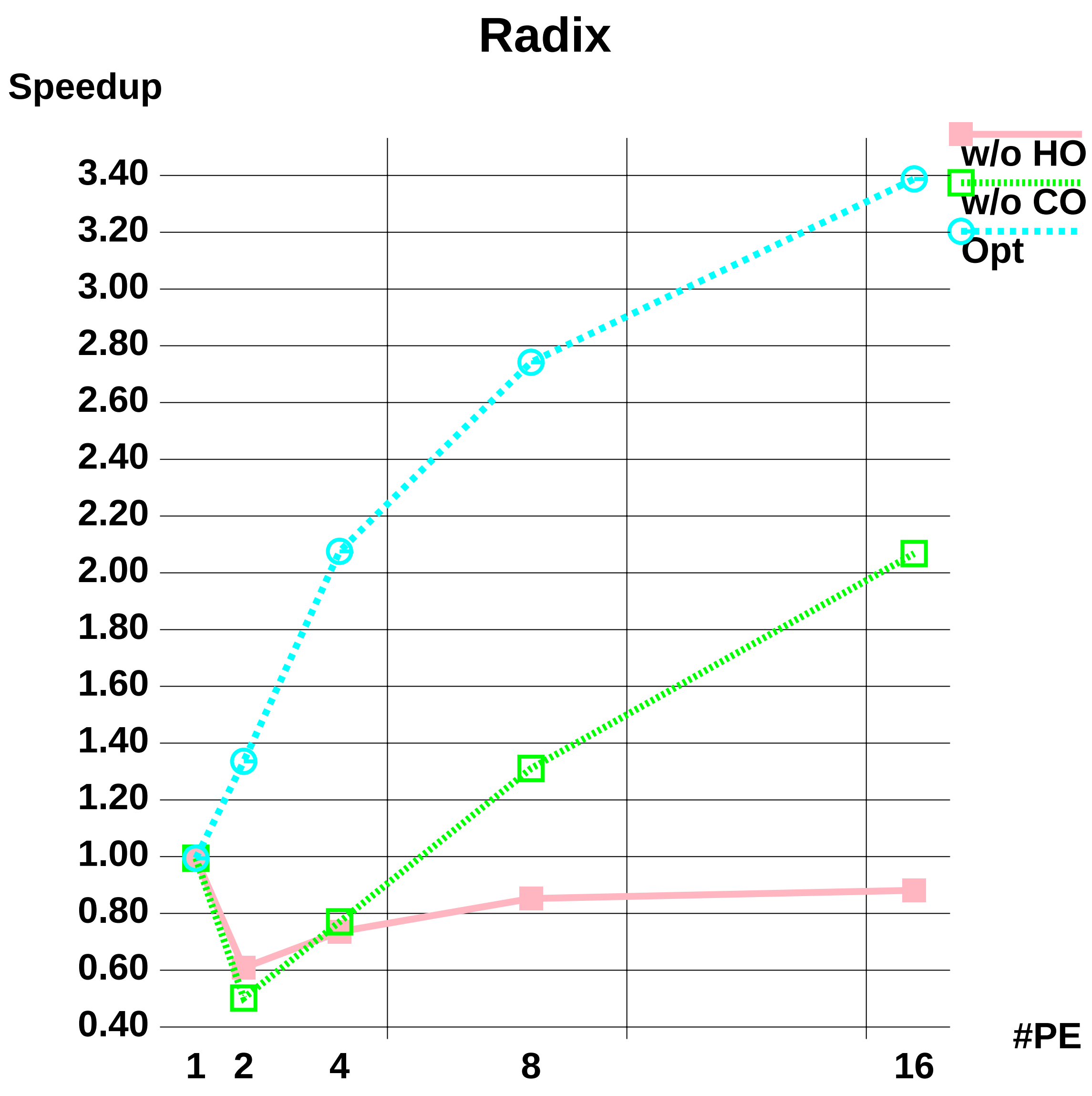


Figure 5.7: Effects of protocol optimization for Radix.

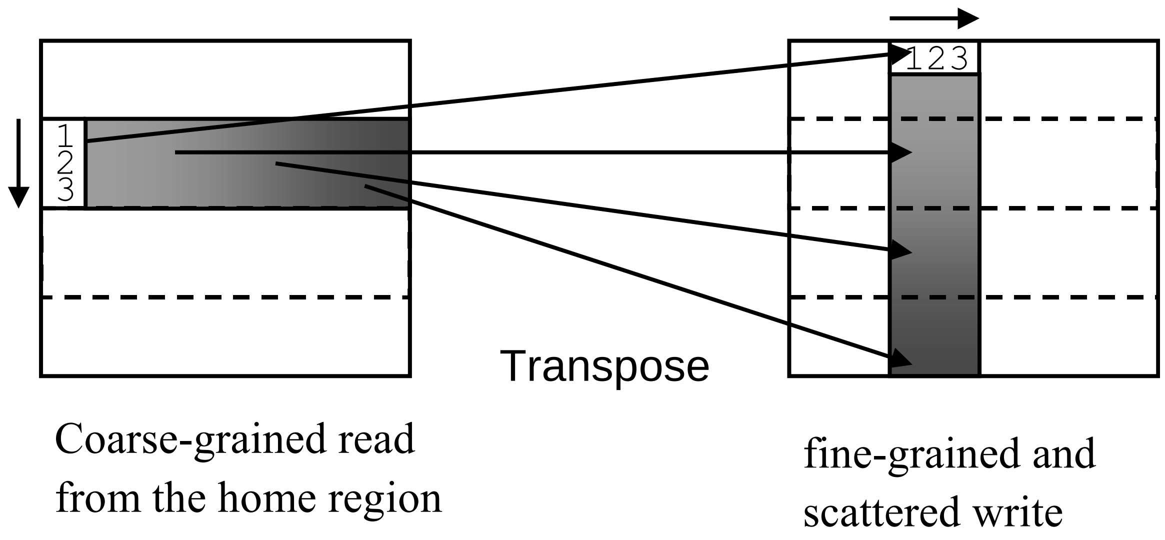

1
2
3
123
Transpose
Coarse-grained read from the home region
fine-grained and scattered write

Figure 5.8: Transpose phase.

```
    blksize = MyLast - MyFirst;
    numblks = (2 * blksize) / num_cache_lines;
    if (numblks * num_cache_lines != 2 * blksize)
      {
        numblks = numblks + 1;
      }
    blksize = blksize / numblks;
    firstfirst = MyFirst;
    row_count = n1 / P;
    n1p = n1 + pad_length;
    for (l=0;l<P;l = l + 1) {
      v_off = l*row_count;
      for (k=0; k<numblks; k = k + 1) {
        h_off = firstfirst;
        for (m=0; m<numblks; m = m + 1) {
          for (i=0; i<blksize; i = i + 1) {
            v = v_off + i;
            for (j=0; j<blksize; j = j + 1) {
              h = h_off + j;
              dest[2*(v*n1p+h)] = src[2*(h*n1p+v)];
              dest[2*(v*n1p+h)+1] = src[2*(h*n1p+v)+1];
            }
          }
          h_off += blksize;
        }
        v_off+=blksize;
      }
    }
```

Figure 5.9: Code for the transpose phase in FFT.

```
blksize = MyLast - MyFirst;
numblks = (2 * blksize) / num_cache_lines;
if (numblks * num_cache_lines != 2 * blksize)
  {
    numblks = numblks + 1;
  }
{
  int j;
  int h_off;
  int m;
  int l;
  for (l = 0; l < P; l += 1)
    for ((m = 0, h_off = MyFirst); m < numblks;
         (m += 1, h_off += (MyLast - MyFirst) / numblks))
      for (j = 0; j < (MyLast - MyFirst) / numblks; j += 1)
        CheckChunkSharedPageTable (&src[((((l * (n1 / P)) * 2
                                         + (h_off* pad_length) * 2)
                                         + (h_off * n1) * 2)
                                         + (j * pad_length) * 2)
                                         + (j * n1) * 2],
                                    ((((MyLast - MyFirst) / numblks)
                                    * numblks) * 2)
                                    * sizeof (double));
}
blksize = blksize / numblks;
firstfirst = MyFirst;
row_count = n1 / P;
n1p = n1 + pad_length;
for (l = 0; l < P; l = l + 1)
  {
    v_off = l * row_count;
    for (k = 0; k < numblks; k = k + 1)
      {
        h_off = firstfirst;
        for (m = 0; m < numblks; m = m + 1)
          {
            for (i = 0; i < blksize; i = i + 1)
              {
                v = v_off + i;
                for (j = 0; j < blksize; j = j + 1)
                  {
                    h = h_off + j;
                    dest[2 * (v * n1p + h)] = src[2 * (h * n1p + v)];
                    dest[2 * (v * n1p + h) + 1] = src[2 * (h * n1p + v) + 1];
                  }
              }
            h_off += blksize;
          }
        v_off += blksize;
      }
  }
{
  int i;
  int v_off;
  int k;
  int l;
  for (l = 0; l < P; l += 1)
    for ((k = 0, v_off = l * (n1 / P)); k < numblks;
         (k += 1, v_off += (MyLast - MyFirst) / numblks))
      for (i = 0; i < (MyLast - MyFirst) / numblks; i += 1)
        CONSISTENCYMANAGECODE (&dest[(((MyFirst * 2
                                    + (v_off * pad_length) * 2)
                                    + (v_off * n1) * 2)
                                    + (i * pad_length) * 2)
                                    + (i * n1) * 2],
                              ((((MyLast - MyFirst) / numblks)
                               * numblks) * 2) * sizeof (double));
}
```

Figure 5.10: Output code for the Transpose.

"Opt" are the same as explained for the above Radix experiment.

When the RCOP does not perform loop-level optimization, the overheads for shared-write operations are relatively high. In "w/o CO" the execution time of parallel 1 PE is about twice as long as the sequential time. These overheads are, however, alleviated as the number of processors increases. In "w/o HO" the performance is quite poor. The reason for this is that cache-misses at fetch-on-write introduce the unnecessary data transfer since the communication granularity is fixed 4K bytes.

These results clearly show that the performance of parallel shared-memory applications is influenced by the applications' access patterns. The programming style avoiding cache-miss improves parallel performance. The RCOP supports the program with fine-grained access patterns effectively as long as the read-accesses are coarse grained and the write-accesses are fine-grained and scattered.

5.7 Effects of Shared-Read Optimization

The effects of the RCOP optimizations for checking codes on 16-processor execution under the UDSM/SSS–CORE system are shown in Figure 5.12. The left bar of each program is the execution time without checking code optimizations (base time). The right bar is that with the optimizations. Note that all the other optimizations are performed to both. Execution time is normalized by the base time.

Shared-read optimization reduces the overheads for cache-state checking routines. Inserting a procedure for cache-state checking routines causes instruction overheads and reduces memory access locality. Checking optimization, therefore, reduces the computation time. Consequently, "Task" time is reduced most by checking optimization. Because of load balancing, "Sync" time is also reduced by checking optimization.

Because the LU-Contig, FFT, and Ocean-RW programs contain regular memory accesses in the loops, their execution times are reduced from 30% to 45% by the proposed loop-level optimization techniques. Although Water-SP and Water-NS have irregular access patterns, relaxed fusion and coalescing optimization are effective, and task time is reduced about 20 %.

In the Barnes program, one process creates the octree of the particles sequentially.

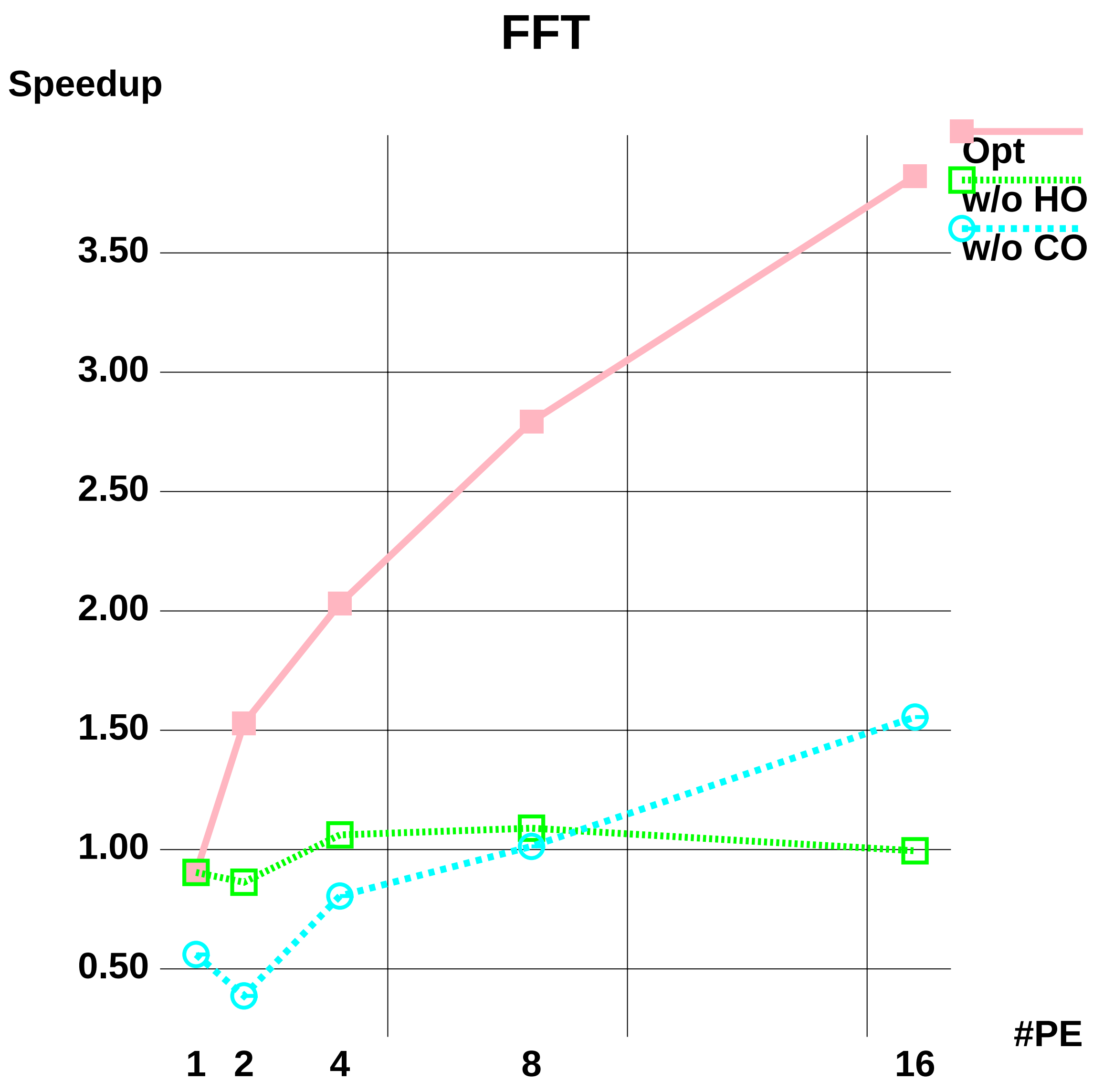


Figure 5.11: Effects of protocol optimization for FFT.

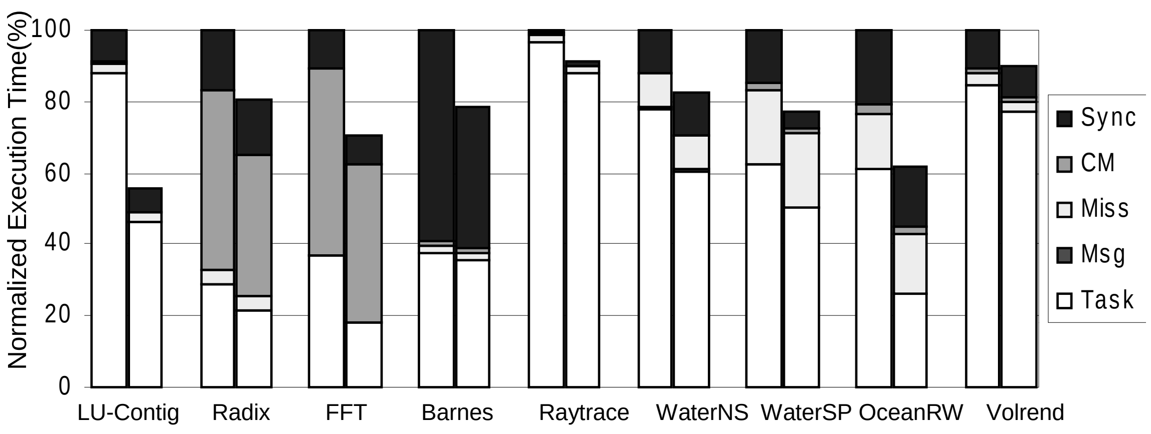


Figure 5.12: Effects of shared-read optimization for UDSM/SSS–CORE (executed on 16 processors).

In this time, the other processs waits at the barrier synchronization. Therefore, as our optimization improves the tree-build time (i.e., "Task" time) of one processor, the "Sync" times of other processors are reduced. That is to say, load-balancing is improved.

The Raytrace and Volrend programs have originally high task ratios and the overheads for shared-memory access are not bottlenecks for parallel execution. The effects of optimizations are less than 10%. As a whole, RCOP optimizations for checking codes have good effects on these nine programs.

5.8 Effects of Run-Time Optimization

The effects of run-time optimization(i.e., packet combining) on 16-processor execution under the UDSM/SSS–CORE system are shown in Figure 5.13. The left bar of each program is the execution time without packet-combining (base time). The right bar is that with the packet-combining. Otherwise, all the optimizing techniques are applied to both. In Ocean-RW, fine-grained shared writes not merged statically are often issued between the middle-grained shared-writes. Therefore, the combining optimization produces satisfactory results. "CM" time and "Sync" time are reduced

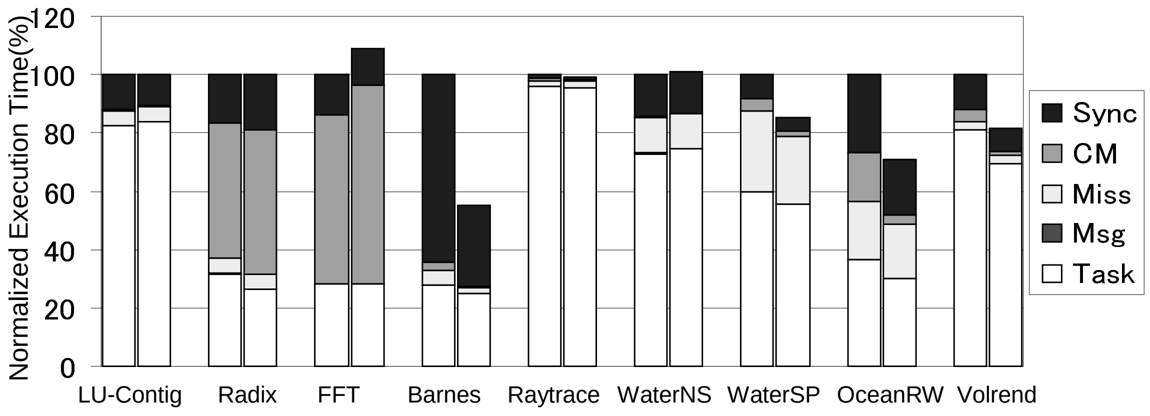


Figure 5.13: Effects of packet combining for UDSM/SSS–CORE (executed on 16 procs)

by 20 %.

In Barnes, fine-grained shared-writes are often issued at the sequential tree-building phase. The combining optimization improves "Task" time of tree-building process and "Sync" times of other waiting processes. Therefore, the combining optimization improves the load-balancing and "Sync" time are improved by 40%.

The response times for a cache-miss (i.e., "CM" time) in Barnes, Water-SP, Ocean-RW and Volrend are reduced. This is because the combining optimization reduces the number and amount of communication. Of course, the total execution times of these applications are reduced.

Packet-combining has no effect on LU-Contig or Radix. The RCOP merges cache-coherence management routines into coarse-grained ones. Hence, it is not necessary to combine packets dynamically in these applications.

The total execution times of Water-NS and FFT are increased by packet-combining. This data thus shows that combining optimization does not always have good results. When the packet-combining is done, the number of packets is reduced and communication overheads are decreased. That is, communication overheads are reduced. But, the run-time system needs to handle write histories and the number of data copies is increased in comparison with that of direct data transfer. That is to say, the instruction overheads are increased. There is a trade-off.

In FFT, the amount of shared writes is quite large compared with the amount of computation. Furthermore, frequently issued shared-write region is middle-grained, about 1KB. The written data is therefore first copied into the combiningbuffer whose size is 1.4KB. However, the next written data with the same size cannot be copied into the combiningbuffer because the buffer will overflow. The contents of combiningbuffer are therefore flushed before the new written contents are copied into it. This process causes overheads.

The experimental results show that the effect of packet combining depends on memory-access patterns of applications and the size of combiningbuffer (i.e., the packet-size).

5.9 Parallel Performance

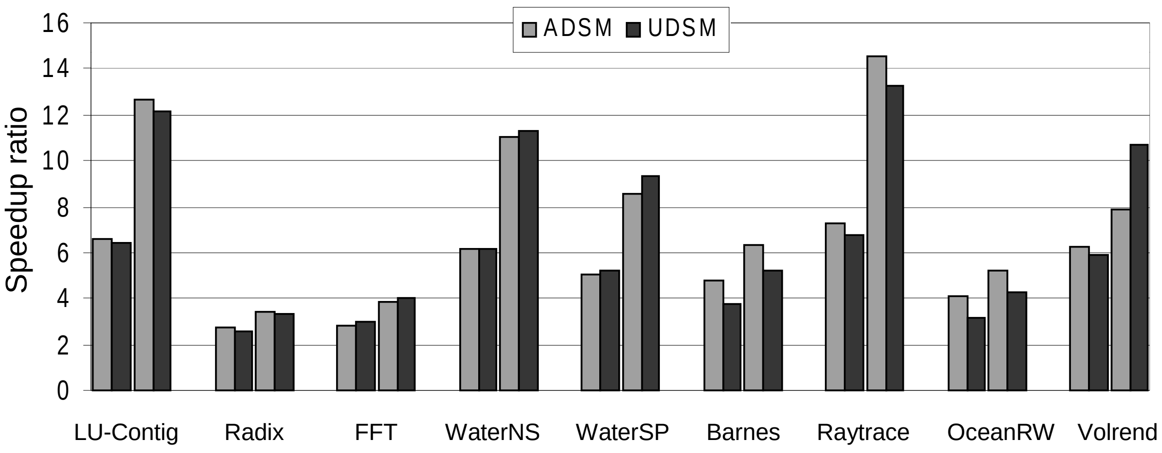


Figure 5.14: Speed-up ratios on 8 and 16 processors.

The speed-up rations for the nine programs with the proposed optimization on 8 and 16 processors are shown, for the ADSM/SSS–CORE system and the UDSM/SSS–CORE system, in Figure 5.14. The computation of the ratios were based on the times of the sequential programs. Overall, both systems had very high speed-up ratios for the LU-Contig, Water-NS, Water-SP, Raytrace and Volrend programs. The Radix,

Table 5.6: Average time breakdowns(sec) for 16-processor execution.

program	scheme	Sync	CM	Miss	Task
LU-Contig	ADSM	3.907	0.035	1.795	28.573
	UDSM	3.721	0.041	1.873	30.171
Radix	ADSM	0.460	0.866	0.083	0.504
	UDSM	0.322	1.063	0.089	0.479
FFT	ADSM	0.712	2.886	0.000	1.341
	UDSM	0.547	2.948	0.000	1.229
Barnes	ADSM	4.614	0.115	0.349	3.743
	UDSM	5.389	0.121	0.290	4.862
Raytrace	ADSM	0.183	0.021	0.906	10.620
	UDSM	0.192	0.030	0.282	12.714
Water-NS	ADSM	8.015	0.070	4.895	30.325
	UDSM	7.003	0.065	4.062	31.338
Water-SP	ADSM	0.390	0.164	1.947	3.688
	UDSM	0.324	0.114	1.537	3.728
Ocean-RW	ADSM	1.300	0.161	0.988	1.434
	UDSM	1.422	0.172	1.052	2.258
Volrend	ADSM	0.0719	0.003	0.146	0.277
	UDSM	0.0369	0.00515	0.0137	0.325

FFT, Barnes and Ocean-RW programs showed good scalabilities on both systems. To make it easier to understand where the time goes, Table 5.6 lists the results of breaking down the program execution times on the 16 processors. The meanings of “Sync”, “CM”, “Miss”, “Task” are the same as in Section 5.4. It should be noted that when the home-only protocol is used, “Task” also includes the time for a system call needed to implement the home-only protocol for the ADSM/SSS–CORE system. It should also be noted that “Task” includes not only the original computation time but also the cache-state checking overheads in the UDSM/SSS–CORE system.

The LU-Contig program uses a tiled data partitioning with each tile of the matrix allocated as a contiguous region. Many shared-read and shared-write operations are performed to each tile. Their cache-management codes are combined into one code by interprocedural shared-access set calculation. The data listed in Table 5.6 shows that the performance difference between ADSM/SSS–CORE and UDSM/SSS–CORE is due to the UDSM/SSS–CORE overhead for checking the cache-state.

The FFT and Radix programs are challenging applications for software DSM because they potentially require a large communication bandwidth because of false sharing [29, 43, 76, 89]. Nonetheless, both ADSM/SSS–CORE and UDSM/SSS–CORE systems obtain good scalability by using the home-only protocol. The performance difference between these two systems is due to the overhead for the system calls needed to implement the home-only protocol in ADSM/SSS–CORE.

In the Raytrace program, there is little sharing. Raytrace uses distributed task queues to share workloads. The lock-home specification for task queues reduces the synchronization overhead. It is a highly parallel program. The data listed in Table 5.6 shows that the performance difference between ADSM/SSS–CORE and UDSM/SSS–CORE is due to the checking overhead.

The Barnes program solves an N-body problem by using the Barnes-Hut $O(n \log n)$ algorithm. It contains many shared accesses that are not statically merged, and we can see from the data listed in Table 5.6 that checking overhead of UDSM/SSS–CORE is large even with 16-processor execution. Building the tree of particles sequentially can cause a load imbalance. Therefore, the synchronization overhead in both systems is large.

The Water-NS program computes the forces and potentials in a system of water molecules. All the molecules are allocated on a shared 1-D array, and the array is decomposed so that each processor modifies a contiguous region. False sharing occurs only at boundaries. This program contains memory accesses for which loop-level relaxed optimizations are performed. Both systems produce high speed-up ratios.

The Water-SP program computes the same problem by spatially partitioning a uniform 3-D grid of cells. The molecule data structures and cell data structures, each with size less than 1 KB, are allocated separately. The ADSM/SSS–CORE system transmits unnecessary data at cache-misses because the cache size is 4 KB fixed. The UDSM/SSS–CORE system, in contrast, sends much less unnecessary data because the cache size is 1 KB. This is why UDSM/SSS–CORE provides better performance.

The Ocean-RW program partitions a grid into blocks of rows. The memory accesses are contiguous in the blocks, and coalescing is performed for these accesses. However, coalescing optimization is not always applied to memory accesses to data in near-neighbor blocks because these accesses are not contiguous. These accesses can sometimes cause cache-misses. Therefore, cache-miss and synchronization overheads are dominant in both systems. In the UDSM/SSS–CORE system, checking overhead is not negligible. This is the reason ADSM/SSS–CORE performs better than UDSM/SSS–CORE.

The Volrend program computes images by using ray-casting techniques. The image planes are distributed among processors and the the partitioned planes are not as large as the page-size. False sharing, therefore, occurs when the processor writes to the image planes. In UDSM/SSS–CORE, this is avoided by the home-only protocol. In ADSM/SSS–CORE, however, the overheads for the incurred system calls that validate theses pages become quite large. Therefore, in this application, applying the home-only protocol slightly degrades the performance. The ADSM/SSS–CORE system, hence, does not use home-only protocol. It follows from this that cache-miss overheads in the ADSM/SSS–CORE system are larger than those in the UDSM/SSS–CORE system (Table 5.6).

In the five programs for which the ADSM/SSS–CORE system performed better than the UDSM/SSS–CORE system, the main source of overhead is checking the (software-)cache-state. Checking the cache-state can cause hardware cache-misses when the valid bit table is loaded. The UDSM/SSS–CORE system performed well for programs that have middle-grained remote-memory accesses. The block size of ADSM/SSS–CORE must be equal to the size of a memory-page (4KB). Unnecessary data can therefore be transmitted when a cache-miss occurs. The block size, of course, has an influence on the waiting time for a cache-miss. For Raytrace, Water-NS and Water-SP, the block size of the UDSM/SSS–CORE system is less than the packet size, and the waiting time is shorter on the UDSM/SSS–CORE system than on the ADSM/SSS–CORE system. Owing to checking code optimization, for Barnes and Volrend, the waiting time for a cache-miss is also shorter on the UDSM/SSS–CORE system than on the ADSM/SSS–CORE system.

5.10 Summary and Discussions

This work has demonstrated by using SPLASH-2 benchmark suite that the combination of optimizing compiler and user-level cache coherence mechanism is effective. Shared-write optimizations eliminate mainly communication overheads and shared-read optimizations reduce the checking overheads for every application. The packet combining optimization is effective for applications on which compiler optimization does not work well. The home-only protocol is indispensable for applications with fine-grained scattered shared-writes.

Our approach deals with explicit parallel shared-memory programs. Explicit parallelism can be used to express applications that is difficult for data parallel language. The experimental results show that the proposed optimizing techniques yield high performance for applications (such as LU-Contig, FFT and Volrend) that can be expressed in data parallel language and parallelized. It follows from this that the Software DSM system exploiting application's semantics can not only accept wider class of applications but also provide high performance with parallelizable applications.

In general, there are more opportunities for optimizing communications and executions under UDSM than under ADSM. This work has confirmed that the performance of ADSM/SSS–CORE is limited by the communication of unnecessary data in programs with fine-grained data accesses, while that of the UDSM/SSS–CORE is limited by the instrumentation overhead. Of course, it is not easy to always reduce the instrumenting overheads for any programs. This work, however, has demonstrated that the optimization methods developed for directly analyzing applications and exploiting applications' semantics are effective in reducing these overheads to a range of 2.5% to 21% for nine applications with various access patterns.

Compare these results with another fully user-level DSM, called Shasta[79, 78]. Shasta is a Software DSM that supports fine-grained access to shared-memory by inserting cache-management codes before shared-loads and shared-stores. Namely, the Shasta resembles the Hardware DSM mechanism. The compiler for Shasta analyzes not the source code but the binary code of an application, and support variable coherence granularities within a single application. It also performs various optimizing techniques, such as batching and invalid flag techniques.

Batching is a technique merging checks of multiple loads and stores only when their base registers are the same and their offsets are less than or equal to the line-size (64-128 bytes). The invalid flag technique is to skip the checking of the cache state if the data is valid. Before state-checking, the processor at first loads the value, and compares the value with the particular "flag" value. If the loaded value is not equal to the flag value, the loaded value is valid and the code continues to run immediately. The checking routine is called only when the loaded value is equal to the flag value.

The Shasta's result reported in Ref [77] are listed in Table 5.7. The experiment environment is AlphaServer 4100. The processor on AlphaServer 4100 is 300 MHz 21164, which has 16 KB on-chip I-cache and D-cache, 96 KB on-chip combined second-level caches and 2 MB board-level cache. The input-size of Barnes and Ocean are different from those in the present work, but the checking overhead percentages are not affected by the input-size.

This comparison shows that the present approach directly analyzing source codes produces better results than does Shasta's approach, except with Volrend and Barnes programs. The present approach is found most effective in regular applications with coarse-grained shared access patterns such as LU-Contig, and to produce satisfactory

Table 5.7: Checking overheads found in the Shasta system (from Ref. [77])

	problem size	sequential time	with Shasta miss checks	miss check overhead
Barnes	16K particles	9.05s	9.92s	9.6%
LU-Contig	2048^2 doubles	140.9s	176.5s	25.3%
Ocean	514^2 ocean	11.04s	13.29s	20.5%
Raytrace	balls4	71.53	79.59	11.3%
Volrend	head	1.62	1.76	8.6%
Water-Ns	4096 molecules	125.9	147.3	17.0 %
Water-Sp	4096 molecules	15.94	18.12	13.7 %

results for irregular applications with fine-grained synchronization and fine-grained sharing patterns (such as Water and Raytrace). The relaxed coalescing optimization and fusion optimization contribute to the superior performance of this approach.

The present approach causes overheads about 20 % for Barnes and Volrend, while Shasta's approach makes the overheads for theses applications about 10%. This difference is due to these applications' fine-grained shared accesses that are not translated into middle-grained/coarse-grained ones. For these applications, each checking cost is the key to performance. Optimization like invalid flag techniques used in Shasta can be suitable for these applications. but, this requires the flag value to actually be written to the shared data at coherence management. This reduces the advantages of optimizations like coalescing and fusion. Therefore, it is concluded that more elaborate checking techniques are required for the UDSM scheme or that ADSM scheme should be used for these kinds of applications.

Chapter 6

Related Work

6.1 Compiler-Assisted DSM

There are three kinds of projects combining compilers and Software DSMs.

1. Software DSM as a target for a parallelizing compiler

   Keleher *et al.* [47] combine the Stanford SUIF compiler system [83] and software DSM. If the compiler can detect communication patterns, the update coherence protocol is selected. If the compiler can detect reduction operation, reduction is executed locally and the local-results are sent in a barrier-arrival message. A barrier master performs all reductions and updates the value of global data. Lu *et al.* [55] use this approach to support irregular computation.

   Boyle et al. [72] utilize features of SPMD program to compute the minimum number of barrier insertion by computing cross processor dependence.

   These results clearly show that software DSM schemes assisted by these optimizing techniques provide good performance for parallelizable applications. These techniques, however, are applied only to parallelizable applications. The class of applications that allow precise analysis is limited. Our approach deals with explicitly parallel programs, more wider classes of applications. Furthermore, our experiment shows that our approach provides high performance for applications that can be expressed in a data-parallel language.

2. Association between shared data and synchronization

   Programmers specify coherence protocol for each shared data [11, 80, 18]. However, they have to rewrite existing programs. Furthermore, it require considerably programming efforts to write these programs from scratch. Our approach deals with applications based on LRC model. It is easy to write programs based on LRC model because LRC is a natural extension of sequential consistency.

   Various optimizations using user-program information are performed. False sharing does not occur in this approach, but the approach always causes overheads of packing and unpacking messages are always caused.

   In Midway [87], software dirty bits are used to detect modifications to shared data and a software dirty bit is associated with each cache line in the system. The compiler generates codes to flip the associated dirty bit on each shared write. No loop-level optimization is performed. The overheads is proportional to the number of shared-writes. Our approach performs the loop-level optimizations for shared-writes and reduces the overheads as much as possible. Our approach reduces the overheads for shared-write operations to a range of 0.017% to 8.9% for various applications.

   In CRL [44], programmers are forced to insert calls to delimit operations on shared-data. Not only allocations but also mapping, read and write are explicitly specified by the programmers. That is to say, all the cache-management routines are explicitly inserted by the programmer. Although association between shared data and synchronization is not explicitly specified by programmers, this programming style requires the much programming effort. Our approach handles the traditional shared-memory programming style and shared-memory accesses are detected and optimized by the optimizing compiler.

3. Direct analysis of explicitly parallel programs

   Several software DSMs that use an optimizing compiler analysing explicitly parallel programs directly have been reported. Tempest [74] provides the programmer with message communication mechanism to construct shared memory protocols and it is implemented on a Thinking Machines CM-5, called Blizzard-S [19]. Blizzard-S rewrites the executable files to insert a state-table lookup before every shared-memory access. However, The Blizzard-S, however, uses sequential consistency protocol and uses a single-writer protocol as a base protocol. The protocol optimization is performed only by the programmers. This requires the same efforts as describing packing/unpacking procedures directly.

Shasta [79], Blizzard-S's successor, also supports fine-grained coherence in full software by rewriting application binary to intercept shared loads and shared stores. The mechanism of Shasta resembles to the Hardware DSM mechanism. A flag value technique is used to skip the checking of the cache state if the data is valid, and a batch technique is used to merge multiple checking codes. Because Shasta does not perform loop-level, interprocedural optimization, it requires large-bandwidth and low-latency network. The goal of Shasta is to reduce instrumentation overheads as preserving fine-grained shared-accesses.

Our approach changes fine-grained shared-memory accesses into coarse-grained ones while keeping the meaning of parallel programs to reduce communications overheads and instrumentation overheads. The optimizing compiler RCOP issues the cache-coherence management routines with coarse-grained size for shared-memory accesses by using source program information. The lightweight run-time system RS3 utilizes the bulk data transfer mechanism through compiler-inserted routines. Our approach, therefore, is more suitable for computer clusters using commodity communication hardware. This is because such commodity network hardware has non-negligible overhead for communication.

Dwarkads ***et al.*** [26] performed compiler analysis on explicitly parallel programs[1] to improve their performance on Software DSM. By using regular section analysis, they detected the data-access patterns at compile-time and used them to help the run-time system aggregate communication and synchronization and reduce consistency-management overheads. However, their compiler analysis was

[1]Fortran programs

performed only intraprocedurally and was also limited by a conditional code. Furthermore, access patterns analysed were limited to induction variables. Each shared array must be page-aligned to ease the compiler analysis and to reduce consistency-management overheads, and this requires a large amount of memory. Our approach, however, deal with C programs handling pointers and does not require page-alignment because it uses precise points-to analysis. The optimizations are performed interprocedurally, and coalescing optimizations can be performed using not only induction variables but also continuous index variables.

6.2 Comparative Study of Software DSM Schemes

This thesis attempts to compare the page-based scheme with segment-based scheme by running fully optimized real applications. Of course, much comparative studies about software DSM have been done as follows.

Adve *et al.* [1] compared lazy release consistency with entry consistency. They have concluded that compiler instrumentation has worse performance than twinning (comparing the current version of shared data with an older version dynamically). But their compiler uses only the dirty-bit mechanism [11]. It is difficult to say that the programs are fully optimized. Their compiler does not utilize the coarse-grained / middle-grained shared-memory-accesses along with lazy release consistency.

Cox et al. [23] compared hardware approach with software approach. They used SGI 4D/480 and TreadMarks running on an ATM network of DEC Stationr5000/240s. The point of this comparison is that two platforms are all the same execpt for shared-memory implementation. That is, processors, caches and compilers are the same. They reported that SGI 4D/480 outperforms TreadMarks for applications with frequent synchronization and communication and that TreadMarks outperforms the SGI for applications with high memory bandwidth requirements. Our comparison also uses the same platform.

Amza et al. [8] proposed adaptive protocols that dynamically choose between single-writer and multiple-writer, based on write-write false sharing and/or write granularity. The protocol avoids the worst case behavior of single-writer protocol (the effect of ping-pong effect) and multiple-writer protocol (the diff accumulation problem). They compared the adaptive protocol with non-adaptive protocols (single-writer-only protocol and multiple-writer-only protocol). They said that adaptive protocol outperforms the non-adaptive protocols. Although our approach supports multiple-writer protocol, there are no diff accumulation problems because we introduce SAURC protocol, that is, home-update protocol.

Dwarkadas ***et al.*** [27] examined the performance tradeoffs between a coarse-grained approach, that is, page-based DSM (Cashmere) and fine-grained approach, an instrumentation-based DSM (Shasta) on the Memory Channel network. The instrumentation-based DSM provided robust performance. The instrumentation-based DSM is assisted by the optimizing compiler. However, this page-based DSM is not supported by the optimizing compiler. This situation is against the page-based DSM.

Little is known about the performance tradeoffs between the page-based scheme and the segment-based scheme when both are fully assisted by the optimizing compiler. Moreover, our goal is to examine the role of optimizing compiler in software DSM schemes through the comparison. The optimization techniques developed in this work put the segment-based DSM scheme to practical use. These optimization techniques are effective in reducing instrumentation overheads to a range of 2.5 % to 21 % for various applications.

6.3 Interprocedural Optimizing Compiler

In the 1970s, a series of basic dataflow analysis techniques were proposed such as global subexpression elimination[21], interval analysis [5] and interprocedural analysis [6]. The basic concept of optimizations in the RCOP is, of course, based on these excellent works.

The conventional approach [36] to interprocedural parallelization analysis is to determine the sections of arrays that are produced/consumed by each procedure call contained in loops. These techniques were shown to be effective in parallelizing linear algebra libraries.

A framework for interprocedural analysis and transformation (FIAT) [33, 34] has been proposed as a general environment for interprocedural analysis. The FIAT uses region-based analysis [2] and the analysis is performed in two passes over the program. The FIAT partitions calling context information into equivalence classes and eliminates irrelevant information. For each equivalence class, dataflow information is replicated. This is called selective procedure cloning. FIAT has been implemented as a part of SUIF compiler system and used for array dataflow analysis (such as array privatization and array reduction) [32]. Our approach for detecting induction variables and continuous variables utilizes this framework.

Interprocedural symbolic analysis has been used to solve symbolic constant propagation, generalized induction variable substitution and loop invariant computations detection. These problems have important roles in dependence analysis that is the basic component of a parallelizing compiler. It is appropriate to consider that our approach for detecting induction variables and continuous variables also utilizes the symbolic-analysis framework. The interprocedural symbolic analysis has been implemented in Parafrase-2 [31], and the results of an experiment using Perfect Benchmarks show that the interprocedural symbolic analysis is a powerful technique. This analysis, however, cannot be applied to array, pointers and abstract data types. Therefore, it cannot find continuous array variables that can be detected by using our approach.

Interprocedural points-to analysis is implemented in SUIF compiler system, and features of the analysis are described in Chapter 3 by using examples. The analysis is iterative one. It includes the effects of intraprocedural control flow and distinguishes information originating from different calling contexts by partial transfer function. Partial transfer functions summarize the points-to information in their calling contexts. The point is that they are reused unless aliases between pointers that are dereferenced are changed. Our approach uses this analysis to detect shared-access precisely.

[2]Regions correspond to loops and procedure-calls

Interprocedural partial redundancy elimination is a general framework for eliminating partial redundancies that includes traditional optimization such as loop-invariant code motion and redundant code elimination. Agrawal et al. [2, 3] uses this framework for inspector/executer scheme. They use concise, full program representation, but do not consider the possibility of aliasing. The RCOP uses interprocedural points-to analysis to solve this problem. The basic dataflow equations in their approach are more complicated than those in our system, and it is not easy to implement.

Interval-based framework for solving interprocedural dataflow equations was investigated in Burke's work [15]. It can be applied to any monotone dataflow problem. This framework is used for alias analysis in this work. We have adopted this framework to efficiently solve redundancy elimination dataflow equations for summaries of shared-memory accesses.

None of these fine analyses have been used for explicitly parallel shared-memory programs. The RCOP is the first optimizing compiler to summarize shared-memory-accesses interprocedurally by using the interval analysis framework.

Chapter 7

Conclusion and Future Direction

7.1 Conclusion

A coherent shared address space provides an attractive programming environment for parallel computing. Software DSM provides shared address space at run-time and accepts a wide range of applications, and it is easy to implement on the existing systems with commodity hardware. Optimizing methods are indispensable for improving the performance of Software DSM schemes. That is, compiler optimization, protocol optimization, run-time optimization and the interfaces that enable these optimizations are required.

We have introduced two compiler-assisted software DSM schemes as the interfaces for these optimizations. These schemes can be considered as hybrid of shared-memory and message-passing. The UDSM scheme is a segment-based and fully user-level system. The ADSM scheme is a page-based system. The UDSM provides a programmer/compiler with opportunities to perform optimization for both shared-read accesses and shared-write accesses. The ADSM, on the other hand, offers a programmer/compiler chances to perform optimization for shared-write accesses.

The purpose of compiler optimization is to generate codes reducing the communication and instruction overheads for software cache-coherence management. The approach described in this thesis exploits the application's semantics (such as loops, procedure calls) by using the relaxed coherence model, interprocedural alias information and an interprocedural redundancy-elimination framework based on interval analysis.

This approach has been implemented in an optimizing compiler developed for shared-memory parallel programs, called a “Remote Communication Optimizer” (RCOP). It performs

- interprocedural points-to analysis and
- interprocedural shared-access-set calculation by using interval analysis to solve redundancy elimination equations.

As a result,

- it detects all the shared-accesses precisely,
- it removes redundant cache-coherence management routines, and
- it merges multiple redundant cache-coherence management routines by using loop structures and procedure calls.

The RCOP exploits continuous variable information for shared-write optimizations.

To make the cache-coherence protocol optimization possible, we have developed the various methods of implementing cache-coherence mechanisms that follow LRC model:

1. History-based LRC (HLRC)
2. Software emulation of AURC (SAURC)
3. Hybrid of HLRC and SAURC

We have collected data showing that it is important to maintain a home for each cache-block and that the timestamp mechanism conventionally used to strictly preserve the partial ordering among write notices incurs relatively high synchronization costs and cache-miss costs and large memory requirements.

To make the run-time optimization, we have also proposed the lightweight run-time system for cache-coherence management, called RS3.

- It maintains a one-bit write-notice for each cache indicating whether that cache has been modified since the last barrier operation. This write-notice mechanism reduces the synchronization costs and memory requirements. Furthermore, the updated block list mechanism is used to reduce the data transfer at synchronization operations.

- It utilizes the bulk data transfer mechanism to efficiently execute coarse-grained communication through the consistency-management routines issued by the optimizing compiler.

- It performs the fine-grained communications efficiently by combining those whose destination processors are the same and transferring as many of them as possible at once.

- It utilize the remote invocation mechanism of the user-specified program to handle remote requests quickly with low overheads.

This run-time system RS3 has been implemented under a general-purpose scalable OS, SSS–CORE, on the SS20 workstation cluster connected with the Fast Ethernet (100BASE-TX), utilizing MBCF.

The proposed optimizing methods have been demonstrated effective under the ADSM/SSS–CORE and UDSM/SSS–CORE systems by using the SPLASH-2 benchmark suite. The followings are found from our experiment:

- The RCOP reduces overheads for software cache-coherence management in the ADSM/SSS–CORE system to a range of 0.017% to 8.9%. The RCOP reduces overheads for software cache-coherence management in the UDSM/SSS–CORE system to a range of 2.5% to 21%.

- Shared-write optimizations eliminate mainly communication overheads and shared-read optimizations reduce the checking overheads for every application.

- The packet-combining optimization is effective for applications on which compiler optimization does not work well.

- The home-only protocol is crucial for applications with fine-grained scattered shared-writes.
- Our proposed optimizing techniques yield high performance for applications (such as LU-Contig, FFT and Volrend) that can be expressed in data parallel language and parallelized.
- The performance of ADSM scheme is limited by the communication of unnecessary data in programs with fine-grained data accesses, while that of the UDSM scheme is limited by the instrumentation overhead. The UDSM scheme reduces false sharing and transmission of unnecessary data, both of which are potential problems in the page-based system.

The optimization effects in Software DSM have been confirmed through the evaluation of proposed frameworks. It should be noted here that the optimizing techniques developed in this thesis made it possible to put the segment-based software DSM to practical use. The general applicability of this approach should also be noted. The approach is not peculiar to ADSM and UDSM. The approach does not perform machine-dependent optimization. And the approach is, therefore, portable. It can be applied to not only to Software DSM but also to Hardware DSM.

We can see now that it is possible to construct general-purpose scalable shared-memory parallel computer with commodity hardware.

7.2 Future Direction

General Applicability of Our Approach

In this work, C language extended by PARMACS macro is used as the parallel programming language. Of course, our approach is applied to any explicitly parallel program as long as parallel constructs are recognized. Recently, OpenMP seems to be attracting wide-spread support among application developers [73]. OpenMP API provides directives that allow users to annotate a sequential program (C/C++ and Fortran) to explicitly specify parallel-execution parts. The users are responsible for ensuring that applications using OpenMP constructs execute correctly. That is, checks for dependencies, conflicts, deadlocks, race conditions are not required for OpenMP system (compilers/run-time library). Furthermore, the modification to shared data by a thread is not visible to other threads until the thread encounters synchronization directives. Therefore, our approach is straightforward applied to OpenMP.

The shared-access summary calculation described in this thesis can be used for other purposes. Let us consider Java [81]. In Java, each array access (such as `iaload`) is checked whether the index is within the array region. Array accesses are often performed in the loop. Therefore, the check per iteration causes considerable overheads. If the JIT compiler utilizes loop-level optimization methods, the index check is merged into one check.

Future Direction of Our Approach

We are now studying techniques to further reduce the compilation time of our framework. Coalescing test, fusion test and redundant index test are frequently calculated in inner loops. Therefore, we use memorization technique to store and reuse previously computed results. In our current implementation, computing the intersection / optimized-union of sets of shared-access sets ($A \bigcap B$ / $A \bigsqcup B$) requires $\mathcal{O}(|A||B|)$ time because several tests such as fusion and inclusion are performed to each shared-access set. This may lead to large compilation time when the number of shared-access sets becomes large. Therefore, we should improve these computations. In UDSM compilation, shared-read optimizations and shared-write optimizations are performed separately. If the results of shared write optimizations can be reused when performing shared-read optimizations, the compilation time will further be optimized.

We assume that commodities (e.g. Fast ethernet) are used as the communication hardware of the system. As has been noted, such commodities have non-negligible overheads for communication and hence they are poor at fine-grained communications. Furthermore, since there is a limit for the size of the sending buffer in the communication hardware, large data can not be sent over the limit at a time. Our current optimization methods attempts to issue checking routines and consistency-management routines with coarse-grained size. Therefore, they are not always optimal on our supposed platform. It is advantageous on performance to adjust the data-size of communication packets statically or dynamically.

Future Direction of Software DSM

Of course, the result of our experiment reveals that the page-based scheme is superior to the segment-base scheme in five applications out of nine. In the future, however, a segment-based scheme may outperform a page-based scheme if the difference between the speeds of processors and networks is much larger than it is now. The reason for this is that the bottleneck for a page-based scheme lies in a network because of false sharing and unnecessary data transfer, while the bottleneck for a segment-based scheme lies in a processor because of overheads for coherence management in all software.

Bibliography

[1] S. V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementations. In Proc. of the 2nd IEEE Symposium on High-Performance Computer Architecture, February 1996.

[2] G. Agrawal and J. Saltz. Interprocedural Compilation of Irregular Applications for Distributed Memory Machines. In Proc. of Supercomputing '95, December 1995.

[3] G. Agrawal, J. Saltz, and R. Das. Interprocedural Partial Redundancy Elimination and its Application to Distributed Memory Compilation. In Proc. of '95 Conf. on PLDI, pages 258–269, June 1995.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers Principles, Techniques, and Tools. Addison-Wesley, Reading, MA, 1986.

[5] F. E. Allen. Control Flow Analysis. Proc. of ACM SIGPLAN Notices, 5(7):1–19, July 1970.

[6] F. E. Allen. Interprocedural Data Flow Analysis. In Proc. of the Information Processing 74, pages 398–402, Stockholm, Sweden, August 1974.

[7] S. P. Amarasinghe and M. S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In Proc. of '93 Conf. on PLDI, pages 126–138, July 1993.

[8] C. Amza, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software dsm protocols that adapt between single writer and multiple writer. In Proc. of the 3rd HPCA, pages 261–271, February 1997.

[9] J. M. Anderson and M. S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In Proc. of '93 Conf. on PLDI, pages 112–125, July 1993.

[10] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In Proc. of the 17th ISCA, pages 125–135, May 1990.

[11] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In Proc. of the 1993 CompCon Conf., pages 528–537, February 1993.

[12] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In Proc. of the 21th ISCA, pages 142–153, April 1994.

[13] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. Portable Programs for Parallel Processors. Holt, Rinehart and Winston, Inc., 1987.

[14] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers: Design, Implementation, and Performance Results. In Proc. of Supercomputing '93, pages 351–360, November 1993.

[15] M. Burke. An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data-Flow Analysis. ACM Transactions on Programming Languages and Systems, 12(3):341–395, July 1990.

[16] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie. Overview of the ksr1 computer system. Technical report, Kendall Square Research, 1992.

[17] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In Proc. of 13th ACM Symposium on Operating System Principles, October 1991.

[18] C. Chang, A. Sussman, and J. Saltz. Object-Oriented Runtime Support for Complex Distributed Data Structures. Technical Report CS-TR-3428, University of Maryland, March 1995.

[19] I. choinas, B. Falasafi, A. R. Lebeck, S. K. Reinhardt, J. R. Laurus, and D. A. Wood. Fine-grain Access Control for Distributed Shared Memory. In Proc. of ASPLOS-VI, pages 297–306, October 1994.

[20] F. C. Chow. Minimizing Register Usage Penalty at Procedure Calls. In Proc. of '88 Conf. on PLDI, pages 85–94, June 1988.

[21] J. Cocke. Global Common Subexpression Elimination. Proc. of a Symposium on Compiler Optimization, SIGPLAN Notices, 5(7):20–24, July 1970.

[22] J. Cocke and J. T. Schwartz. Programming Languages and Their Compilers. Courant Institute of Mathematical Science. New York University Press, 2nd edition, April 1970.

[23] A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software versus hardware shared memory implementation: A case study. In Proceedings of the 21th ISCA, pages 106–117, 1994.

[24] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment and the Control Dependence Graph. ACM Transactions on Programming Languages and Systems, 13(3):451–490, October 1991.

[25] R. Das, M. Uysal, J. Saltz, and Y. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. Journal of Parallel and Distributed Computing, 22(3:462–479, September 1994.

[26] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In Proc. of ASPLOS-VII, pages 186–197, October 1996.

[27] S. Dwarkadas, K. Gharachorloo, L. Kontothanassis, D. J. Scales, M. L. Scott, and R. Stets. Comparative evaluation of fine- and coarse-grain approaches for software distributed shared memory. In Proc. of the 5th HPCA, pages 260–269, January 1999.

[28] M. Emami, R. Ghiya, and L. J. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In Proc. of '94 Conf. on PLDI, pages 242–256, June 1994.

[29] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the Performance of Clusterd Distributed Virtutal Shared Memory. In Proc. of ASPLOS-VII, pages 210–220, October 1996.

[30] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In Proc. of the 17th ISCA, pages 15–26, May 1990.

[31] M. Haghighat and C. Polychronopoulos. Symbolic program analysis and optimization for parallel compilers. Technical report tr-1237, Center for Supercomputing Research and Development, 1993.

[32] M. W. Hall, S.P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Detecting Coarse-Grain Parallelism Using an Interprocedural Parallelizing Compiler. In Proc. of Supercomputing '95, 1995.

[33] M. W. Hall, J. M. Mellor-Brummey, A. Carle, and R. G. Rodriguez. Fiat: A framework for interprocedural analysis and transformations. In Proc. of the 6th Int. Workshop on LCPC, pages 522–545. Springer-Verlag, August 1993.

[34] M. W. Hall, B. R. Murphy, S. P. Amarasinghe, S. Liao, and M. S. Lam. Interprocedural Analysis for Parallelization. In Proc. of the 8th Int. Workshop on LCPC. Springer-Verlag, August 1995.

[35] R. Hanxleden and K. Kennedy. Give-N-Take– a balanced code placement framework. In Proc. of '94 Conf. on PLDI, 1994.

[36] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. IEEE Transactions on Parallel and Distributed Systems, 2(3):350–360, July 1991.

[37] K. Hayashi, T. Doi, T. Horie, Y. Koyanagi, O. Shiraki, N. Imamura, T. Shimizu, H. Ishihata, and T. Shindo. AP1000+: Architectural support for parallelizing compilers and parallel programs. In Third Parallel Computing Workshop, pages P1–F1–P1–F9, November 1994.

[38] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, CRPC-R, January 1993.

[39] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In Proc. of the 2nd HPCA, February 1996.

[40] L. Iftode, J. P. Singh, and K. Li. Understanding Application Performance on Shared Virtual Memory Systems. In Proc. of the 23th ISCA, pages 122–133, May 1996.

[41] T. Inagaki, J. Niwa, T. Matsumoto, and K. Hiraki. Supporting Software Distributed Shared Memory with a Optimizing Compiler. In Proc. of the 1998 ICPP, pages 225–234, August 1998.

[42] J. B. Carter. Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency. PhD thesis, Department of Computer Science, Rice University, September 1993.

[43] D. Jiang, H. Shan, and J. P. Singh. Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors. In Proc. of the 6th ACM SIGPLAN Symp. on PPOPP, pages 217–229, June 1997.

[44] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. Crl: High-performance all-software distributed shared memory. In Proc. of 15th ACM Symposium on Operating System Principles, pages 213–228, December 1995.

[45] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In Proc. of the Winter 1994 USENIX Conf., pages 115–131, January 1994.

[46] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In Proc. of the 19th ISCA, pages 13–21, May 1992.

[47] P. Keleher and C. Tseng. Enhancing Software DSM for Compiler-Parallelized Applications. In Proc. of the 11th International Parallel Processing Symposium, March 1996.

[48] C. Koelbel and P. Mehrotra. Compiling Global Name-Space Parallel Loops for Distributed Execution. IEEE Transactions on Parallel and Distributed Systems, 2(4):440–451, October 1991.

[49] D. J. Kuck, D. A. Padua R. H. Kuhn, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In Proc. of 8th Annual ACM Symposium on Principles of Programming Languages, pages 207–218, January 1981.

[50] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Si Moni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. L. Hennessy. The Stanford FLASH Multiprocessor. In Proc. of the 21th ISCA, pages 302–313, April 1994.

[51] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers, C-28(9):690–691, September 1979.

[52] D. E. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In Proc. of the 17th ISCA, pages 148–159, May 1990.

[53] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In Proc. of the 1988 ICPP, pages 94–101, August 1988.

[54] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. ACM Transactions on Computer Systems, 7(4):321–359, November 1989.

[55] H. Lu, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Compiler and Software Distributed Shared Memory Support for Irregular Applications. In Proc. of 1997 Principles and Practice of Parallel Programming, pages 48–56, June 1997.

[56] T. Matsumoto. Fine Grain Support Mechnisms. In IPSJ Computer Architecture SIG Notes, volume 89-ARC-77, pages 91–98, July 1989. (in Japanese).

[57] T. Matsumoto, S. Furuso, and K. Hiraki. Resource management methods of the general-purpose massively-parallel operating system: SSS–CORE. In Proc. of 11th Conf. of JSSST, pages 13–16, October 1994. (in Japanese).

[58] T. Matsumoto and K. Hiraki. A shared-memory architecture for massively parallel computer systems. In IEICE Japan SIG Reports CPSY, pages 47–55, August 1992. (in Japanese).

[59] T. Matsumoto and K. Hiraki. MBCF: A Protected and Virtualized High-Speed User-Level Memory-Based Communication Facility. In Proc. of 1998 International Conference on Supercomputing, pages 259–266, July 1998.

[60] T. Matsumoto and K. Hiraki. Memory-Based Communication Facilities and Asymmetric Distributed Shared Memory. In Proc. of the 1997 International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems, pages 30–39, Los Alamitos, CA, 1998. IEEE Computer Society.

[61] T. Matsumoto, T. Komaarashi, S. Uzuhara, and K. Hiraki. The Asymmetric Distributed Shared Memory using Memory-Based Communication Facilities. In Proc. of the IPSJ Computer System Symposium, November 1996. (in Japanese).

[62] T. Matsumoto, T. Komaarashi, S. Uzuhara, S. Takeoka, and K. Hiraki. A General-Purpose Massively-Parallel Operating System: SSS-CORE – Implementation Methods for Network of Workstations –. In IPSJ SIG Notes, volume 96-OS-73, pages 115–120, August 1996. (in Japanese).

[63] T. Matsumoto, J. Niwa, and K. Hiraki. Compiler-Assisted Distributed Shared Memory Schemes Using Memory-Based Communication Facilities. In Proc. of the 1998 PDPTA, volume 2, pages 875–882, July 1998.

[64] E. Morel and C. Renvoise. Global Optimization by Suppression of Partial Redundancies. Communications of the ACM, 22(2):96–103, February 1979.

[65] J. Niwa, T. Inagaki, T. Matsumoto, and K. Hiraki. Efficient Implementation of Software Release Consistency on Asymmetric Distributed Shared Memory. In Proc. of the 1997 ISPAN, pages 198–201, December 1997.

[66] J. Niwa, T. Inagaki, T. Matsumoto, and K. Hiraki. Performance Evaluation of Compiling Techniques on Asymmetric Distributed Shared Memory. IPSJ Transactions on Parallel Processing, 39(6):1729–1737, June 1998. (in Japanese).

[67] J. Niwa, T. Inagaki, T. Matsumoto, and K. Hiraki. Evaluation of Compiler-Assisted Software DSM Schemes for a Workstation Cluster. In Proc. of the 1999 International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems, Los Alamitos, CA, 2000. IEEE Computer Society. Accepted for publication.

[68] J. Niwa, T. Inagaki, Takashi Matsumoto, and Kei Hiraki. Compiling Techniques on Asymmetric Distributed Shared Memory. In IPSJ High Performance Computing SIG Notes, volume 97-HPC-67, pages 121–126, August 1997. (in Japanese).

[69] J. Niwa, T. Inagaki, Takashi Matsumoto, and Kei Hiraki. Compiling Techniques for ADSM on General-Purpose Massively-Parallel Operating System: SSS-CORE. JSSST Journal of Computer Software, 15(3):54–58, May 1998.

[70] J. Niwa, T. Matsumoto, and K. Hiraki. Evaluation of Compiler-Assisted Software DSM Schemes: ADSM and UDSM. In IPSJ High Performance Computing SIG Notes, volume 99-HPC-77, pages 95–100, August 1999. (in Japanese).

[71] O. Shiraki and Y. Koyanagi and N. Imamura and K. Hayashi and T. Shimizu and T. Horie and H. Ishihata. Architecture of highly parallel computer AP1000+. In Third Parallel Computing Workshop, pages P1–G–1–P1–G–8, November 1991.

[72] M. O'Boyle and F. Bodin. Compiler Reduction of Synchronization in Shared Virtual Memory Systems. In Proc. of 1995 International Conference on Supercomputing, pages 318–327, July 1995.

[73] The OpenMP Forum. OpenMP C and C++ Application Program Interface, October 1998. http://www.openmp.org.

[74] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In Proc. of the 21st Annual International Symposium on Computer Architecture, April 1994.

[75] J. Saltz, R. Mirchandaney, and J. Crowley. Run-Time Parallelization and Scheduling of Loops. IEEE Transactions on Computers, 40(5):603–611, 1991.

[76] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh. Home-based SVM protocols for SMP clusters: Design and Performance. In Proc. of the 4th HPCA, pages 113–124, 1998.

[77] D. J. Scales and K. Gharachorloo. Design and Performance of the Shasta Distributed Shared Memory Protocol. In Proc. of 1997 International Conference on Supercomputing, pages 245–252, July 1997.

[78] D. J. Scales and K. Gharachorloo. Performance of the Shasta Distributed Shared Memory Protocol. Research Report 97/2, DEC Western Research Laboratory, February 1997.

[79] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In Proc. of ASPLOS-VII, pages 174–185, October 1996.

[80] D. J. Scales and M. S. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In Proc. of the 1st OSDI, November 1994.

[81] Sun Microsystems Inc. The Java Language Specification, 1996. Also available at http://java.sun.com/docs/books/jls/.

[82] R. P. Wilson. Efficient Context-Sensitive Pointer Analysis for C Programs. PhD thesis, Stanford University, December 1997.

[83] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S. Liao, M. W. Hall C. Tseng, M. S. Lam, and J. L. Hennessy. An overview of the SUIF compiler system. http://suif.stanford.edu/suif/suif1/suif-overview/suif.html.

[84] R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In Proc. of '95 Conf. on PLDI, pages 1–12, June 1995.

[85] M. E. Wolf and M. S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. IEEE Transactions on Parallel and Distributed Systems, pages 452–471, October 1991.

[86] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In Proc. of the 22nd ISCA, pages 24–36, June 1995.

[87] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad. Software Write Detection for a Distributed Shared Memory. In Proc. of the 1st Symp. on OSDI, November 1994.

[88] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In Proc. of the 2nd Symp. on OSDI, 1996.

[89] Y. Zhou, L. Iftode, K. Li, J. P. Singh, B. R. Toonen, I. Schoinas, M. D. Hill, and D. A. Wood. Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation. In Proc. of the 6th ACM SIGPLAN Symp. on PPOPP, pages 193–205, June 1997.