OPTIMIZING PARALLEL PROGRAMS WITH DYNAMIC DATA STRUCTURES

by
Yingchun Zhu

School of Computer Science
McGill University, Montreal

May 2000

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Abstract

Distributed memory parallel architectures support a memory model where some memory accesses are local, and thus inexpensive, while other memory accesses are remote, and potentially quite expensive. In order to achieve efficiency on such architectures, we need to reduce remote accesses. This is particularly challenging for applications that use dynamic data structures.

In this thesis, I present two compiler techniques to reduce the overhead of remote memory accesses for dynamic data structure based applications: locality techniques and communication optimizations. Locality techniques include a static locality analysis, which statically estimates when an indirect reference via a pointer can be safely assumed to be a local access, and dynamic locality checks, which consists of runtime tests to identify local accesses. Communication techniques include: 1) code movement to issue remote reads earlier and writes later; 2) code transformations to replace repeated/redundant remote accesses with one access; and 3) transformations to block or pipeline a group of remote requests together. Both locality and communication techniques have been implemented and incorporated into our EARTH-McCAT compiler framework, and a series of experiments have been conducted to evaluate these techniques. The experimental results show that we are able to achieve up to 26% performance improvement with each technique alone, and up to 29% performance improvement when both techniques are applied together.

Résumé

Les architectures parallèles à mémoire distribuée sont dotées d'un modèle de mémoire où certains accès à la mémoire sont locaux, et donc peu coûteux, et d'autres sont éloignés et potentiellement très coûteux. Pour donner un haut rendement sur de telles architectures, il faut réduire le nombre d'accès éloignés. Ceci est un défi de taille, particulièrement pour les applications qui utilisent des structures de données dynamiques. Dans cette thèse, nous présentons deux techniques de compilateur pour réduire les coûts supplémentaires des accès éloignés à la mémoire dans les applications basées sur des structures de données dynamiques: des techniques de localité et des optimisations de communication. Les techniques de localité incluent une analyse statique de localité qui estime quand une référence indirecte par pointeur peut être considérée comme un accès local en toute sécurité, et des vérifications dynamiques de localité qui consistent en tests identifiant les accès locaux pendant l'exécution. Les techniques de communication incluent: 1) Des mouvements de code pour émettre les lectures éloignées plus tôt et les écritures plus tard. 2) Des transformations pour remplacer les accès répétés/superflus par un accès unique. 3) Des transformations pour émettre en bloc ou en pipeline un groupe de requêtes éloignées. Ces deux techniques ont été implantées et incorporées dans note système de compilation EARTH-McCAT, et une série d'expériences a été effectuée pour évaluer ces techniques. Les résultats démontrent une amélioration de rendement atteignant 26% pour chacune des techniques utilisée séparément, et atteignant 29% quand les deux techniques sont utilisées conjointement.

Acknowledgments

First and foremost, I would like to thank my supervisor, Professor Laurie Hendren, whose constant support, guidance, and encouragement have been the mainstay of my PhD years. Her careful training, constructive criticism, and confidence in my abilities greatly helped me in honing the skills required for research. The stimulating discussions during our weekly meetings, the practice rounds for conference talks, the emphasis on experimenting with new ideas have all contributed much to my reasearch training, and to this thesis. She also supported me all through the PhD program, which kept me from many a worry. She has been the best supervisor one can hope for: quite caring and understanding, very insightful, and lots of fun to work with. Words are not enough to express my gratitude for all she has done for me!

During my stay at the school of computer science, I have met many wonderful people. I want to thank Professor Gao, for starting an excellent ACAPS group. I admire his hard-working approach, and enjoyed all the holiday parties at his house. I would also like to thank Professor Newborn, for his insightful seminar on finance, and for starting the option investment group. Though the group lasted only one year, we had much fun in the activities. Besides, we set a good portfolio as well! I owe many thanks to my fellow students from ACAPS: Rakesh Ghiya, for his friendship, cool-headed advice, and good-humored company during the many coffee-breaks to escape work; Christopher Lapkowski, a guy proud of his heritage, Canadian or Polish, for his kindness and prompt help; Yingwei Zhang, for his invaluable help in initially settling down in Montreal; and Andres Marquez, for surviving the comprehensive examination together, and for the many interesting discussions during our PhD studies.

The analyses presented in the thesis are built upon the EARTH-McCAT compiler framework, which is a result of team work. Thanks to all the people who have contributed to the framework, especially to Xun Xue, Xinan Tang, Shereen Ghobrial, Haiying Cai and Pierre Ouellet.

The administrative staff in the school has been very understanding. Many thanks to Vicki Keirl, Franca Cianci, and Lise Minogue for all their help.

Finally, I want to thank my parents and sisters, for their love, patience, and unshakable support. In particular, to my father, whom I most love and respect, who inspired my early interest in computer science, and whose wisdom and advice I will cherish all my life.

This work was partly supported by NSERC and FCAR.

Dedicated to the memory of my father Kun Zhu

Contents

Abstract ii

Résumé iii

Acknowledgments iv

1 Introduction and Motivation 1

1.1 Related Work 3

1.2 Our approach 5

1.3 Thesis Goal and Organizations 7

2 Setting 9

2.1 The EARTH-C Language 9

2.1.1 Local and Remote Memory Accesses 12

2.1.2 Remote and Basic Functions 16

2.1.3 Library Functions and Primitives 18

2.2 The EARTH-McCAT C Compiler 20

2.2.1 Phase I: standard transformations and analyses 20

2.2.2 Phase II: locality and parallelism enhancement 22
2.2.3 Phase III: code generation 23
2.3 The EARTH Execution Model 28
2.4 Benchmark Suite 30

3 Locality Analyses 31
3.1 Motivation 32
3.1.1 Pointers to Local Variables and Parameters 32
3.1.2 Dynamic Memory Allocation 33
3.1.3 Mapping Computation to the `OWNER_OF` Data 34
3.1.4 Mapping Computation to `HOME` 34
3.2 Static Locality Analysis and Specialization 36
3.2.1 Intraprocedural Locality Propagation 39
3.2.2 Interprocedural Locality Propagation 43
3.2.3 Specializing Functions 44
3.3 Dynamic Locality Checks and Specialization 46
3.3.1 Fine-grain dynamic tests 46
3.3.2 Coarse-grain dynamic checks 46
3.4 Experimental Results 49
3.4.1 Effect of Static Locality Analysis and Specialization 49
3.4.2 Effect of Dynamic Locality Tests 58
3.4.3 Summary of Experimental Results 61
3.5 Related Work 62
3.6 Summary 63

4 Communication Optimizations **65**

4.1 Motivating Examples 66

4.2 The Algorithm 68

4.2.1 Possible-Placement Analysis 68

4.2.2 Communication Selection 80

4.3 Experimental Results 85

4.4 Related Work 97

4.5 Summary 100

5 Combining Locality and Communication Optimizations **102**

5.1 Experimental Results 102

5.1.1 Overall Performance Improvement 103

5.1.2 Improvement on Individual Benchmarks 105

5.2 Summmary 113

6 Conclusions and Future Work **114**

6.1 Future Work 116

Bibliography **118**

List of Figures

1.1 Programmer Effort vs. Application Performance 6

2.1 Examples of `forall` Loops 10

2.2 Examples of Statement Sequences 12

2.3 The EARTH-C Memory Model 13

2.4 Examples of Local-pointer Declarations 15

2.5 Example of Local Pointers and Basic Functions 18

2.6 Example Functions Written in EARTH-C 19

2.7 Overall Structure of the Compiler 21

2.8 DDG and List Scheduling 24

2.9 Improved DDG 26

2.10 Threaded-C Code 27

2.11 The EARTH Architecture 28

3.1 Locality for Pointers 33

3.2 Locality for Dynamic Memory Allocation 34

3.3 Locality Generated Using `OWNER_OF` 35

3.4 Locality Generated Using `HOME` 36

3.5 Overall Algorithm for Locality Analysis 37
3.6 Example of Interprocedural Locality Propagation 45
3.7 Example of Fine-grain Locality Tests 47
3.8 Example of Coarse-grain Locality Checks 48
3.9 Sample Specialized Codes from Benchmarks 54
3.10 Sample Specialized Codes from Benchmarks 56
3.11 Sample Code from Health Benchmark 59
4.1 Optimizing Remote Reads 67
4.2 Optimizing Remote Reads/Writes 69
4.3 Example of a Safely Placed Read Tuple 71
4.4 Example of a Safely Placed Write Tuple 72
4.5 Possible Placement Analysis - Basic Rules 74
4.6 Possible Placement Analysis - Compound Rules 75
4.7 Assuming p and t Are Aliased 76
4.8 Assuming p and t Are Aliased 77
4.9 Example Propagation of RemoteRead Sets 79
4.10 Example Propagation of RemoteWrite Sets 80
4.11 Applying Communication Selection 82
4.12 Selection Analysis - Compound Rules 84
4.13 Applying Write Communication Selection 85
4.14 Applying Write Communication Selection 86
4.15 Applying Communication Selection 86

4.16 Applying Communication Selection 87

4.17 Improvement on Dynamic Communication Counts 88

4.18 Improvement on Dynamic Communication Size 89

4.19 Benchmark Codes Illustrating Communication Optimization 95

4.20 Benchmark Codes Illustrating Communication Optimization 96

5.1 Performance Improvement for PowerII 106

5.2 Performance Improvement for Tsp 107

5.3 Performance Improvement for Em3d 108

5.4 Performance Improvement for PowerI 109

5.5 Performance Improvement for Perimeter 110

5.6 Performance Improvement for Health 111

5.7 Performance Improvement for Voronoi 112

Chapter 1

Introduction and Motivation

Parallel computing has been an active topic of research for the past twenty-five years. The goal of parallel computing is accelerating the execution of programs by exploiting the parallelism available at the fine-grain or coarse-grain level. Modern microprocessors, with superscalar or VLIW (very long instruction word) architectures, exploit the fine-grain instruction-level parallelism in programs. Parallel machines targeted towards coarse-grain parallelism use multiple processors, and achieve faster execution by assigning independent tasks within an application to different processors within the machine. As the upper-limit of computation power for microprocessors is approaching, parallel machines begin to play an important role. Furthermore, the progress on hardware technology has made parallel machines cost-effective for many new application areas.

Parallel machines can either have a shared-memory model, where all processors share a single centralized memory, or a distributed memory model, where each processor has been assigned its own local memory. Memory accesses in multiprocessor distributed machines are either local or remote. Local accesses belong to the processor issuing the access. Remote accesses occur when the accessed data resides on another processor's memory. Naturally, remote memory accesses tend to be more expensive. Depending on the communication mechanism, type of interconnection network, and scale of the machine, existing parallel machines may take from 1 μs to 100 μs to communicate data between processors. Table 1.1 shows the typical round-trip delays to retrieve a word from a remote memory for several different parallel machines.

To obtain maximum performance from parallel machines, especially distributed

Machine	*Communication mechanism*	*Network interconnection*	*Remote access*
SGI Challenge	Shared memory	Bus	1 μs
Cray T3D	Shared memory	3D torus	1 μs
KSR-1	Shared memory	Hierarchical ring	2-6 μs
CM-5	Distributed Memory	Fat Tree	10 μs
Intel Paragon	Distributed memory	2D mesh	10-30 μs
IBM SP-2	Distributed memory	Multistage switch	30-100 μs
EARTH MANNA	Distributed memory	Multistage switch	20-90 μs

Table 1.1: Typical remote access time to retrieve a word from a remote memory

memory machines, the programmer needs to manage both parallelism in the application, and the way data is distributed across the memory modules. This is typically a cumbersome and error-prone task for the programmer. Therefore, the programmer needs support from the architecture, the compiler and the programming language to achieve this task efficiently and effectively. For array-based regular scientific applications, compilers are able to provide a significant help through automatic detection of parallelism and its efficient exploitation via suitable distribution of work among the processors available. However, for irregular applications that typically use dynamic data structures, such as lists, trees and DAGs, the current compiler technology is not equally mature. Nevertheless, irregular applications form a significant portion of the problems that need the power of parallel computing. Typical areas that use irregular applications include fluid dynamics, simulation, graphics and optimization packages.

The goal of the EARTH-McCAT project at McGill University is to build a system to efficiently parallelize such dynamic applications with minimum effort from the programmer. To achieve this, we have designed a system, consisting of the EARTH-C language (a parallel dialect of C), a memory model and an advanced EARTH-McCAT compiler. Using EARTH-C, programmers are able to express locality and parallelism of their applications at a high level. The EARTH-McCAT compiler automatically generates efficient low-level threaded code for EARTH-C programs, which executes on the EARTH (Efficient Architecture for Running THreads) multithreaded architecture. Thus, programmers are completely insulated from low-level details. The efficiency of the compiler-generated code comes from a combined effort of advanced compilation techniques and an efficient run-time system. As an integral part of the project, this thesis focuses on compiler techniques to reduce the overhead of communication that arises from remote memory accesses. It presents two types of optimizing techniques: locality techniques and communication optimizations to infer further locality and minimize communication overhead.

The remainder of this chapter is organized as follows. Section 1.1 provides a review for the research work that has been done in the area. Section 1.2 presents our approach. Finally, Section 1.3 summarizes the thesis contributions, and provides an outline for the rest of the thesis.

1.1 Related Work

In this section, we discuss the approaches for applications using dynamic data structures.

Although applications with irregular data structures exhibit parallelism similar to array-based applications, where parallelism exists among the computations on relatively independent subcomponents of a data structure, they tend to have their data structures built dynamically, and therefore need to be distributed dynamically. The dynamic nature of the data structure further complicates the parallelization of applications because the program can affect the data distribution at runtime. Consequently, a combined effort from the programmer, the compiler and the runtime system is necessary in order to achieve optimal results. To address this issue, the following systems have been developed for high performance programming.

Cilk [BJK+95, MF98], [Ran98] is a C-based multithreaded computing system developed at MIT's Laboratory for Computer Science. The philosophy behind Cilk is that a programmer should concentrate on exposing parallelism and exploiting locality, and let the runtime system take care of details like scheduling, load balancing and communication protocols. Using Cilk, programmers are able to express their applications either by writing multithreaded code in a continuation-passing style, or by using normal call and return semantics with specified parallel calls. The Cilk system provides efficient and guaranteed performance through a "two-clone" compilation strategy and work-stealing scheduler in the runtime system. The Cilk system is mainly designed for shared memory multiprocessors. With the support of a software shared-memory system, it is able to run on distributed memory environments as well.

The EARTH system [HMT+96, MGH+96], originally developed at McGill University, is another C-based multithreaded system for programming on distributed memory architectures. A programmer may use a low-level multithreaded language

THREADED-C to explicitly express threads, remote communications and synchronization within a program, and the runtime system implements a set of dynamic load balancing strategies to optimize the performance. In this thesis, we concentrate on a higher-level language, EARTH-C, which is compiled into THREADED-C.

EM-C [SKSY94] is a similar multithreaded system developed at RWC Tsukuba Research Center, Japan. The language EM-C provides the notion of a global address space, and a simple data distibution in that space for a programmer to exploit data locality. In addition, a programmer may use such language constructs as task blocks and parallel sections to express parallelism. An EM-C program is compiled into threaded code, running on EM-4, a distributed memory architecture with a dataflow mechanism.

Split-C [CDG+93] is a system developed at University of California, Berkeley. The language Split-C provides a global address space, with a clear notion of local and global access and a simple data layout strategy, to allow programmers to minimize the frequency of remote access. It also supports a rich set of assignment operators, to optimize the remote accesses and to integrate co-ordination among processors with the flow of data. With these simple primitives, it captures many of the best aspects of shared memory, message passing and data parallel programming styles, while retaining a "small language" philosophy as well.

Olden [RCRH95, CR95] developed at Princeton University, is a system specially designed for parallelizing dynamic applications on distributed memory architectures. Like Split-C, Olden provides a global memory model, where a programmer is able to explicitly map dynamic data structures over a distributed heap, and annnotes parallel function calls using futures. The system employs the techniques of software caching and computation migration for efficiency. Based on programmer hints regarding the data layout, a compile-time heuristic is developed to select between these techniques. An Olden profiler is designed to help a programmer verify the data layout hints. In addition, Olden also implements a local-knowledge cache coherence mechanism for the cache for efficiency.

In summary, all these systems provide language features for a programmer to express parallelism and locality in a program. Furthermore, some systems have developed a set of techniques, including various load balancing strategies and cache coherence mechanisms to improve the performance. However, few of these systems provide the techniques to address the issue of remote communication overhead, which is one of the main performance challenges for distributed memory architectures. In order to obtain good performance on such architectures, a significant amount of effort is required from a programmer to exploit data locality. This is true for the EARTH THREADED-C system, where a programmer has to take care of threads, synchronizations, remote communications and locality exploition as well. This can be overwhelming for most nontrivial applications. Similarly, little emphasis has been placed on reducing remote communication overhead in Cilk, a system mainly designed for shared memory architectures, as its optimization techniques are centered around the load balancing issue. Although Split-C and EM-C both provide a high level language for programming on distributed architectures, when it comes to write dynamic applications, they all require a programmer to write at a lower level, in order to better exploit data locality. Nevertheless, the Olden system, specially designed for the performance of dynamic applications on distributed memory architectures, implements software caching and computation migration techniques to optimize remote memory accesses.

Work directly related to our new techniques is presented in more detail in Section 3.5 and 4.4.

1.2 Our approach

It is our belief that one main challenge in parallel processing is to provide a programming model that is both simple for the programmer, and efficient for the system. One would like to give the programmer a familiar programming language, and have the programmer focus on high-level aspects such as coarse-grain parallelism and perhaps some sort of static or dynamic data distribution. Compiler techniques are then required to effectively map the high-level programs to actual parallel architectures.

This idea is better illustrated using the graph in Figure 1.1, in which the X-axis represents the programmer effort involved in a program, and the Y-axis represents the speedup on N nodes, where N is fixed. This picture is drawn from our parallel programming experience on distributed memory architectures, and the curves are intended to illustrate the general trend rather than an exact quatitative measurement. In general, a parallel program can be implemented with four different versions, each representing a different degree of programmer effort involved in the specification of parallelism and data distribution. A *parallelized sequential* version is completely sequential, and relies solely on automatic parallelization to achieve efficiency, while a *naive parallel* version is one in which a programmer exploits parallelism using parallel annotations. Based on the *naive parallel* version, there is a *parallel + data distribution* version in which a programmer specifies a coarse-grain data distribution, and a *hand-tuned parallel* version where a programmer hand-tuned the locality and communication at the finest degree in order to maximize the efficiency.

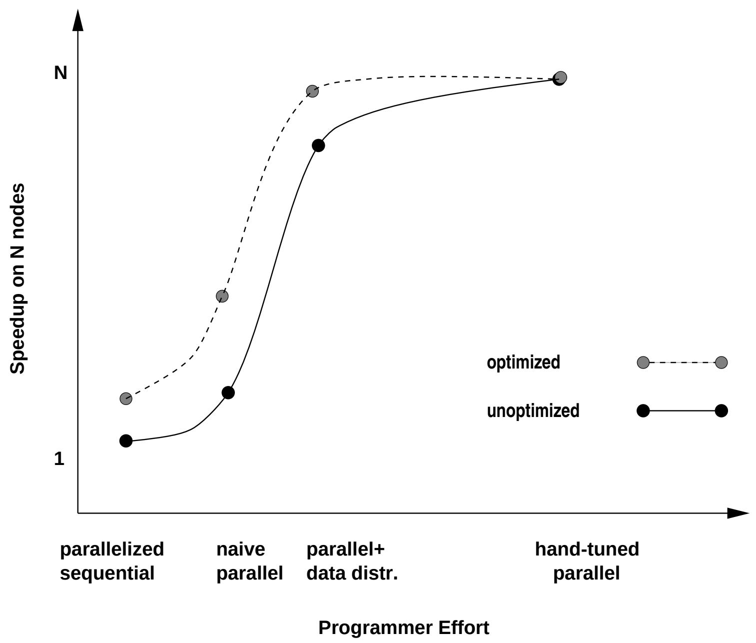


Figure 1.1: Programmer Effort vs. Application Performance

The solid curve line represents the speedup of these versions without compiler optimizations. It is obvious that we achieve little speedup with a minimum program effort in a *parallelized sequential* version. The more programmer effort is involved, the bigger speedup we are able to achieve. With a good data distribution, better data locality can be exploited, which leads to a big performance jump from the *naive parallel* version to the *parallel + data distribution*. Ideally, a hand-tuned program would obtain a linear speedup(N) when running on N nodes. Compiler optimizations generally improve the speedup for these versions, as indicated by the dotted line. Although compiler optimizations improve both the *naive parallel* version and the *parallel + data distribution* version, we are particularly interested in *parallel + data distribution* version. With a little more programmer effort than that of *naive parallel* version, but much less than that of the *hand-tuned parallel* version, we are able to achieve a speedup almost as good as that of the *hand-tuned parallel* version. And this is exactly the goal of the research presented in this thesis.

Guided by this philosophy, we designed a high-level parallel language called EARTH-C [HTZ+97], and developed an associated compiler that translates EARTH-C programs to low-level THREADED-C programs that execute on the EARTH multithreaded architecture [HMT+96, MGH+96]. Our main emphasis is on the effective compilation of programs that use irregular, dynamically-allocated data structures. To achieve the goal, we implemented a set of advanced compilation techniques in the EARTH-McCAT. This thesis focuses on the compiler techniques implemented to reduce the overhead of communication, thus to improve the performance for applications that use dynamic data structures. Given an EARTH-C program, with user-specified parallelism, and a coarse-grain data distribution, we aim to optimize its performance to the level which a hand-tuned version would achieve, but with much less programmer effort. Two kinds of optimization techniques are presented in the thesis: locality techniques and communication optimizations. Locality techniques include a static locality analysis, which statically estimates when an indirect reference via a pointer can be safely assumed to be a local access, and dynamic locality checks, which consist of runtime tests to identify local accesses. Communication techniques include: 1) rearrangement of code to issue remote reads earlier and writes later; 2) code transformations to replace repetitious or redundant remote accesses with a single access; and 3) transformations to block or pipeline a group of remote requests together. The experimental results we performed on the benchmark suite demonstrate that we are able to achieve moderate to significant speedups with these techniques.

1.3 Thesis Goal and Organizations

The goal of the thesis is to design a set of advanced compilation techniques, being incorporated into the EARTH-McCAT framework, in order to generate efficient code running on the EARTH-MANNA system. In the context of this goal, the thesis makes the following the main contributions:

- A set of locality techniques. A static locality analysis based on type-inference techniques is designed. This analysis identifies the indirect references through pointers that can be safely assumed to be local accesses. In addtion, two dynamic locality techniques, in which runtime tests are inserted to determine if a memory reference is local, are also studied.

- A set of communication optimizations. A framework for communication optimizations is proposed in an effort to reduce the overhead of communication. The framework consists of an analysis phase and a transformation phase. The analysis phase finds all possible points for insertion of remote memory operations, while the transformation phase both selects the "best" place for inserting the communication, and determines if pipelining or blocking of communication should be performed.

- Implementations and empirical studies of each analysis on EARTH-MANNA. All these analyses have been implemented and incorporated in the EARTH-McCAT compiler and a set of experiments have been performed on EARTH-MANNA. The experimental results show that up to 26% performance improvement over the unoptimized benchmarks is achieved by applying analyses individually.

- Empirical study of the interaction between locality analysis and communication optimizations. Further experiments for evaluating the combined effect of these analyses are conducted, and the results are analyzed and compared with ones obtained by applying each analysis individually. The results demonstrate that these two techniques are orthogonal, and the performance improvement reaches 29% when both techniques are applied together.

The rest of the thesis is organized as follows. Chapter 2 gives an overview of the EARTH-McCAT project, including the EARTH-C language, its memory model, the EARTH-McCAT compiler and the EARTH execution model. The benchmark suite is also described in this chapter. Chapter 3 presents locality-related optimizations, including a static locality analysis as well as dynamic locality tests. Chapter 4 describes a uniform framework for communication optimizations which reduce remote accesses and overlap remote latency with computation. Chapter 5 provides an empirical study of the interaction between locality analysis and communication analysis. The final chapter summarizes the contributions and the current status of the research.

Chapter 2

Setting

This chapter outlines the setting in which our analyses and transformations have been implemented. We choose EARTH-C as our source language, the EARTH-McCAT compiler as the platform for our implementations, and EARTH-MANNA as our testbed. EARTH-C is a high-level parallel C language. Once a programmer writes a parallel program in EARTH-C, the EARTH-McCAT C compiler transforms it into a low-level threaded-C program, which is executed on EARTH-MANNA, a distributed memory architecture.

The rest of the chapter is organized as follows. Section 2.1 describes the major EARTH-C extensions to C. The extensions include the specification of parallelism and locality, the specification of function distribution, and the libraries. Section 2.2 provides an overview of the EARTH-McCAT C compiler. Section 2.3 presents the EARTH-MANNA execution model. Finally, the benchmark suite is discussed in section 2.4. Material for this chapter has been extracted from several sources [HTZ+97, HMT+96].

2.1 The EARTH-C Language

The design of EARTH-C has been driven by three factors: (1) the goal to create a high-level parallel language hiding all low-level details; (2) the desire to be compatible with C, with extensions simple, yet expressive; and (3) the need to convey some information about control parallelism and data locality to the compiler. Any C program is a valid EARTH-C program, and the compiler will automatically generate a correct low-level threaded program. However, a programmer will usually use EARTH-C constructs to expose parallelism and improve data locality. The following sections describe these EARTH-C features and their design purposes.

- Parallel Loops

  The EARTH-C language supports parallel loops in which the loop header may have inter-iteration dependencies, while the loop body must have no inter-iteration dependencies.[1] Dependencies among loop headers are allowed as the compiler will handle them with the appropriate transformations. Parallel loops are declared using keyword forall. A forall loop construct is expressed as follows:

  $$\textsf{forall}\ (\ v = \textit{init\_expr}\ ;\ \textit{test\_expr}\ (v);\ v = \textit{update\_expr}\ (v)\ )$$
  $$\textit{loop\_body}\ (\ v\ )$$

  The operational semantics of a parallel loop can be stated as follows. Let $v_1 \ldots v_n$ be the values generated by executing the loop header. Then all loop bodies *loop_body*(v_1), *loop_body*(v_2), ..., *loop_body*(v_n) **may** be executed in parallel. Figure 2.1 gives two examples of forall loops. The first, in Figure 2.1(a), is a typical parallel loop in scientific computing, where the different iteration values for `i` can be computed with a closed form. The second example, in Figure 2.1(b), gives an example of traversing a linked list, where the different values of `p` must be calculated sequentially, but many iterations of the body may proceed in parallel.

`forall (i=0; i<N; i++)` `{ a[i] = b[i] * s; }`	`forall (p=hd; p!=NULL; p=p->next)` `{ f(p); }`
(a) completely parallel	(b) with sequential header

Figure 2.1: Examples of `forall` Loops

[1]dependencies within an iteration will be allowed as the compiler will perform the appropriate renaming in the parallelization phases.

- Statement Sequences

  EARTH-C supports three kinds of statement sequence: *ordinary*, *parallel*, and *strictly sequential*. Ordinary sequences are the same as those used in C. Occasionally, a programmer would like to explicitly specify parallel or strictly sequential sequences. EARTH-C introduces `{^ ... ^}` and `{! ... !}` for this purpose. The following table describes each type of statement sequence and provides an example of each:

Statement Type	Description

Ordinary
`{ a = f(b);`
`  foo(a);`
`  goo(b);`
` }`
	As in C, an ordinary sequence is denoted using the block syntax `{ ... }`, and its statements may be executed in any order, as long as all data dependencies are obeyed. In the left example, `a = f(b);` must precede `foo(a)` because of the flow-dependence on `a`.
	An EARTH-C programmer is expected to usually use ordinary statement sequences, and leave it to the compiler to exploit most of the statement-level parallelism.

Parallel
`{^`
` f(t->left);`
` f(t->right);`
` ^}`
	A parallel statement sequence is a sequence where there are no dependences among the statements, and all statements may be safely executed in parallel. This kind of sequence is particularly useful in the case when the compiler's dependence analysis is unable to detect the parallelism automatically.
	In the left example, the function calls to `f` can be done in parallel (i.e. the programmer guarantees that there is no interference between the calls).

Strictly Sequential
`{!`
` a = f(x);`
` b = g(y);`
` foo(a,b);`
` !}`
	`{!...!}` defines a strictly sequential sequence, where statements must be executed in their lexical order. Strictly sequential sequences are used when the data dependencies would allow the compiler to rearrange the statements, but the programmer wants to ensure a specific order (for example, they may want to limit the amount of parallelism exposed).
	In the left example , the programmer is specifying that the call to `f` must be executed before the call to `g`, even though there is no dependence between the two calls.

  These different sorts of statement sequences are allowed to appear in any place where an ordinary sequence will be placed. For example, it may be used with compositional statements such as **if**, **switch**, **while**, **for** and **do** statements. Figure 2.2(a) gives an example of two **for** loops that can execute in parallel. Moreover, these different statement sequences can be arbitrarily nested. Figure 2.2(b) gives an example with a parallel sequence nested inside an ordinary sequence.

<pre>{^ for (i1=0; i1 < n/2; i1++)
      f(i1);
    for (i2=n/2+1; i2<n; i2++)
      f(i2);
 ^}</pre>
	<pre>{ t_l = t->left;
	  t_r = t->right;
	  {^ f(t_l);
	     f(t_r);
	  ^}
	}</pre>

(a) sequence of `for` statements	(b) nested sequences

Figure 2.2: Examples of Statement Sequences

2.1.1 Local and Remote Memory Accesses

As mentioned previously, in a distributed memory architecture, where memory is physically distributed among the processors, there exists a crucial difference between the cost of a *remote* memory access (on a different processor) and that of a *local* memory access (memory is on the local processor). Thus, when designing a parallel language targeting such an architecture, it is important to introduce local and remote concepts. In EARTH-C, we introduce a few locality related features. Before we discuss these features, let us first look at the memory model of EARTH-C.

The memory model

Figure 2.3 presents the memory model supported by EARTH-C. In this memory model, we may have global variables, formal parameters, local variables and indirect references via pointers. The following sections presents the syntax, the semantics and the locality attribute for each kind of these variables.

Global variables: EARTH-C supports three kinds of global variables: **ordinary** globals, **replicated** globals and **shared** globals.

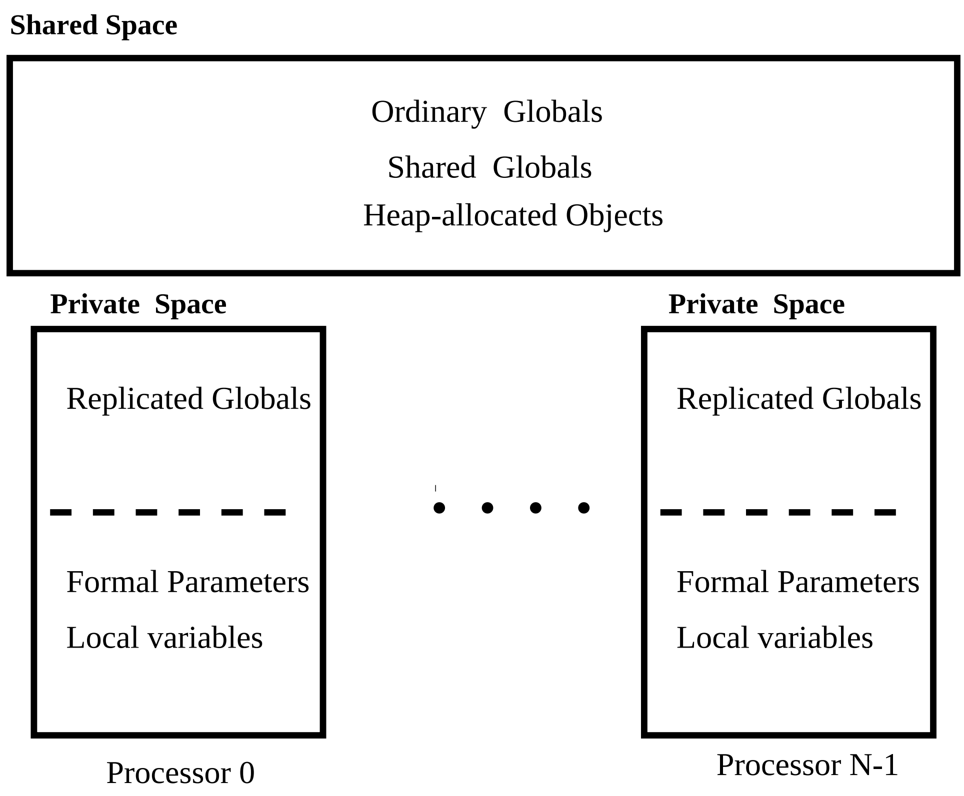


Figure 2.3: The EARTH-C Memory Model

Ordinary globals *int a[4];* *int b[4]@MAXP;*	By default, all global variables are allocated in the memory space of a fixed processor. For example, in the EARTH model, global variables are always allocated on processor 0. Thus, all direct references (i.e. references via the **name** of the variable, and not indirect references through pointers) to global variables are considered to be remote. An EARTH-C programmer may also specify a global variable allocated on a particular processor. For instance, the second example on the left declares a global array located on processor MAXP. Accesses to this kind of global will be local to the processor on which they reside, but remote from all other processors.
Replicated globals *replicated int c[4];*	Another kind of globals are replicated globals, where there is a copy of the variable on all processors. A common use of replicated variables occurs during the initialization stage, where an identical copy of data is distributed across all processors. Once declared as a replicated global, the compiler treats each copy of the global as a distinct variable, so there is no consistency between the copies. Accesses to a replicated global are always made to the local copy of the global, and will be considered local accesses by a compiler.
Shared globals *shared int count;*	The programmer may also concurrently access the memory. This is done only through shared variables in EARTH-C. Shared variables are explicitly declared using the keyword shared and may be any type (scalars, pointers, arrays or structures). Shared variables must be accessed via atomic operations. They are handled with a weak consistency model where each thread sees its own writes in an order that obeys data dependencies.

Formal Parameters and Local Variables: All formal parameters and local variables for a function invocation are allocated in an activation frame that resides in the memory of the processor executing the invocation. Thus, all direct references to parameters and local variables are to local memory.

Indirect References via Pointers: The rules for locally-scoped variables and globals allow the compiler to determine if a direct memory reference is local or remote. In summary, references to locally-scoped variables (local declaration or formal parameter), and replicated globals are local references, while references to ordinary globals are remote references. However, the presence of aliases complicates the situation for indirect references, as pointer dereferences may refer to globals, to local variables outside of the current function activation, or to heap-allocated objects. A conservative assumption will be that all pointer dereferences refer to remote memory, which unavoidably slows down a program.

In order to improve the situation, EARTH-C defines a local pointer type to express the case when a pointer is guaranteed to point to local memory. Local pointers point to local memory locations, and are created by using local * in type declarations. In Figure 2.4 we are declaring `p` to be a remote pointer, `q` to be a local pointer, `r` to be an remote pointer to a remote pointer, `s` to be a local pointer to a remote pointer, and `t` to be a local pointer to a local pointer. Thus, assuming that `p`, `q`, `r`, `s` and `t` are locally-scoped variables, the memory accesses to `p`, `q`, `*q`, `r`, `s`, `*s`, `t`, `*t` and `**t` would be local, whereas the memory accesses to `*p`, `*r`, `**r` and `**s` would be remote.

```
int * p;
int  local  * q;
int * * r;
int *  local * s;
int local  * local * t;
```

Figure 2.4: Examples of Local-pointer Declarations

Interference Rules

An important point about the EARTH-C language is that parallel computations expressed via parallel statement sequences or `forall` loops may **not** interfere except on explicit shared variables. It is the programmer's responsibility to ensure this non-interference for explicit parallel constructs. The compiler ensures this non-interference for all parallelizing transformations.

Shared variables are accessed through atomic operations. They are handled with a weak consistency model where each thread sees its own writes in an order that obeys data dependencies. Any variable that is not shared is called an *ordinary* variable; most memory accesses in EARTH-C programs are made via such variables. The program given in Figure 2.6(b) shows one case of parallel computations on large dynamic data structures. Since the sub-computations are only reading the data structure that do not interfere, ordinary variables are used instead of shared variables.

Execution of an EARTH-C program can lead to many parallel threads. The memory model interacts with concurrency as follows:

- A *local memory reference* in some thread T **must** access memory that is local to the processor executing T, and no other concurrent thread may interfere with this memory location (i.e. if one thread writes to the location, no other concurrent thread can read or write the location). Local memory references usually are inexpensive, and they arise from references to locally-scoped variables, parameters, and accesses via local pointers.

- A *shared remote memory reference* in some thread T accesses memory that **may** be on another processor. Other threads may concurrently access the same memory location, but all accesses to this location must be done via explicit shared memory operations, which guarantee atomicity. Shared memory references are the most expensive sort of reference, and they are typically used sparingly.

- An *ordinary remote memory reference* in some thread T accesses memory that **may** be on another processor. No other concurrent thread T' may interfere with this memory location (i.e. if one thread writes to the location, no other concurrent thread can read or write the location). Ordinary remote memory references are compiled to low-level EARTH primitives, and are considerably more expensive than local memory references. They arise from accesses via global variables and non-local pointers.

2.1.2 Remote and Basic Functions

In order to support the parallel execution of a program, it is necessary to be able to invoke functions on different processors. In fact, function invocation is the only way for an EARTH-C programmer to explicitly send computation to another processor.

Functions may be invoked on different processors using one of the following ways:

- foo(...);
- foo(...)@i;
- foo(...)@HOME;
- foo(..p..)@OWNER_OF(p);

If a function is called without specifying the processor on which it is invoked, the runtime load balancer assigns it to an appropriate processor. If the programmer wants to invoke a function on a specific processor, the call may include an expression that denotes the processor number, for example, `foo(...)@i`. The expression after the `@` symbol must denote an integer value where a positive value, ($p \geq 0$), will cause the function to be invoked on processor p, whereas a negative value, ($p < 0$), indicates that the run-time load balancer should be used to select the appropriate processor. Often an invocation occurs on the same processor as the calling function, which is denoted by `foo(...)@HOME`. In some cases, we may take advantage of data locality, and distribute a function to a processor which owns the data, using the expression @OWNER_OF.

A substantial overhead is incurred whenever functions are invoked on other processors. In order to reduce the invocation overhead, the programmer may use *basic* functions when there is enough parallelism. Basic functions are indicated by including the keyword basic at the beginning of the function declaration. A basic function must not contain any remote operations. In other words, it only accesses to the memory local to the current processor, and it may only call other basic functions, In addition, it is always invoked on the processor from which it is called. Thus a compiler may assume that all direct and indirect memory references within a basic function are local.

Consider a typical example in Figure 2.5. In this example, the function `incr_tree` is a remote function that traverses a binary tree, incrementing the key field of each node by k. The two recursive calls to `incr_tree` are invoked on the processors that own the roots of the left and right sub-trees using the built-in primitive OWNER_OF which returns the processor that owns the address. The function `incr_node` is declared as a basic function, so it will be invoked on the current processor, and all memory accesses to `root` will be assumed to be local. Note that inside basic functions all memory references are implicitly local, so we do not need to explicitly give a local pointer type to the parameter `node` in the function `incr_node`, although it would also be correct to give the explicit local declaration.

The main idea of the program in Figure 2.5 is that the remote invocations for the recursive traversal via incr_tree expose parallelism and send computation to the processor owning the data, thus it is worth the extra overhead. Whereas, it is preferable to make the computation for incr_node less expensive by indicating that it is a basic function. If we had omitted the keyword basic in the declaration of incr_node the program is still correct, but less efficient: the call would be assigned by the run-time load-balancer, it could not be inlined, and some memory accesses would be assumed to be remote.

```
void incr_tree(NODE local * root, int k)
{ if (root != NULL)
    {^ incr_tree(root->left,k)@OWNER_OF(root->left);
       incr_tree(root->right,k)@OWNER_OF(root->right);
       incr_node(root,k);
    ^}
}
basic void incr_node(NODE * node, int k)
{  node->key = node->key + k;
}
```

Figure 2.5: Example of Local Pointers and Basic Functions

2.1.3 Library Functions and Primitives

The programmer may sometimes want to access blocks of data, and to copy blocks of data between processors. For these cases the primitive blkmov(source, dest, nbytes) is used primarily to copy arrays and records between processors. The primitive blkmov is the EARTH-C equivalent of the usual C library function memcpy.

Other useful built-in primitives include OWNER_OF(addr) which returns the processor owning the addr, NODE_ID which returns the currently executing processor number, and NUM_NODES that returns the number of processors allocated to the program.

Two sample list processing functions, written in EARTH-C, are given in Figure 2.6. In both cases the functions take a pointer to a list head, and a pointer to a node x, and return the number of times x occurs in the list. Figure 2.6(a) uses a forall loop to indicate that all iterations of the loop body may be performed in parallel. Since a **forall** loop must not have any loop-carried dependencies on ordinary variables, we have used the shared variable `count` to accumulate the counts. Shared variables must always be accessed via atomic functions and in this case we have used the built-in functions **writeto**, **addto** and **valueof**. Figure 2.6(b) presents an alternative solution using recursion. In this example we use a parallel sequence (denoted using `{^ ... ^}`) to indicate that the call to `equal_node` and the recursive call to `count_rec` can be performed in parallel.

```
int count(node *head, node *x)
{ shared int count;
  node *p;

  writeto(&count,0);
  forall(p=head; p!=NULL; p=p->next )
    if (equal_node(p,x)@OWNER_OF(p))
      addto(&count,1);

  return(valueof(&count));
}
int equal_node(node local *p, node *q)
{  return(p->value == q->value );
}
```

(a) iterative solution

```
int count_rec(node *head, node *x);
{ node *next;
  int c1, c2;
  if (head != NULL)
    { {^ c1=equal_node(head,x)
                  @OWNER_OF(x);
         c2=count_rec( head->next ,x);
      ^}
      return(c1+c2);
    }
  else
    return(0);
}

int equal_node(node *p, node local *q)
{  return( p->value == q->value);
}
```

(b) recursive solution

Figure 2.6: Example Functions Written in EARTH-C

As the target architecture for EARTH is a distributed-memory machine, the distinction between local memory accesses and remote memory accesses is very important. Local memory accesses are expressed in the generated lower-level threaded-C program as ordinary C variables that are handled efficiently, and may be assigned to registers or stored in the local data cache. However, remote memory references must be resolved by calls to the underlying EARTH runtime system. Thus, for remote memory accesses, there is the additional cost of the call to the appropriate EARTH primitive operation, plus the cost of accessing the communication network. In compiling the EARTH-C language, the compiler can assume that all direct (non-pointer) references to parameters and locally-scoped variables are local references. In contrast, unless further information is known, the compiler must assume that all indirect (via a pointer) references and all references to global variables are remote. In our program examples we underline all remote references.

Explicit local pointer declarations can be used to provide extra information to the compiler. In Figure 2.6(b) the call to `equal_node(head,x)` is specified to occur at the OWNER_OF(x). This means that within the body of `equal_node`, the second parameter can be assumed to be a local pointer.

2.2 The EARTH-McCAT C Compiler

The analyses presented in the thesis are incorporated into the EARTH-McCAT C compiler framework. The EARTH-McCAT C compiler takes an EARTH-C program, and generates a low-level threaded-C program, to be run on the EARTH-MANNA architecture. The overall structure of the compiler is given in Figure 2.7. The compiler is divided into three phases. Each phase is explained in detail in the following sections.

2.2.1 Phase I: standard transformations and analyses

The current EARTH-McCAT C compiler is built upon the McCAT C compiler framework. At phase I, it inherits the McCAT front-end, the AST-based intermediate representation–SIMPLE [HDE+92], and its advanced compiler analyses. Our development effort is thus significantly reduced, and we benefit from a rich set of standard and advanced compiler analyses in McCAT as well.

The McCAT compiler

McCAT was designed as a sequential compiler framework for studying modern compiler analyses and transformations. It takes a sequential C program, and generates an AST-based intermediate representation SIMPLE [HDE+92]. At this point, programs have been made structured via goto-elimination [EH94], and each statement has been simplified into a series of SIMPLE statements.

A series of compiler analyses have been implemented at the SIMPLE level, including points-to analysis [EGH94], connection analysis [GH96a], shape analysis [GH96b], and read/write set analysis. These sophisticated analyses provide the information about side effects, and a set of locations read or written by the statement. The availability of this read/write information allows our analyses to be simple and efficient.

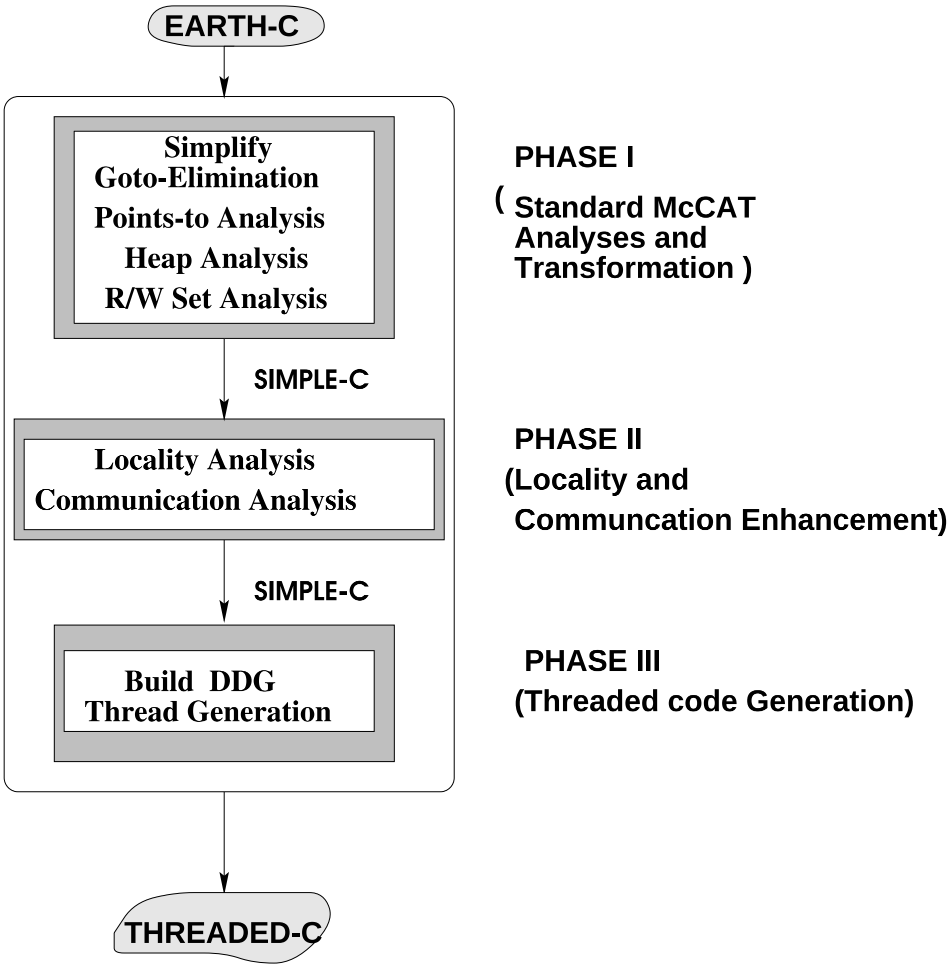


Figure 2.7: Overall Structure of the Compiler

EARTH-McCAT extensions

Based on the McCAT framework, minor changes are made for the compiler to accept an EARTH-C program, to record parallelism and locality information at the SIMPLE level, and to preserve the correctness of compiler analyses and transformations under parallel semantics. The changes occur in several modules. At the front-end level, the parser is modified to accept the EARTH-C constructs, and semantic checks for these constructs are inserted in the type-checking module. At the intermediate level, new constructs are introduced in the SIMPLE representation, in order to maintain the parallelism and locality information coming from an EARTH-C program. For instance, a new field `list.par_list` is added to a SIMPLE tree node, in order to distinguish among three kinds of statement sequences at the SIMPLE: PAR, SEQ and STRICT SEQ. The information may be retrieved using one of the macros: `TREE_IS_PAR`, `TREE_IS_STRICT` and `TREE_IS_SEQ`. Other EARTH-C constructs, such as parallel loops, basic functions, local pointers, and call sites, are all handled in a similar fashion.

Aside from the changes made to the SIMPLE structure, adjustments are also made for the sequential compiler analyses to maintain the correctness under the EARTH-McCAT semantics. According to the memory interference rule, no parallel computations will interfere unless through shared variables, thus the sequential analyses work correctly for computations without `shared` variables, which is the most often case. Consequently, changes are limited to shared variables. This is accomplished by integrating conservative analysis rules for shared variables into the analyses. Shereen Ghobrial's master thesis [Gho97] provides a detailed account of the support for shared variables and atomic functions in the EARTH-McCAT compiler.

2.2.2 Phase II: locality and parallelism enhancement

The analyses presented in the thesis are found in Phase II, where locality enhancement and communication optimizations are done. These analyses use the results of the analyses from Phase I, in order to transform the SIMPLE program representation into a semantically-equivalent, but more efficient program. The parallelization transformation at this stage detects when a loop can be safely declared as a parallel loop, based on heap and connection analyses. More information about the parallelization transformation can be found in [RGZ98].

2.2.3 Phase III: code generation

Phase III takes the transformed SIMPLE program from Phase II, partitions the statements into threads, and generates the threaded-C code for the target architecture. By exposing more locality and optimizing remote communications in Phase II, we allow the thread generator to deal with fewer remote memory accesses; this should lead to fewer threads, fewer calls to EARTH primitives, and more efficient parallel programs.

In our low-level EARTH model, a thread is a sequence of instructions that execute continuously without interruption. Each thread may contain one or more remote communications. A remote communication is implemented using a split-phase transaction. That is, when a thread issues a remote operation, it does not wait for the return of the signal indicating completion, but continues the execution until it reaches a statement dependent on that remote message. At this point, the control is switched to another available thread to proceed. Taking advantage of the split-phase implementation, statements related via a remote data dependence are always partitioned into different threads, and synchronized through a synchronization slot, whereas several independent remote statements may be grouped into one thread, and issued in a fast pipelined fashion. This way, remote latency can be reduced through overlapping communications with the computation.

The EARTH-McCAT compiler uses a Data Dependence Graph(DDG) partitioning strategy to partition threads. A DDG is a directed graph, with nodes standing for statements, and arcs representing data dependence between statements. A program is associated with a set of DDGs, with one DDG generated for each statement sequence. A statement node can either be a basic statement, or a compound statement. The data dependences between statements are calculated using the results of Read/Write analysis in McCAT. Two statements `s1` and `s2` are not data dependent if the following conditions are satisfied:

$$(Read(s1) \cup Write(S1)) \cap Write(s2) = Empty$$
$$(Read(s2) \cup Write(s2)) \cap Write(s1) = Empty$$

The implementation of Read/Write analysis relies on the information provided by other McCAT analyses, in particular, array dependence analysis[JH94] and a set of alias analyses. The set of advanced alias analyses available in McCAT, namely, points-to analysis[EGH94], connection analysis[GH96a] and heap analysis[Ghi97], help generate precise DDGs, and expose parallelism even in the presence of pointers. The existence of these analyses is crucial for our dynamic data structure based applications.

Each statement node in a DDG is labeled with one of the types: LOCAL or REMOTE. The basic idea is that: a statement is given a LOCAL type if it requires no split-phase operations, whereas any statement containing a split-phase operation (remote read, remote write or remote function call) is given a REMOTE type. The type information is determined by the compiler through looking at the type and scope of each variable referenced. Direct references to replicated globals, locals, parameters, and indirect references via local pointers are local operations, while direct references to ordinary globals, and indirect references via ordinary pointers are remote operations. If a statement contains any variable reference that is remote, the statement is labeled as REMOTE. A REMOTE data dependency occurs when a statement node depends on a statement node of REMOTE type. An example of a DDG graph is given in Figure 2.8, where REMOTE data dependences are drawn with solid arcs, LOCAL data dependences with dotted ones, and statement nodes of REMOTE type are shaded. As the indirect references to both parameter `r` and ordinary pointer `node` are remote operations, the corresponding statements are of REMOTE type.

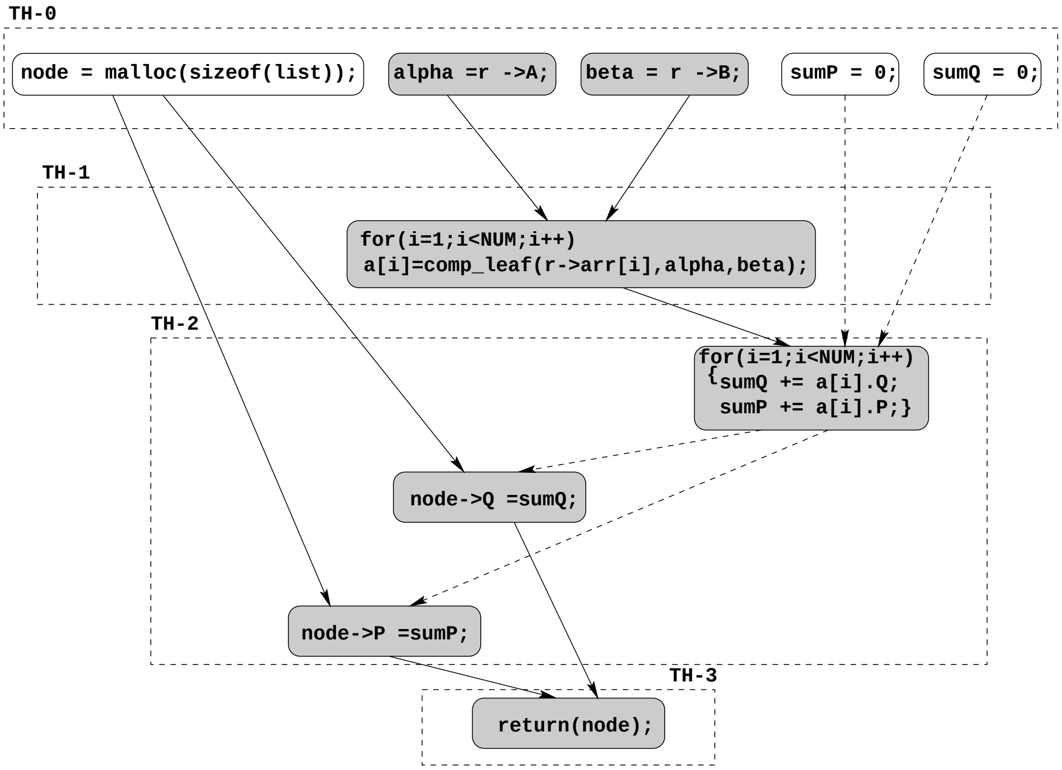


Figure 2.8: DDG and List Scheduling

The thread partitioning strategy assigns each node in a DDG the earliest thread in which the statement could be placed. In compliance with the split-phase rule, the fundamental observation is that if a node P has a REMOTE type, and $thread\_num(P) = i$, then the earliest thread number for all successors of P is thread $i + 1$. Similarly, if node Q has many REMOTE ancestors, say $P_1 \ldots P_k$, then the earliest thread number for Q is $max(thread\_num(P_1) + 1, \ldots, thread\_num(P_k) + 1)$. Based on these observations, we can assign earliest thread numbers to each node in the DDG using a top-down search. Since we place each statement in the earliest possible thread, the total number of threads needed for the statement sequence is minimized and thus the average thread length is maximized. By applying this partitioning strategy, four threads generated in Figure 2.8. We use a dotted box labeled with a thread number to include all statements assigned to the same thread.

Accoding to this thread partitioning strategy, the threads partitioned are determined by the number of remote communicaitons in a DDG. Therefore, the fewer remote communications a DDG has, the fewer threads it generates, the less thread switching overhead is incurred, which leads to the generation of more efficient threaded code. In Figure 2.8, if the compiler is able to detect that `node` is allocated on a local heap, and not modified through the function call `comp_leaf`, all indirect references to `node` become local operations, thus all the corresponding statements will be of LOCAL type. Figure 2.9 shows the improved DDG. With fewer REMOTE nodes in a DDG, fewer threads are generated. As a result, the number of threads generated in Figure 2.9 has been successfully reduced from the original four to three. This improvement on locality can be achieved either by a programmer though local declarations, or by the locality and communication techniques presented in the thesis.

After assigning the earliest thread number, we use a list scheduling strategy to actually generate the threads. Statements with the smallest thread number are chosen first. This implies that all statements with earliest thread number i will be scheduled before statements with earliest thread number $i + 1$. Furthermore, we schedule all statements with the same earliest thread number i together into thread number i. If there are many operations ready with the same thread number, we schedule the split phase operations as early as possible, and then do all the computation that is local. By issuing remote read/write operations early, local computation is overlapped with the communication. By issuing remote function calls early, function-level parallelism is exposed as early as possible.

This basic list scheduling approach is applied hierarchically for each DDG in order to generate threads for each function body. The scheduler is applied first to the outermost statement sequence, and then recursively applied to the bodies of any remote compound statement. When compiling the bodies of `forall` loops, the code is optimized by pulling the synchronization associated with the last thread in the

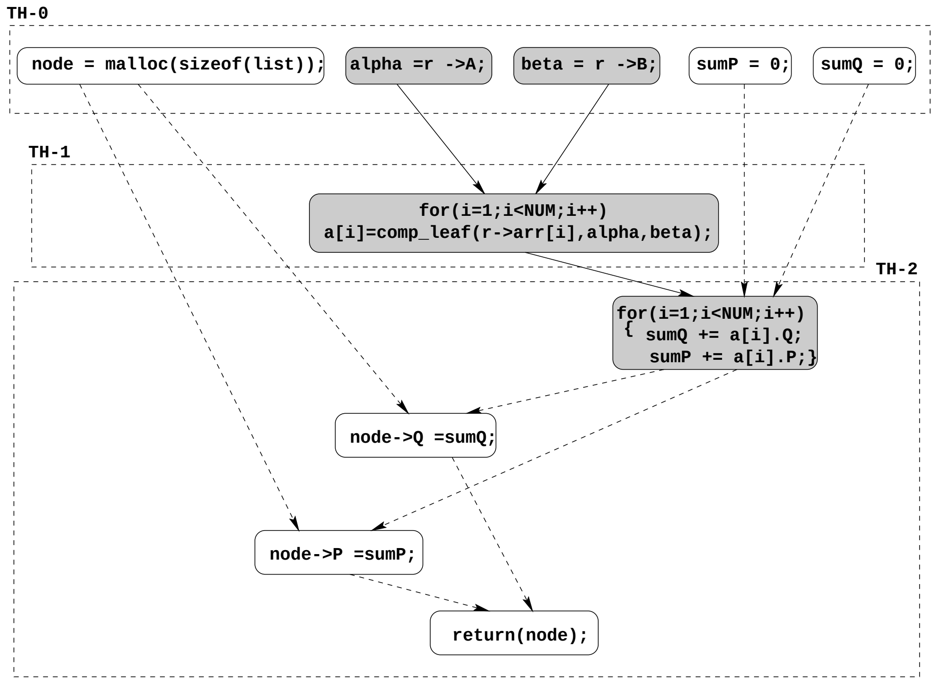


Figure 2.9: Improved DDG

body outside of the loop. Thus, for loops which have only one thread in the body, multiple iterations can run in parallel and a barrier synchronization is enforced at the end of the loop.

Once the threads have been partitioned, the threaded-C code is generated. Each *remote* function is compiled as a `THREADED` function, using the keyword `THREADED` at the beginning and `END_FUNCTION` at the end. Each thread starts with the appropriate thread label, and terminated with the `END_THREAD();` statement. Each remote read is translated into an Threaded-C `GET_SYNC` and each remote write is translated into a `DATA_SYNC`. Calls to `basic` functions are not modified, but calls to `remote` functions are compiled into `TOKEN` or `INVOKE` calls. The `TOKEN` mechanism is used when the run-time load balancer should be used, and the `INVOKE` mechanism is used when an explicit processor number is given. Atomic functions operating on shared variables are replaced by built-in single-threaded functions, which guarantees the atomic semantics. Figure 2.10 shows the threaded code generated respectively for the DDGs in Figure 2.8 and Figure 2.9. Comparing Figure 2.10(b) with Figure 2.10(a), we found not only some expensive remote communications are replaced with fast local communications, but fewer threads are generated as a result of better locality information.

```
THREADED Comp_Tree(list *ret_parm,
           SLOT *rsync_parm,
           tree *r)
{ SLOTS SYNC_SLOTS[4];
  int alpha, beta;
  list *node, a[NUM];
  int i, sumP, sumQ;

/* THREAD_0:; */
  INIT_SYNC(0, 2, 2, 1);
  INIT_SYNC(1, NUM, NUM, 2);
  INIT_SYNC(2, 1, 1, 3);
  INIT_SYNC(3, 1, 1, 4);

  sumP = 0;
  sumQ = 0;
  node = (list *)malloc(sizeof(list));
  GET_SYNC_L(&(r->A), &alpha, 0);
  GET_SYNC_L(&(r->B), &beta, 0);
  END_THREAD();

THREAD_1:;
  for (i=0; i< NUM; i++)
    TOKEN(Comp_Leaf, &a[i], SLOT_ADR(1),
                     r->arr[i], alpha, beta);
  END_THREAD();

THREAD_2:;
  for (i=0; i< NUM; i++)
   { sumP += a[i].P;
     sumQ += a[i].Q;  }

  DATA_SYNC(sumP, &(node->P), 2);
  END_THREAD();

THREAD_3:;
  DATA_SYNC(sumQ, &(node->Q), 3);
  END_THREAD();

THREAD_4:;
  DATA_RSYNC_L(node, ret_parm, rsync_parm);
  goto end_of_function;

end_of_function:;
  END_FUNCTION();
}
```

(a) Threaded-C Code

```
THREADED Comp_Tree(list *ret_parm,
           SLOT *rsync_parm,
           tree *r)
{ SLOTS SYNC_SLOTS[2];
  int alpha, beta;
  list *node, a[NUM];
  int i, sumP, sumQ;

/* THREAD_0:; */
  INIT_SYNC(0, 2, 2, 1);
  INIT_SYNC(1, NUM, NUM, 2);

  sumP = 0;
  sumQ = 0;
  node = (list *)malloc(sizeof(list));
  GET_SYNC_L(&(r->A), &alpha, 0);
  GET_SYNC_L(&(r->B), &beta, 0);
  END_THREAD();

THREAD_1:;
  for (i=0; i< NUM; i++)
    TOKEN(Comp_Leaf, &a[i], SLOT_ADR(1),
                     r->arr[i], alpha, beta);
  END_THREAD();

THREAD_2:;
  for (i=0; i< NUM; i++)
   { sumP += a[i].P;
     sumQ += a[i].Q;  }
  node->P = sumP;
  node->Q = sumQ;
  DATA_RSYNC_L(node, ret_parm, rsync_parm);
  goto end_of_function;

end_of_function:;
  END_FUNCTION();
}
```

(b) Improved Threaded-C Code

Figure 2.10: Threaded-C Code

2.3 The EARTH Execution Model

In order to evaluate the analyses presented in the thesis, experiments are performed on the EARTH system. Figure 2.11 illustrates an EARTH model, in which a multiprocessor consists of multiple EARTH nodes and an interconnection network [HMT+96, MGH+96]. Each EARTH node consists of an *Execution Unit* (EU) and a *Synchronization Unit* (SU) linked together by buffers. The SU and EU share a local memory, which is part of a *distributed memory* architecture in which the aggregate of the local memories of all the nodes represents a global memory address space.

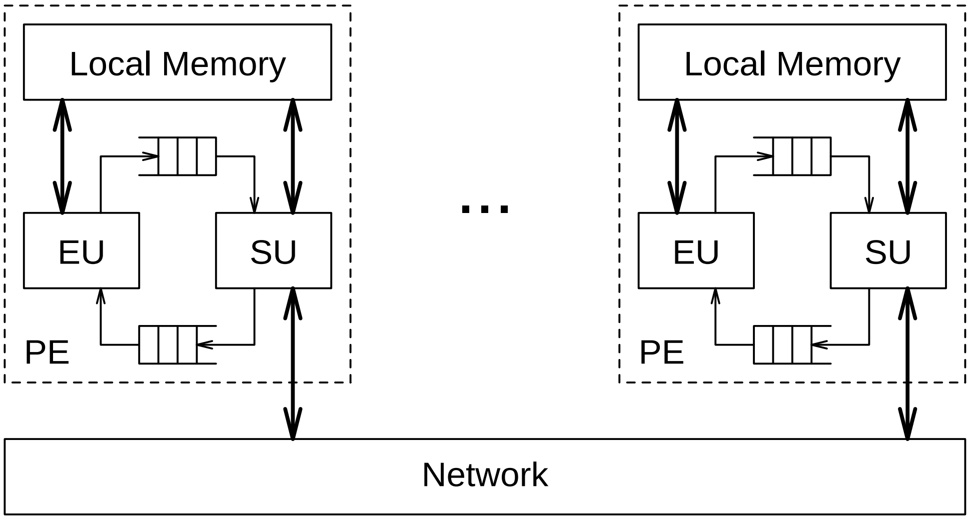


Figure 2.11: The EARTH Architecture

The EU fetches a thread from the ready queue, processes instructions in the thread, and completes the execution before moving to another thread. The EU interacts with the SU and the network by placing messages in the event queue. The SU handles these messages, plus messages coming from remote processors through the network. It responds to remote synchronization commands and requests for data, and also determines which threads are to be run and adds their thread IDs to the ready queue.

The EARTH system is built on top of the MANNA (Massively parallel Architecture for Non-numerical and Numerical Applications) developed at GMD-FIRST in Berlin, Germany [BGSP94]. Each MANNA node consists of two Intel i860 XP CPUs, clocked at 50MHz, 32MB of dynamic RAM and a bidirectional network interface capable of

EARTH Operation	*Sequential Local*	*Sequential Remote*	*Pipelined Local*	*Pipelined Remote*
Read word	2648 ns	7109 ns	1137 ns	1908 ns
Write word	2560 ns	6458 ns	1060 ns	1749 ns
Blkmov word	5296 ns	11600 ns	3188 ns	2602 ns

Table 2.1: Cost of Communication on EARTH-MANNA

transferring 50MB/S in each direction. The two processors on each node are mapped to the EARTH EU and SU.

The EARTH runtime system supports efficient remote data operations. Table 2.1 shows the cost for three data operations on the EARTH-MANNA, each under four different usages. The cost includes both the network delays and the synchronization overhead. In general, remote operations are more expensive than local operations, as they necessitate network accesses. Table 2.1 shows two typical usages, sequential and pipleined. Sequential numbers indicate the time it takes to complete an operation sequentially. In the pipelined case, on the other hand, operations are issued as early as possible, without the need to synchronize before issuing the next operation. Consequently, the numbers for the pipleined case are lower, as EU, SU and network can all work in parallel. Usually, a typical latency varies between these two cases, depending on the location of the referenced data.

Besides remote data operations, function invocations are also implemented in an efficient way on the EARTH-MANNA. The EARTH system supports four different types of function invocations: normal sequential call, sequential call of a threaded function, invocation on a specific remote node, and invocation on an arbitrary node through dynamic load balancing. Like in the sequential implementation, frames for normal functions are allocated on a runtime stack. This was also the case for the standard system libraries. However, frames for threaded functions, for instance, functions that contain a thread or that invoke other threaded functions, are dynamically allocated from the heap. This is necessary as they can run in parallel with each other and therefore terminate in an arbitrary order. Through maintaining a freelist for most common frame sizes, the typical overhead for dynamic frame allocation is reduced to 200 ns. Amortized by the low cost of normal functions, the average cost for function invocations is quite reasonable.

In spite of the efficient implementation of the EARTH-MANNA system, remote operations remain significantly more expensive compared with local operations. The cost comes from both local overhead and network accesses. As a result, reducing remote operations is essential to the efficiency of a program running on a distributed memory architecture. The subsequent chapters will show how to eliminate or overlap these overheads through a series of analyses, thus improving the overall performance.

2.4 Benchmark Suite

In order to evaluate the performance of the analyses presented in this thesis, we have performed experiments on a set of benchmarks. The benchmark suite used are from the Olden suite [RCRH95], as described in Table 2.2. All the benchmarks use either trees, lists, dynamically-allocated arrays or DAGs as the main data structures. Thus the benchmark suite is suitable to evaluate our locality and communication analyses focused on pointers.

Benchmark	Description	Problem Size
power	Optimization Problem based on a variable k-nary tree	10,000 leaves
perimeter	Computes the perimeter of a quad-tree encoded raster image	Maximum tree-depth 11
quicksort	Parallel version of quicksort	256K integers
tsp	Find sub-optimal tour for traveling salesperson problem	32K cites
health	Simulates the Colombian health-care system using a 4-way tree	4 levels and 600 iterations
em3d	Models the propagation of electromagnetic waves through objects in three dimensions	60K nodes
voronoi	Computes the Voronoi Diagram of a set of points	32K points

Table 2.2: Benchmark Programs

Chapter 3

Locality Analyses

In the previous chapter, we presented an example to illustrate where better locality information leads to improved threaded code, with fewer remote accesses and fewer threads generated. This type of locality information may either come from a programmer, where he/she uses type expressions to declare local pointers, or from a compiler, where static analyses are performed. In our experiments with applications using irregular, dynamically allocated data structures, we found it tedious for a programmer to explicitly exploit all the locality. Often, the programmer was forced to make many function specializations, and to declare the appropriate pointer parameters and locally-scoped pointer variables as local pointers. Thus, in order to experiment with various locality approaches, the programmer needed to edit many places in his/her program, and to make several copies of the same function, each copy specialized for a particular type of locality. To ease the burden on the programmer, we present in this chapter some new compiler techniques to infer the locality of pointer variables, and then automatically produce the specialized versions of the functions. This allows the programmer to make very minimal changes to his/her high-level program in order to try various approaches to a problem. It also leads to shorter source programs because the programmer does not need to make several similar copies of the same function.

The main idea behind our approach is that the compiler uses the information about the context of function calls and memory allocation statements to infer when indirect memory references must refer to local memory. It then automatically creates specializations of functions with the appropriate parameters and locally-scoped variables explicitly declared as local pointers. This information is then used by the thread generator to reduce the number of remote operations required in the low-level threads.

The locality inference and specialization use static program analysis. That is, it only infers that a memory reference is local when it is sure to be local for all program executions. Another approach is to use dynamic (run-time) tests to determine if a particular memory reference is local or remote. Then, based on the result of this check, control branches to code specialized for local references, or code specialized for remote references. Since it is interesting to compare the static inference results with dynamic methods, we study two dynamic techniques. The first is a *fine-grain* technique, where dynamic checks are introduced for individual memory references. The second is a *coarse-grain* technique where dynamic checks are introduced for larger blocks of code.

The remainder of the chapter is organized as follows. Section 3.1 provides some examples to motivate the locality analysis, and Section 3.2 describes the static locality analysis and specialization. In section 3.3 we introduce the fine-grain and coarse-grain dynamic tests, and provide examples of where such tests can be inserted. In Section 3.4 we give experimental results for our set of benchmarks programs. Finally, in Section 3.5 we discuss some related work, and in Section 3.6 we give the conclusions.

3.1 Motivation

In the preceding chapter (Figure 2.10) we presented an example where better locality information could be used to improve the threaded code generated. In this section we present some more typical examples of where locality information is used in EARTH-C programs, and we show how locality analysis and specialization can lead to better programs. These examples should give the intuitive ideas behind the actual locality analysis as presented in Section 3.2. In each of the example programs, all remote variable references are underlined. Thus, a program with fewer underlined references exhibits more locality, and will be more efficient.

3.1.1 Pointers to Local Variables and Parameters

As outlined in Chapter 2.3, the underlying EARTH runtime system maps a function's activation frame to the processor executing the invocation, and thus it is safe to assume that parameters and locally-scoped variables are references to local memory. This assumption can be extended to pointer variables, if it can be shown that the pointer must point to locally-scoped variables and/or parameters. Figure 3.1(a) gives a somewhat contrived example that serves to illustrate the basic point. In function foo, pointer p points to x, and x is a parameter. Since all parameters are allocated in local memory, it is safe to assume that *p points to local memory. Pointer q points either to parameter x, or to locally-scoped variable y. Since both x and y are local, we can assume that *q is local as well. Figure 3.1(b) gives the localized version of function foo. Note that the indirections *p and *q are remote (underlined) references in the original version of foo, but are local references in the localized version.

```
int foo(int x)
{ int y, *p,
  int *q;
  y = expr();
  p = &x;
  if (expr())
    q = p;
  else
    q = &y;
  return( *p + *q );
}
```

(a) no locality inference

```
int foo(int x)
{ int y, local *p;
  int  local *q;
  y = expr();
  p = &x;
  if (expr())
    q = p;
  else
    q = &y;
  return(*p + *q);
}
```

(b) after locality inference

Figure 3.1: Locality for Pointers

3.1.2 Dynamic Memory Allocation

A rich source of locality information comes from the fact that dynamic memory allocation always allocates memory on the processor from which the allocator is called. Thus, if function f calls a memory allocation function like malloc, then the memory returned by malloc is local within the body of f. Consider the example in Figure 3.2(a). Without any locality inference, or type declarations, the compiler must assume that pointer t may refer to remote memory. Thus, as indicated by the underlined sections, all indirect references via t must be assumed to be possibly remote. However, one can note that t only points to the memory returned from malloc, and thus t can safely be declared as a local pointer, as illustrated in Figure 3.2(b). In this case, all memory accesses in the body of alloc_point can be assumed to be local.

```
node *alloc_point(double x, double y,
                          int colour)
{  node * t;
   t = malloc(sizeof (node));
   if (t != NULL)
     {  t->x  = x;   t->y  = y;
        t->colour  = colour;
     }
   return(t);
}
```

(a) no locality inference

```
node *alloc_point(double x,
          double y, int colour)
{  node local *t;
   t = malloc(sizeof (node));
   if (t != NULL)
     { t->x = x;  t->y = y;
       t->colour = colour;
     }
   return(t);
}
```

(b) after locality inference

Figure 3.2: Locality for Dynamic Memory Allocation

3.1.3 Mapping Computation to the `OWNER_OF` Data

The most common kind of locality information comes from the programmer mapping function invocations to the owner of a piece of data, using the `@OWNER_OF` expression. A typical example is given in Figure 3.3(a). The function `count_equal` recursively descends through a binary tree `t`, counting the number of nodes with value `v`. The first recursive call, to the left sub-tree, is not explicitly mapped to any particular processor, and so there is no locality information for it. However, the second recursive call is explicitly sent to the owner of the right sub-tree. This means that these invocations can assume that all references via pointer `t` are local. As illustrated in Figure 3.3(b), to express this properly, a specialized copy of `count_equal` must be created (called `count_equal_spec` in the example), and in that copy the parameter `t` is declared to be a local pointer, and thus all memory accesses in the body are local.

3.1.4 Mapping Computation to `HOME`

Another common method for mapping computation to specific processors is to use function calls of the form `f(...)@HOME`. This indicates that `f` should be invoked on the processor executing the call. From a locality standpoint, this gives us two kinds of information. First, if `f` returns a pointer value that is local within `f`, it must also be local within the body of function calling `f`. Second, if an argument to `f` is local in the caller, then the corresponding parameter must be local in the body of `f`.

Figure 3.4(a) gives an example, and Figure 3.4(b) gives the result of applying locality analysis. First note that we can infer that `t` is a local pointer in the function

```
int count_equal(tree *t, int v)
  { int c1, c2, c3;
    if (t == NULL)
      return(0);
    else
    {{^c1=count_equal( t->left,v);
       c2=count_equal( t->right,v)
        @OWNER_OF(t->right);
       c3=t->value == v;
      ^}
     return(c1+c2+c3);
     }
  }
```

(a) no locality inference

```
int count_equal(tree *t, int v)
  { int c1, c2, c3;
    if (t == NULL)
      return(0);
    else
    {{^c1=count_equal(t->left,v);
       c2=count_equal_spec(t->right,v)
            @OWNER_OF(t->right);
       c3 =t->value == v;
      ^}
     return(c1+c2+c3);
     }
  }
int count_equal_spec(tree local *t,int v)
  { int c1, c2, c3;
    if (t == NULL)
      return(0);
    else
    {{^c1=count_equal(t->left,v);
       c2=count_equal_spec(t->right,v)
               @OWNER_OF(t->right);
       c3=t->value == v;
      ^}
      return(c1+c2+c3);
      }
  }
```

(b) after locality inference and specialization

Figure 3.3: Locality Generated Using OWNER_OF

newnode using the ideas presented for dynamic allocation given in section 3.1.2. Thus, the two calls to newnode in f must return local pointers, and both p and q must be local pointers. Now, consider the call to lessthan in f. Since both arguments, p and q, are local pointers, the corresponding formals, a and b, in the body of lessthan must also be local pointers.

```
void f()
{ node *p, *q;
  p = newnode(expr1())@HOME;
  q = newnode(expr2())@HOME;
  ...
  if (lessthan(p,q)@HOME)
    { ... }
 }
node *newnode(int val)
 { node *t;
   t = malloc(sizeof (node));
   if (t != NULL)
      t->value = val;
   return(t);
  }
int lessthan(node *a, node *b)
 { return( a->val < b->val );
 }
```

(a) no locality inference

```
void f()
{ node local *p, local *q;
  p = newnode(expr1())@HOME;
  q = newnode(expr2())@HOME;
  ...
  if (lessthan(p,q)@HOME)
    { ... }
}
node *newnode(int val)
 { node local *t;
   t = malloc(sizeof (node));
   if (t != NULL)
     t->value = val;
   return(t);
  }
int lessthan(node local *a,
                 node local *b)
 { return(a->val < b->val);}
```

(b) after locality inference

Figure 3.4: Locality Generated Using HOME

3.2 Static Locality Analysis and Specialization

In the last section, we identified the language/program features which are the sources of locality information. In this section we present the complete algorithm and associated analysis rules for static locality analysis. The overall algorithm is presented in Figure 3.5. It works iteratively in two inter-related intra- and inter-procedural steps.

At the beginning of the analysis, all the functions in the program are considered as candidates for specialization and put in the set spclPool. Further, the locality attribute for all formal parameters and global variables is initialized to Remote, unless

```
fun locality_analysis(prog) =
  spclPool = allFuncs(prog); /* need to analyze all functions initially */
  initialize_locality(prog);
  while (spclPool != Empty)/*pool of functions to be analyzed is non−empty*/
   propagate_locality(spclPool);

fun propagate_locality(prog, spclPool) =
  foreach func in spclPool /* intraprocedural propagation */
    propagate_intraprocedural_locality(func);
    deletFromSplcPool(func, spclPool); /* no need to be analyzed again */
  foreach callSite in prog /* interprocedural propagation */
    callSpecialized = specializeCall(callSite);
    if (callSpecialized) /* if a newly specialized function is created */
      addToSpclPool(callSite.func, spclPool);/*new func needs to be analyzed*/

fun propagate_intraprocedural_locality(func) =
  localitySet = func.localitySet;
  foreach assignmentStmt in func
   localitySet = locality_analyze_stmt(assignmentStmt, localitySet);
  foreach callStmt in func /* handle function calls */
   localitySet = locality_analyze_call(callStmt, localitySet);
  func.localitySet = localitySet;

fun specializeCall(callSite)
  if (callSite.type == @HOME || callSite.type == @OWNER_OF)
   /* find which params will be Local in the callee function */
   funcLocality = mapParams(callSite, callSite.func, callSite.func.localitySet);
   /* if a specialzed version already exists with same locality set */
   if (specialFuncExists(callSite.func, funcLocality))
    callSite.func=specialFunc(callSite.func, funcLocality);/*rename function call*/
    return 0; /* no new function is created */
   else
    callSite.func = createSpclFunc(callSite.func); /* new func for callSite */
    callSite.func.localitySet = funcLocality;/*initial locality for new func */
    return 1; /* a new specialized function is created */

fun initialize_locality(prog) =
  foreach func in prog
   /* conservatively assume all parms and globals point−to remote memory */
   initLocality = setParmsGlobalsRemote(func);
   func.localitySet = initLocality;
```

Figure 3.5: Overall Algorithm for Locality Analysis

the programmer has given explicit local pointer declarations. For pointers explicitly declared as local in the program, the locality attribute is initialized to Local. After this initialization, the analysis proceeds in the following two steps.

Step I: This step individually analyzes each function in the pool of functions to be specialized (`spclPool`). It starts with the current locality attribute of variables, and propagates this information throughout the procedure using a flow-insensitive intraprocedural approach. The details of this step are given in section 3.2.1.

Step II: This step performs interprocedural propagation of locality, and procedure specialization when applicable. It looks at each call site in the functions belonging to `spclPool`, which is called with either the `@HOME` or `@OWNER_OF` primitive. Based on the locality information at the call-site, it infers locality information for formal parameters of the callee function as illustrated in sections 3.1.3 and 3.1.4. If a specialized version of the callee function with this locality already exists, the call-site is modified to invoke this function instead. Otherwise, a newly specialized version of the callee function is created for the given call-site. The locality attributes of the parameters of the newly-created function are appropriately initialized. For a call site like `foo(x, ...)@OWNER_OF(x);`, function `foo` will be invoked where the data of `x` is located, thus the points-to locality of the formal parameter corresponding to `x` is initialized to Local, while those of other formal parameters are conservatively initialized to Remote. The handling for the `@HOME` primitive is different. According to the semantics, a function with a `@HOME` call site will be invoked on the same processor as its caller function, therefore all the formal parameters inherit the points-to locality of their corresponding actual parameters. As call-sites within the specialized function can trigger further specializations, the newly-created function is put in `spclPool`.

At the end of step II, if `spclPool` is non-empty we go back to the first step. Clearly this process will terminate because we have only a finite number of functions/parameters that can be specialized, and specializations always add locality information.

In the actual implementation, we just create a new locality context to represent a specialized function, and we do not actually create the complete new function. The decision to actually create specialized functions is taken after the analysis, depending upon the benefit achievable from a particular specialization. The details of the specialization step are given in section 3.2.3.

3.2.1 Intraprocedural Locality Propagation

We perform intraprocedural propagation of locality information using type-inference techniques [Hen91], which have been previously adapted to perform almost linear points-to analysis[Ste96]. The basic idea of the type inference algorithm is to partition program variables into a set of equivalence classes. To achieve this classification, a merging-based approach is used.

For example, a simple assignment `x = y`, leads to assignment of the same type class to variables `x` and `y`, or in more general terms merging of the current type classes of `x` and `y`. If one wants to collect points-to information instead, the assignment would lead to merging of the points-to classes of variables `x` and `y`, where the points-to class of a variable contains the set of locations it may point to at runtime. Fast union/find data structures can be used to make merging fast. This is the technique used by Steensgaard [Ste96].

To collect locality information, we enhance this technique by attaching an additional locality attribute with each points-to class. The locality attribute can have one of the three possible values: (i) $\perp$: indicating that the locality information is not yet determined, (ii) Local: all locations are *definitely* allocated on *local memory*, and (iii) Remote: some locations *may be* allocated on *remote memory*. When two points-to classes are merged, the new locality attribute is obtained by merging the locality attributes of the two classes using the merge operator $\bowtie$ defined as follows:

$\bowtie$	$\perp$	*Local*	*Remote*
$\perp$	$\perp$	*Local*	*Remote*
Local	*Local*	*Local*	*Remote*
Remote	*Remote*	*Remote*	*Remote*

Below we give a small program fragment, and the points-to and locality information obtained by the type-inference based algorithm for it.

```
{ int *a, *b, *c;
  int x, y, z;
S1: if (cond)
      a = &x;
    else
      b = &y;
S2: a = b;
S3: c = &z;
}
```

The points-to and locality information at different program points is as follows:

After S1	PointsToClass(a) = {x, Local} PointsToClass(b) = {y, Local} PointsToClass(c) = {Empty, ⊥}.
After S2	PointsToClass(a) = {x, y, Local} PointsToClass(b) = {x, y, Local} PointsToClass(c) = {Empty, ⊥}.
After S3	PointsToClass(a) = {x, y, Local} PointsToClass(b) = {x, y, Local} PointsToClass(c) = {z, Local}

The locality attribute is Local for all three pointers as they contain addresses of local variables. One can also note that the information provided is flow-insensitive, and there is no kill information (otherwise `x` should not be in the points-to class of `a` or `b` after statement S2). Thus the final information after S3 is conservatively valid for the entire program fragment.

Our locality analysis uses the above type-inference based algorithm in an intraprocedural setting. The focus of the analysis is on accurately computing locality attributes, and not on computing complete points-to information. Thus, our analysis does not account for points-to information that holds due to aliasing between parameters and globals. However, since we make the worst case assumption about the locality of parameters and globals, this loss of information does not affect the correctness of our technique. Further, we have found that this loss of information does not affect the quality of the locality information that we find. Thus, it appears that this very inexpensive intraprocedural propagation is a good choice.

In the following paragraphs we provide detailed rules for the intraprocedural locality analysis. Our analysis is performed at the SIMPLE intermediate representation of the EARTH-McCAT compiler. The SIMPLE representation provides eight basic statements that can affect points-to/locality information. Below, we provide locality analysis rules for each statement.

Address Assignment

`x = &y`: Variable `x` points to variable `y`, so we merge the points-to class of `x` with the class to which variable `y` belongs.

```
CreateClass(y)
if (IsGlobal(y))
   Locality(Class(y)) = Remote
else
   Locality(Class(y)) = Local
Merge(PointsToClass(x), Class(y))
```

Dynamic Allocation

`x = malloc()`: For each statement S containing a malloc() call (or a related memory allocation call), we create a new variable called HeapS, and also create a class for it. The locality attribute of this class is initialized as Local. This is done because the EARTH programming model requires a malloc call to always be allocated memory from the local processor. After creating this new class, we merge it with the points-to class of variable `x`, giving the following rule:

```
HeapS = CreateHeapVar(S)
CreateClass(HeapS)
Locality(Class(HeapS)) = Local
Merge(PointsToClass(x), Class(HeapS))
```

Pointer Assignments

The statements belonging to this category include `x = y`, `x = y op z`, `x = *y`, `*x = y`, and the rules for analyzing them are discussed below.

For the statement `x = y op z`, we merge `x` with both the operands `y` and `z` on right-hand side. This is conservatively safe, and the result does not depend on the operation being performed, or the type of the operands.

For the statement `*x = y`, we need to follow an additional level of indirection on left-hand side. We need to know what `x` points to to perform the appropriate mergings, i.e. find the points-to-class of the points-to-class of `x`. If such a class does not yet exist, we simply create such a class, which gets filled in as the analysis proceeds. The same argument applies to statement `x = *y` with respect to its right-hand side. The following table summarizes these rules.

`x = y`	Merge(PointsToClass(x), PointsToClass(y))
`x = y op z`	Merge(PointsToClass(x), PointsToClass(y)) Merge(PointsToClass(x), PointsToClass(z))
`x = *y`	Merge(PointsToClass(x), PointsToClass(PointsToClass(y)))
`*x = y`	Merge(PointsToClass(PointsToClass(x)), PointsToClass(y))

Function Calls

`x = f(a1,...,an)[@expr]`: A function call can considerably affect locality information. By using pointer arguments and global variables, it can modify the locality attribute, of any set of points-to classes. If we handled function calls in the same way as Steengaard's points-to algorithm, we would often make worst-case assumptions for function calls, and lose many potential opportunities for specializations. To avoid always making worst-case assumptions at function calls, locality analysis uses the results of interprocedural read-write (mod/ref) analysis, which is computed by our read-write set analysis in Step I of the compiler (refer to Figure 2.7 in Section 2.2). Based on the read-write information we have two important cases, one when the function does not write to visible pointers, and the other when the function does write to visible pointers. In the first case we get very precise answers, and in the second case we get conservative answers. However, we find that the first case is most common. The treatment of each case is as follows:

Case I: The function call does not write to any pointer variable visible in the caller (including globals). This guarantees that the call does not affect the points-to and hence also locality information in the caller. In this case the statement statement `x = f(a1,...,an)[@expr]`, can only affect the points-to relationship/locality attribute of variable `x`. The locality attribute of the points-to class of `x` is updated depending upon the locality attribute of the points-to class of `return_f` (`return_f` is a symbolic name which represents the value returned by a function `f`), and the optional `@expr` used for the call.

If the function `f` is a basic function, or called `@HOME`, and `f` returns a pointer (`return_f`) pointing to Local, i.e. Locality(PointsToClass(`return_f`)) is Local, it implies that the location pointed to by `return_f` resides on the same processor. In this case, the new locality attribute of points-to class of `x` is obtained by merging it with Locality(PointsToClass(`return_f`)). Otherwise it is simply assigned Remote, as per the rule below.

```
if (expr == HOME Or IsBasicFunc(f))
  Locality(PointsToClass(x)) ⋈=
    Locality(PointsToClass(return_f))
else
   Locality(PointsToClass(x)) = Remote
```

Case II: The other alternative is that the function call possibly writes to pointer variables of the caller. In this case we make worst-case assumptions and set the locality attribute of points-to classes of all arguments as Remote. We do it recursively for points-to classes of variables reachable via indirection on arguments : i.e. we also set Locality($PointsToClass^*$(arg)) as Remote. We do not need to consider globals here as we already initialize their locality attribute as Remote at the start of the analysis.

3.2.2 Interprocedural Locality Propagation

At each step, after the intraprocedural locality propagation for each function in the `spclPool`, we examine all call sites within these functions, to see if there is any new locality information that will trigger another round of the intraprocedural locality propagation.

An example in Figure 3.6(a) is used to illustrate the iterative process. At the beginning, the `spclPool` contains both functions `count_eq` and `is_equal`. After the first round of intraprocedural locality propagation, we get the locality map for these functions in Figure 3.6(b). Note that without any call site information, both parameters `t` and `p` are conservatively initialized to point to Remote. Then we check the call sites for these functions. For the call site `@OWNER_OF`, we create a specialized function `count_eq_spec`, with its parameter `t` initialized to point to Local, and put the specialized function into `spclPool`. No specializations are created for the call site `@HOME`, since its parameter `p` maintains the points-to locality of `t`, which is Remote. The second round of interprocedural locality propagation produces a locality map for `count_eq_spec` in Figure 3.6(c), where the points-to locality for `t` remains Local. Now we look at all the call sites within this function. For the call site `@OWNER_OF`, we will create another specialized version, for which we already had one with the same locality context, thus nothing is done. For the `@HOME` site, with the new points-to locality for the argument `t`, the parameter `p` gets a new points-to locality Local as well, which leads to a specialized version `is_equal_spec` being created and put into `spclPool`. After the third round of interprocedural locality propagation, we get the locality map for `is_equal_spec` in Figure 3.6(d). As the function contains no function calls, no new specializations is created, and `spclPool` remains empty, and the whole process stops.

3.2.3 Specializing Functions

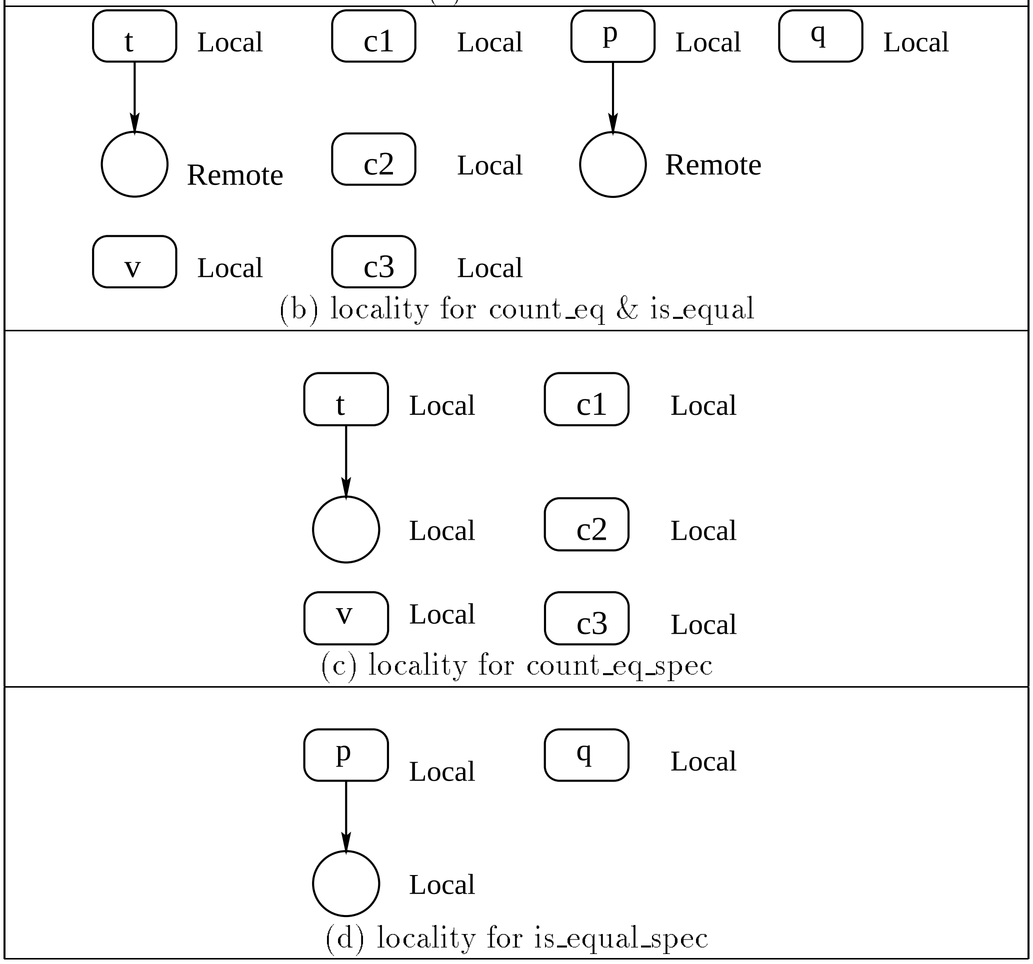
After completion of Step II of the algorithm, we have computed a set of possible specializations, and the associated locality. We use static estimates of the number of remote accesses saved to decide which specialized functions to actually create. For a given locality context of a function, we compute the following data: (i) weight of the corresponding call-site that reflects its potential execution frequency, and (ii) count of the remote accesses that can be eliminated by creating the specialized function. A specialized function is created for the given locality context if we find its (Weight * Count) estimate greater than a threshold.

To compute the weight for call-sites, we first initialize the weight of all call sites to one. For each loop or recursion cycle, in which a call-site is embedded, its weight is multiplied by ten. The count of remote accesses saved is similarly estimated. A simple remote access saved is counted as one, while a remote access saved inside a loop is counted as ten. Further if some call sites inside the specialized function can also be specialized, we also add the number of remote accesses saved from this chain-specialization to the count.

There are many ways to choose a threshold, for example, using profile information to set a threshold for each benchmark. In our experiments, we set the threshold to be 1, which means we will specialize a function when there is at least one remote access saved. This proves to be good enough for our benchmarks. In fact, for our benchmarks the lowest estimated frequency was 20, and so all specializations are almost certain to be worthwhile. Although, in general, the number of specialized versions of a function is exponential in the number of its parameters, we are only interested in pointer parameters, and we specialize a function only when one parameter is a possible local reference based on the function invocation site information. Thus, a function with `m` pointer parameters will have at most 2^m kinds of specializations. Therefore, if a program has `n` functions with the call site information, and the maximum number of pointer parameters for these functions are `m`, the upper bound of the number of specializations will be $n * 2^m$, which is reasonable for our benchmarks. For larger benchmarks the user may want to give a larger threshold as a parameter to the compiler, and this would reduce the number of specializations.

```
int count_eq(tree *t, int v)
  { int c1, c2, c3;
    if (t == NULL)
      return(0);
    else
      { c1 = count_eq( t->left,v);
        c2 = count_eq( t->right,v)@OWNER_OF(t->right);
        c3 = is_equal(t, v)@HOME;
        return(c1+c2+c3); } }
int is_equal(tree *p, int q)
  { if (p != NULL && p->value == q)
      return(1);
    else  return(0); }
```

(a) source code



(b) locality for count_eq & is_equal

(c) locality for count_eq_spec

(d) locality for is_equal_spec

Figure 3.6: Example of Interprocedural Locality Propagation

3.3 Dynamic Locality Checks and Specialization

The techniques presented in the previous section were based on static analysis techniques. These techniques indicate when a memory reference is guaranteed to be local, for all program executions. However, there may be interesting cases where, at run-time, the same static reference is sometimes local and sometimes remote. There may also be cases where the reference is always local, but a static analysis cannot determine that fact. Thus, another approach is to introduce dynamic locality checks into the program. In Section 3.3.1 we present a very naive technique that introduces fine-grain dynamic checks for all indirect memory references, and in Section 3.3.2 we give a more realistic method where dynamic checks are inserted for larger program segments.

3.3.1 Fine-grain dynamic tests

Consider the example program in Figure 3.7(a). If we have no information about the invocation site of `is_same_node`, then we cannot determine if parameters `n` and/or `m` point to local memory. However, as indicated in Figure 3.7(b), the compiler could insert dynamic checks using the primitive IS_OWNER in order to determine if the location is local. For the case that it is local, a cast is introduced to indicate that a cheap local memory instruction can be generated. Note that such fine-grain checks are not necessarily beneficial. For each memory reference there is the extra overhead of the IS_OWNER test, and the branch. If the memory reference is seldom really local, then the extra overhead will outweigh the benefit.

It should be noted that we do not actually implement fine-grain checks as a source-to-source transformation of the EARTH-C program. Instead, we treat the location as remote, and then insert the conditional when we generate the low-level threaded code. This was done to minimize the effect of this transformation on the thread generator. Too many conditional checks would lead to very short threads, which in turn hinders overall performance.

3.3.2 Coarse-grain dynamic checks

In general, fine-grain locality checks are not likely to be practical because they impose overhead for each memory reference. The objective of coarse-grain dynamic checks is to locate a segment of the program, with many related memory references,

```
int is_same_x(node *n, node *m)
  { int n_x, m_x;
    n_x =  n->x;
    m_x =  m->x;
    return(n_x == m_x);
  }
```

(a) no locality information

```
int is_same_x(node *n, node *m)
  { int n_x, m_x;
    if (IS_OWNER(n))
      n_x = ((node local *) n)->x;
    else
      n_x =  n->x;
    if (IS_OWNER(m))
      m_x = ((node local *) m)->x;
    else
      m_x =  m->x;
    return(n_x == m_x);
  }
```

(b) with fine-grain locality tests

Figure 3.7: Example of Fine-grain Locality Tests

that can benefit from the same locality check. Consider, for example, the function `count_equal` in Figure 3.8(a). This example is the same as in Figure 3.3(a), except that in this case there is no information at the call sites (no occurrence of @OWNER_OF) that allows static locality analysis and function specialization. Since static analysis will not work, we need to introduce a dynamic test to determine if parameter `t` refers to a local memory location or not. There are two obvious places to put the test, at the beginning of the function body, or inside the `else` clause.[1] Since placing the test inside the `else` clause minimizes the amount of code to specialize, it is a better choice. The result of the dynamic check and specialization is given in Figure 3.8(b). Note that one dynamic check can determine that three different memory references are local.

The overhead of coarse-grain checks is not as great as with fine-grain checks, but there is still the overhead of the IS_OWNER check, and the extra control flow branches which can disturb both the thread generator and the low-level instruction scheduler. If the dynamic check rarely succeeds in finding local references, then the overhead of this check will outweigh the benefit. Thus, one must be very careful in where dynamic checks are placed. In general, the checks should be placed: (1) where more than one memory reference will benefit from it, (2) where the reference is likely to be local

[1]Note that we cannot specialize the call sites, as in Figure 3.3(b), because there is no locality information given using the `@OWNER_OF` or `@HOME` primitive which would assign the invocation to a specific processor. When no invocation site is given the run-time system may map the invocation to any processor. Thus, dynamic locality checks must be made inside of the invoked function, and not at the call site.

```
int count_equal(tree *t, int v)
  { int c1, c2, c3;
    if (t == NULL)
      return(0);
    else
      { {^ c1 = count_equal( t->left,v);
           c2 = count_equal( t->right,v);
           c3 = t->value == v;
        ^}
        return(c1+c2+c3);
      }
  }
```

(a) no locality information

```
int count_equal(tree *t, int v)
  { int c1, c2, c3;
    if (t == NULL)
      return(0);
    else
      if (IS_OWNER(t))
        { {^ c1 = count_equal((tree local *) t)->left,v);
             c2 = count_equal((tree local *) t)->right,v);
             c3 = ((tree local *) t)->value == v;
          ^}
          return(c1+c2+c3);
        }
      else
        { {^ c1 = count_equal( t->left,v);
             c2 = count_equal( t->right,v);
             c3 = t->value == v;
          ^}
          return(c1+c2+c3);
        }
  }
```

(b) after coarse-grain check and specialization

Figure 3.8: Example of Coarse-grain Locality Checks

a significant fraction of the time, and (3) so that it encloses the smallest program segment that includes all of the memory references of interest.

3.4 Experimental Results

In order to evaluate our approach, we have experimented with six benchmarks described in Section 2.4. We have divided our results into two sections. In Section 3.4.1 we examine the performance of the static locality analysis and specialization as presented in Section 3.2, and in Section 3.4.2 we examine how well the dynamic techniques presented in Section 3.3 perform.

3.4.1 Effect of Static Locality Analysis and Specialization

In order to study the effect of static locality analysis, we produced experimental results for three versions of each benchmark: a simple EARTH-C version, a localized EARTH-C version and an advanced EARTH-C version. Henceforth, we will refer to them respectively as *simple*, *localized* and *advanced* versions. The *simple* version implements the benchmark with the *best data distribution* that we have discovered to date for these benchmarks, and exploits locality using the `@OWNER_OF` and `@HOME` primitives. It uses neither local pointers nor basic functions for this purpose. However, it can use basic functions for performing computations. Furthermore, we produce two *simple* versions for `power` benchmark, in order to measure the effect of call site information on locality techniques. These two versions are referred as `powerI` and `powerII` respectively. `PowerI` is a version in which major computation functions in the program are invoked using the `@OWNER_OF` primitive, while in `powerII`, these functions are called without any call site specifications, thus depending on the runtime system to distribute them across the architecture. The *localized* version is the benchmark obtained by applying our locality analysis and subsequent function specialization on the *simple* version. This version tries to find as many local pointers as possible. The *advanced* version is the hand-coded benchmark where the user tries to optimally exploit locality using local pointers, basic functions and other possible tricks. This version is based on the best efforts of our group to produce good performance by hand.

Static Measurements

Note that all three versions use the same general dynamic data distribution. However, the generated low-level program exploits locality to different degrees. Statically, we can divide memory references into those that **must** be local, and those that **might** be remote. Each reference that must be local is translated into an ordinary C variable reference, which may be allocated to registers by the target compiler, or may be cached by the architecture. Each reference that might be remote is translated into a call to the EARTH runtime system. These calls to the runtime system may be resolved into memory requests to local memory, or to remote memory, depending on the calling context. If at runtime, it resolves to a local memory reference, then we call this a *pseudo-remote* memory access. Note that *pseudo-remote* references are much more expensive than local references, but not as expensive as real remote references that must read or write data over the communication network.

Our locality analysis and specializations effectively introduce more static declarations for local pointer variables, and thus at runtime we execute fewer *pseudo-remote* references and more local references. As explained in the previous sections, this is done by automatically introducing local pointer declarations, and by introducing specialized versions of functions that capture locality for specific calling contexts.

In Table 3.1 we summarize the static effect of applying our locality analysis and specializations to the *simple* versions of our benchmarks. For each benchmark, the first two columns list the number of local pointer declarations introduced, and the number of function specializations made in producing the *localized* version of the benchmark. The third column gives the relative sizes (lines of code) of the *simple* and *advanced* versions. Note that *simple* versions of most benchmarks are shorter, and sometimes substantially shorter than the advanced, hand-specialized programs.

Dynamic Measurements

Although static program measurements are interesting, the important measurement is what happens at run time. In Table 3.2 we provide data on the actual execution time, in seconds, for all three versions of each benchmark. These experiments were performed on the EARTH-MANNA architecture as described in Chapter 2.3. The last two columns give the percentage speedup obtained over the *simple* version for the *localized* and *advanced* versions respectively. For example, the column labeled "Localized vs. Simple" reports $(T_{simple} - T_{localized})/T_{simple} * 100$, and the column labeled "Advanced vs. Simple" reports $(T_{simple} - T_{advanced})/T_{simple} * 100$. We provide data for benchmarks that run on 1, 8 and 16 processors.

Benchmark	#locals	#spcls	$\frac{size(Simple)}{size(Advanced)}$
powerI	41	3	0.99
powerII	11	0	1.00
tsp	28	4	0.71
perimeter	17	2	0.89
health	14	9	0.62
voronoi	5	1	0.99
em3d	35	0	1.00

Table 3.1: Static Measurements

In Table 3.3 we present the actual number of remote data accesses and remote calls performed by different versions of each benchmark. The last two columns of this table give the percentage reduction in the number of remote data accesses/remote calls over the *simple* version for the *localized* and *advanced* versions respectively. Note that the number of remote accesses performed is independent of the number of processors used for a given EARTH-C program, when we do not differentiate between a real remote access and a pseudo-remote access.

The data in Tables 3.2 and 3.3 indicates that the *localized* version always performs better than the *simple* version, i.e. our locality analysis is always able to identify some additional locality. Further, the percentage improvement can vary a lot, depending upon the benchmark. The *localized* version comes very close to the *advanced* version for four of the benchmarks: `powerI`, `perimeter`, `voronoi` and `em3d`. In the other three cases the *localized* version does give an improvement, but does not compete with the hand-coded advanced version. We analyze these results in detail individually for each benchmark below.

Power: This benchmark implements the `power` system optimization problem [Car96]. It uses a four-level tree structure with different branching widths at each level.

For `powerI` version, our locality analysis achieves 4% improvement over the *simple* version in execution time, which is quite close to the *advanced* version (5.5%). However, it is able to achieve an 89% reduction in the number of remote data accesses (Table 3.3). This happens because function calls in this benchmark are typically of the format `compute_node(node)@OWNER_OF(node)`, and the function performs numerous scalar data accesses of the form `(*node).item`. Our analysis captures the locality of the pointer `node` and eliminates all remote data accesses with respect to it.

Benchmark		Simple EARTH-C (sec)	Localized EARTH-C (sec)	Advanced EARTH-C (sec)	Localized vs. Simple (% impr)	Advanced vs. Simple (%impr)
powerI	1 proc	66.95	64.28	63.26	3.99	5.51
	8 procs	9.07	8.70	8.56	4.19	5.62
	16 procs	4.72	4.51	4.46	4.45	5.51
powerII	1 proc	67.76	66.52	63.26	1.83	5.51
	8 procs	9.77	9.60	8.56	1.54	5.62
	16 procs	5.23	5.15	4.46	1.53	5.51
tsp	1 proc	26.95	26.42	22.56	1.96	16.29
	8 procs	4.67	4.54	3.97	2.78	12.63
	16 procs	3.77	3.64	2.99	3.45	20.69
perimeter	1 proc	7.32	6.23	5.38	14.89	26.50
	8 procs	1.12	0.93	0.82	16.96	26.79
	16 procs	0.75	0.59	0.53	21.33	29.33
health	1 proc	144.09	143.84	143.80	0.17	0.20
	8 procs	42.74	36.45	26.63	14.72	37.69
	16 procs	33.88	28.54	20.25	15.76	40.23
voronoi	1 proc	7.27	6.19	5.52	14.86	24.07
	8 procs	1.31	1.11	0.98	15.27	25.19
	16 procs	1.04	0.87	0.77	16.35	25.96
em3d	1 proc	29.15	22.80	19.48	21.78	33.17
	8 procs	5.94	4.35	4.01	26.77	32.45
	16 procs	3.48	2.64	2.38	24.14	31.61

Table 3.2: Execution Time from Static Locality Analysis

Benchmark	Simple EARTH-C (# remote)	Localized EARTH-C (# remote)	Advanced EARTH-C (# remote)	Localized vs. Simple (% impr)	Advanced vs. Simple (% impr)
power	2293437	242835	203479	89.41	91.13
powerII	2293437	2268235	203479	1.10	91.13
tsp	3637853	2324521	719203	36.10	80.23
perimeter	2434987	1721448	1563505	29.30	35.79
health	3800488	3039636	945561	20.02	75.12
voronoi	26635962	22698979	18820971	14.78	29.34
em3d	16024851	7114341	5047097	55.60	68.50

Table 3.3: Remote Accesses Saved

The specialized version of the function `Compute_branch` from this benchmark, shown in Figure 3.9(a), illustrates this feature (note that we have framed those memory references that can be assumed to be local due to locality inference).

The significant reduction in remote accesses does not reflect in the execution time, as the benchmark spends most of the time in performing floating point operations, which far exceeds the time spent in data accesses. The *advanced* version uses basic functions to replace those threaded functions which have no remote data accesses, thus eliminates the overhead of allocating those function frames on the heap. This factor enables it to achieve slightly better speedup than the *localized* version.

For the `powerII` version, our locality analysis is able to insert the local declarations for some variables. However, without any specific call site information, the locality analysis could not exploit the locality further, thus it fails to eliminate most of the remote accesses coming from parameters. As a result, its percentage reduction on the number of remote accesses is only 1.0%, compared with 89% reduction for `powerI`. The difference demonstrates the effectiveness of the interprocedural locality exploration techniques, and its reliance on the call site specifications. With a little more programmer effort, the locality analysis effectively reduce most of the remote accesses in the benchmark. Nevertheless, as `power` is a computation intensive benchmark, we still achieve 1.5% improvement on the overall performance for `powerII`.

Tsp: This benchmark solves the traveling salesman problem using a divide-and-conquer approach based on close-point algorithm [Car96]. This algorithm first searches a suboptimal tour for each subtree(region) and then merges subtours into bigger ones.

```
Compute_branch_spec(branch local *br)
{ ...
  temp_237 = (theta_I * (*br).X );
  temp_236 = (temp_237 / (*br).R );
  temp_235 = (theta_R + temp_236);
  temp_233 = ( (*br).alpha * temp_235);
  new_pi_R = (pi_R + temp_233);
  temp_244 = (theta_R * (*br).R );
  temp_243 = (temp_244 / (*br).X );
  temp_242 = (theta_I + temp_243);
  temp_240 = (temp_241 * temp_242);
  new_pi_I = (pi_I + temp_240);
  ...
}
```

(a) Power

```
R_distance_spec(tree local *a, tree *b)
{ ...
  ax = a->x ;
  ay = a->y ;
  bx = b->x;
  by = b->y;
  dis = sqrt((ax-bx)*(ax-bx) +
             (ay-by)*(ay-by));
  ...
}
```

(b) Tsp

Figure 3.9: Sample Specialized Codes from Benchmarks

The tour found is built as a circular linked list sitting on top of the root nodes of subtrees.

This benchmark is irregular in nature and spends a significant amount time on data accesses. The *localized* version achieves 1.9–3.4% speedup from a 36% reduction in remote data accesses. Figure 3.9(b) shows a piece of specialized code in the program.

The *localized* version, however, fails to compete with the *advanced* version, which achieves upto 20% speedup resulting from an 80% reduction in remote data accesses. This happens because the linked lists representing tours, are distributed in segments and there are only very few links across processors. With the knowledge that an entire sublist is local, the *advanced* version exploits significant data locality by using basic functions to traverse these local sublists. Our locality analysis is not designed to identify locality of recursive fields. This kind of locality is implicit in the programmer's organization of the data, and is very difficult to find with compiler analyses.

Perimeter: This benchmark computes the `perimeter` of a quad-tree encoded raster image [Car96]. The unit square image is recursively divided into four quadrants until each one has only one point. The tree is then traversed bottom-up to compute the `perimeter` of each quadrant.

The *localized* version achieves 14–21% speedup and comes close to the *advanced* version. The *localized* version has 29% fewer remote accesses. This reduction more significantly affects the execution time than for `power` benchmark, because the benchmark does not involve much computation and spends most of its time in traversing the quad-tree and hence performing data accesses.

It is an irregular benchmark, and each computation requires accesses to tree nodes which may not be physically close to each other, as illustrated by the specialized version of the `Get_neighbor` function shown in Figure 3.10(a). Due to this characteristic, the *advanced* version cannot exploit additional locality using `basic` functions. The *localized* version thus competes very well with it.

Health: This benchmark simulates the Colombian health-care system using a 4-way tree [Car96]. Each village has four child villages, and a village hospital, treating patients from the villages in the same subtree. At each time step, the tree is traversed, and patients, once assessed, are either treated or passed up to the parent tree node. The 4-way tree is evenly distributed among the processors and only top-level tree nodes have their children spread among different processors.

```
Get_neighbor_spec(quadtree local *t)
{
  ...
  p = t->parent ;
  is_adj = adj(d,t->childtype);
  if ((p != NULL) && is_adj)
    q =Get_neighbor_spec(p,d)@OWNER_OF(p);
  else
    q = p;
  if (q && q->color==grey)
   { type = t->childtype;
     retval = R_child(q, reflect(d,type));
    }
  ...
}
```

(a) Perimeter

```
check_assess_spec(village local *v,list l)
{ ...
  while (l != NULL)
   { p = l->patient;
     p->time_left = p->time_left-1;
     label = v->label ;
     if (p->time_left == 0)
      { s = v->seed ;
        rand = my_rand(s);
        v->seed = (long)(rand * IM);
        label = v->label ;
       }
     ...
     l = l->forward;
    }
}
```

(b) Health

Figure 3.10: Sample Specialized Codes from Benchmarks

With the locality analysis, we are able to achieve up to 15% speedup, resulting from a 20% reduction in remote data accesses. This benchmark is similar to `power` in its call pattern. However, for a call of the format `foo(village_node, list_node)@OWNER_OF(village_node)`, it accesses recursive data structures through `list_node` (linked list of patients) as opposed to scalar data items like in `power`. Our analysis is able to eliminate all remote accesses with respect to the `village_node` like `label =(*village_node).label`, but not further accesses like `(*list_node).patient`, as list is distributed across all processors (Figure 3.10(b)).

Thus the *localized* version gets decent speedup over the *simple* version, but does not compete with the *advanced* version. With the knowledge that only top level tree nodes can have remote children, the *advanced* version eliminates most of the remote data accesses and calls. In this regard, this benchmark is similar to `tsp`.

Voronoi: The `voronoi` benchmark computes a Voronoi Diagram for a random set of points. The points are generated and stored in a binary tree, and the algorithm computes the Voronoi Diagrams of the two subtrees recursively, and merges them to form the final diagram.

The *localized* version achieves about 15% speedup in the performance, which is close to the *advanced* version. The *localized* version reduces the number of remote accesses by 14%. Similar to `perimeter`, the reduction in remote accesses for `voronoi` significantly affects the execution time, because the benchmark spends most of its time in traversing and merging the subtrees which results in many data accesses.

It is an irregular benchmark, and the merge phase walks along the convex hull of the two sub-diagrams, alternating between them in an irregular fashion. Therefore the *advanced* version could only exploit additional locality using `basic` functions at some low levels. This results in a slightly better percentage improvement 25% on the performance. Thus, the *localized* version competes very well with it.

Em3d: The `em3d` benchmark does the computation on an irregular bipartite graph to model the propagation of electromagnetic waves through objects in three dimensions. The graph contains nodes representing electric and magnetic field values (E nodes and H nodes, respectively). At each time step, new waves for the E nodes are computed from a weighted sum of the neighbouring H nodes, and new waves for the H nodes are computed similarly from the E nodes.

With the locality analysis, we are able to obtain 26% speedup in the performance. This improvement comes mainly from the intraprocedural locality exploitation, where we successfully reduce the number of remote accesses by 55% through local declarations. Unlike many other benchmarks implemented with tree data structures, `em3d` uses flat linked lists, which are distributed across the machine. Thus the benchmark lacks the locality that is often exploited by the *advanced* version using `basic` functions. As a result, the *localized* version matches well with the *advanced* version.

3.4.2 Effect of Dynamic Locality Tests

In order to compare the effect of static locality analysis with dynamic locality checks, we performed another set of experiments. For these experiments we used three versions of the same benchmarks: one with fine-grain dynamic checks, one with coarse-grain dynamic tests and one where we combined dynamic checks with static locality optimizations. We will refer to them respectively as: *fine-grain*, *coarse-grain* and *combined*. The *fine-grain* version is similar to the *simple* version, except that the compiler automatically inserts a run-time locality check for every remote reference, while in the *coarse-grain* version, a programmer carefully inserts a dynamic check only when a remote reference is likely to be local, and the check will benefit a number of remote references. For the *combined* version we first produced a specialized version using static locality analysis, and then inserted coarse-grain dynamic checks to any remaining remote references using the same strategy as in the dynamic case.

Figure 3.11 demonstrates a typical case for the static locality analysis, coarse-grain locality checks and the combined approach. In Figure 3.11(a) we show a piece of code extracted from the statically localized version of the `health` benchmark. In this case the calling site of `check_assess` patient has led to a specialized version where the first parameter, `v`, is known to be local. This means that the framed memory references can be assumed to be local. In Figure 3.11(b) we show an example of the coarse-grain dynamic checks. In this case we do not have any static locality inference information connecting calling sites with specializations, so parameter `v` cannot be assumed to be local. Using our strategy for inserting dyanmic checks, it appears to be worthwhile to know if `v` is local , and also if `p` is local within the while loop body. It is not worthwhile testing the locality of parameter `l` because this memory location is only dereferenced on the first iteration of the loop. Thus, dynamic checks are introduced at the beginning of the function body for `v`, and at the beginning of the while loop body for `p`. Figure 3.11(c) illustrates the combined approach. In this case the function is first specialized on parameter `v` using the static locality analysis, and then the dynamic check for `p` inside the while loop is introduced. Note that it is not possible to statically infer the locality of `p` because the definition of `p` is different for each loop iteration.

```
check_assess_spec(village local *v, list l)
{
  while (l != NULL)
   { p = l->patient;
     p->time_left = p->time_left-1;
     label = v->label ;
     if (p->time_left == 0)
      { s = v->seed ;
        rand = my_rand(s);
        v->seed = rand * IM;
        label = v->label ;
        ...
        p->time_left = 10;
        t = p->time;
        p->time = t+10; }
     l = l->forward; }
}
```

(a) Localized version

```
check_assess_spec(village local *v, list l)
{
  while (l != NULL)
   { p = l->patient;
     if (IS_OWNER(p))
     {(patient local *) p->time_left =
        (patient local *) p->time_left -1;
      label = v->label ;
      if ((patient local *) p->time_left ==0)
       {s = v->seed;
        rand = my_rand(s);
        v->seed = rand * IM;
        label = v->label ;
        ...
        (patient local *) p->time_left = 10;
        t = (patient local *) p->time;
        (patient local *) p->time = t+10; }}
     else {
      p->time_left = p->time_left-1;
      label = v->label ;
      if (p->time_left == 0)
      {s = v->seed ;
       rand = my_rand(s);
       v->seed = rand * IM;
       label = v->label ;
       ...
       p->time_left = 10;
       t = p->time;
       p->time = t+10; } }
     l = l->forward; }
}
```

(c) Combined version

```
check_assess_spec(village *v, list l)
{
if (IS_OWNER(v)) {   // localized for v
 while (l != NULL)
  {p = l->patient;
   if (IS_OWNER(p))
   {(patient local *) p->time_left =
      (patient local *) p->time_left -1;
    label=(village local *) v->label;
    if ((patient local *) p->time_left ==0)
     {s =(village local *) v->seed;
      rand = my_rand(s);
      (village local *) v->seed =rand*IM;
      label=(village local *) v->label;...
      (patient local *) p->time_left =10;
      t =(patient local *) p->time;
      (patient local *) p->time =t+10;}}
   else {
    p->time_left = p->time_left-1;
    label = (village local *) v->label;
    if (p->time_left == 0)
     {s = (village local *) v->seed;
      rand = my_rand(s);
      (village local *) v->seed =rand*IM;
      label=(village local *) v->label;...
      p->time_left = 10;  t = p->time;
      p->time = t+10; }}
   l = l->forward; }}
else {    // v not local
 while (l != NULL)
  {p = l->patient;
   if (IS_OWNER(p))
   {(patient local *) p->time_left =
     (patient local *) p->time_left -1;
    label = v->label;
    if ((patient local *) p->time_left ==0)
     {s = v->seed;
      rand=my_rand(s); v->seed=rand*IM;
      label = v->label; ...
      (patient local *) p->time_left =10;
      t = (patient local *) p->time;
      (patient local *) p->time =t+10;}}
   else {
    p->time_left = p->time_left-1;
    label = v->label;
    if (p->time_left == 0)
     {s = v->seed;
      rand=my_rand(s);v->seed=rand*IM;
      label = v->label; ...
      p->time_left = 10;  t=p->time;
       p->time = t+10; }}
   l = l->forward; }}
}
```

(b) Coarse-grain version

Figure 3.11: Sample Code from Health Benchmark

Table 3.4 summarizes the results of three versions of the benchmark suite running on EARTH-MANNA with 1, 8 and 16 nodes. The first column provides the execution time of the *simple* version in seconds (the same data as in column 1 of Table 3.2). This provides us with a base case. Since the objective of this experiment was to compare the benefit of static techniques, dynamic techniques and a combined technique, the remaining columns list the percentage improvement over the *simple* version from static locality analysis, fine-grain and coarse-grain dynamic locality checks, and the combined version respectively.

Benchmark		Simple EARTH-C (sec)	Localized vs. Simple (%impr)	Fine-grain vs. Simple (%impr)	Coarse-grain vs. Simple (% impr)	Combined vs. Simple (%impr)
powerI	1 proc	66.95	3.99	0.12	2.57	3.99
	8 procs	9.07	4.19	0.15	2.76	4.19
	16 procs	4.72	4.45	0.21	2.97	4.45
powerII	1 proc	67.76	1.83	0.15	4.12	4.15
	8 procs	9.77	1.54	0.17	2.15	2.56
	16 procs	5.23	1.53	0.23	3.06	3.25
tsp	1 proc	26.95	1.96	-0.16	7.64	8.01
	8 procs	4.67	2.78	-1.68	4.07	7.49
	16 procs	3.77	3.45	-3.07	6.37	6.90
perimeter	1 proc	7.32	14.89	-0.65	5.83	26.2
	8 procs	1.12	16.96	-1.91	9.85	26.3
	16 procs	0.75	21.33	-2.26	8.64	29.2
health	1 proc	144.09	0.17	-3.27	0.08	0.19
	8 procs	42.74	14.72	-2.23	9.01	23.33
	16 procs	33.88	15.76	-1.75	4.93	21.78
voronoi	1 proc	7.27	14.86	-3.17	14.18	14.86
	8 procs	1.31	15.27	-3.47	3.06	15.27
	16 procs	1.04	16.35	-1.70	2.99	16.35
em3d	1 proc	29.15	21.78	-5.69	2.00	25.58
	8 procs	5.94	26.77	-3.70	3.74	29.76
	16 procs	3.48	24.14	-3.16	3.79	26.53

Table 3.4: Performance Improvement from Dynamic Locality Checks

Most of the results for the fine-grain dynamic checks (column 3) shows negative improvement over the *simple* version. This was to be expected since the overhead of the check outweighs the benefit of inserting the check in most cases. The overhead comes from the additional branch check overhead when data is actually a remote reference. Furthermore, these checks introduce extra branch tests that can complicate low-level instruction scheduling, and other scalar optimizations. Note that in most benchmarks the dynamic checks result in worse performance even for the one processor case, where all data is actually local. The failure of fine-grain checks serves to demonstrate that dynamic checks must be inserted with care in order to achieve performance improvement. The overhead of the checks must be carefully weighed with respect to potential benefit.

The results for coarse-grain checks (column 4), are more promising. In this case we carefully inserted dynamic checks at points where the memory reference is likely to be local, and the cost of the check can be amortized over several memory references. However, even with this careful placement of dynamic locality checks, we found that the gain from the *localized* version outperforms the *coarse-grain* version, for the same reason - dynamic check overhead. For example, in the `perimeter` benchmark, we gain 14–21% percentage improvement from *localized* version, while only 5–9% percentage improvement from *coarse-grain* version. Also, we should note that we are reporting the best result from many attempts at inserting dynamic checks. We found that it was quite difficult to find exactly those places that benefit from dynamic checks.

The results of the combined experiments (column 5) are the most encouraging, as it demonstrates the potential of the combination of static locality analysis and dynamic locality checks. We achieve better performance from the *combined* version than from either the *localized* version, or the *coarse-grain* version alone, particularly for the `tsp` , `perimeter` and `health` benchmarks. Just as the *fine-gain* version proves that too many dynamic locality checks will slow down the performance, the *coarse-grain* version shows that there is a limit of performance improvement for dynamic locality checks, due to the overhead caused by dynamic checks. Therefore the *combined* version achieves the best overall performance since the static locality analysis exploits some locality with no runtime cost, while dynamic locality checks further improve the performance, when properly inserted.

3.4.3 Summary of Experimental Results

In summary, we find that our static locality analysis does give significant improvements in all cases. For dynamic locality checks, it appears that fine-grain checks are seldom beneficial, and coarse-grain checks can give some benefit when carefully used. It was very interesting to note that the combination of static locality analysis and dynamic locality checks gave the best performance on all benchmarks. Thus, it seems like a good strategy to first use static analysis to handle all cases where memory references must be local, and then use well-placed dynamic checks to resolve any remaining references that are likely to be local.

In some cases locality analysis, even with dynamic checks, cannot compete with hand-coded locality mapping and declarations provided by the programmer. This is because the programmer may have some high-level idea of the layout of the data structure that he/she exploits to improve program performance (for example, the programmer may know that after a certain depth in a tree all sub-trees must be local). Thus, we cannot totally rely on compiler analyses and transformations, and it is important to retain the ability to explicitly declare local pointers. Since in many cases such declarations can be automated, and the locality analysis and specialization is a source-to-source transformation, one could imagine that the programmer could use the output of the compiler to produce a *localized* version of the program, and then test the program to see if acceptable speedup is achieved. If there is more locality in the program, then the programmer could add further locality declarations in order to further improve the program.

Finally, note that our experiments are performed on EARTH-MANNA. Compared with other distributed memory systems like the IBM SP2, or a network of workstations, EARTH-MANNA has a much smaller memory latency, which sometimes can be further hidden by multi-threading techniques. Therefore, we can expect better speedup from locality analysis for those machines with larger memory latencies and those that do not support multithreading techniques.

3.5 Related Work

Our intraprocedural locality propagation approach is similar to Steensgaard's [Ste96] linear points-to analysis algorithm. Other related work in the area of parallelizing programs with dynamic data structures is that of Carlisle [Car96] for distributed memory machines, and of Rinard and Diniz [RD96] for shared memory machines. In particular, Carlisle's [Car96] affinity analysis has similar goals as our locality analysis, albeit we target different kinds of locality.

Steensgaard proposed a type-inference based algorithm for points-to analysis with almost linear time complexity. His algorithm is both flow-insensitive and context-insensitive. On encountering a call-site he simply merges the formal parameters with their respective actual arguments. Thus his algorithm cannot distinguish between information arriving at a function from different calling contexts.

For our locality analysis, the calling context information is crucial. The invocation specification (`@HOME, @OWNER_OF`) of the call site is a major source of locality information. Thus we do not want to merge locality information arriving from different calling contexts. On the contrary, we want to create specialized versions of a function for each calling context that provides us substantial locality. To this end, we use type-inference to only propagate information intraprocedurally, and employ a different technique for interprocedural propagation as explained in Section 3. However, to ensure that our intraprocedural propagation collects conservatively safe information, we need to make conservative assumptions at the start of the procedure and on encountering call-sites (using read-write set information). Thus, although our approach was originally inspired by the points-to analysis, it is really specifically tailored to capture the information most relevant to locality analysis.

Carlisle's affinity analysis is designed to exploit the locality with respect to linked fields. His analysis relies upon the information regarding the probability (affinity) that nodes accessible by traversing a linked field are residing on the same processor. If the affinity is high, then he puts runtime checks to eliminate remote accesses. However, the affinity information is not infered by the compiler, but is provided via programmer annotations.

Our locality analysis is not designed to exploit the locality achievable via linked fields (as discussed in Section 3.4) because we wanted our analysis to be an automatic compiler analysis, and we did not want to burden the user for additional detailed information. Locality of recursive fields can be explicitly declared using local pointer declarations in EARTH-C.

3.6 Summary

In this chapter we have presented a static locality analysis for parallel C programs, based on type-inference techniques. These techniques use both locality information provided by the programmer and other program characteristic like malloc sites to infer additional locality and to automatically specialize functions. We also introduced two potential dynamic techniques (fine-grain and coarse-grain), where dynamic locality checks are introduced into the program.

We evaluated the effectiveness of the static locality analysis for a set of benchmarks and found that it can eliminate significant number of pseudo-remote accesses and provide speedups ranging from a modest 1.5% up to 26% over the original parallel program. For several benchmarks, this speedup also comes very close to the speedup obtained by an advanced hand-coded version. Furthermore, we found that the locality analysis reduced the burden on the programmer, and allowed us to develop shorter, more general benchmarks.

In order to compare the static techniques with dynamic techniques, we also provided experimental results comparing the static locality analysis, fine-grain dynamic checks, coarse-grain dynamic checks, and a combined approach. We found that the fine-grain dynamic checks rarely provided any benefit, and often produced a degradation in performance due to too much overhead. The coarse-grain dynamic checks did give some benefit, but not as much as the static locality analysis. Further, it was quite difficult to find the best places to insert dynamic checks. The best overall technique was a combination of the static analysis and coarse-grain dynamic checks. This demonstrates that one should first use static techniques that introduce no run-time overhead, and then use dynamic techniques on the remaining interesting parts of the program.

Chapter 4

Communication Optimizations

In the previous chapter, we studied the static and dynamic locality techniques, which detect when a pointer reference can be safely assumed to be a local memory reference, thus to be replaced with an inexpensive local memory reference, in order to minimize the remote communication overhead. However, there remain many situations where remote accesses cannot be made local. In some cases the programmer implements a naive algorithm that does not fully exploit locality, and in other cases applications have inherent irregularities that require a significant number of remote memory accesses.

The focus of this chapter is on how to optimize the placement and type of remote communication. Our framework includes: (1) code movement to issue remote reads earlier and writes later, (2) code transformations to replace repeated/redundant remote accesses with one access, and (3) transformations to block or pipeline a group of remote requests together. Our framework consists of an analysis phase that determines where it is safe to move communication, and a transformation phase that selects the best location and type of communication primitives.

The remainder of the chapter is organized as follows. Section 4.1 provides some small examples to motivate the communication optimizations, and Section 4.2 presents the analysis itself. In Section 4.3 we provide experimental results, and illustrative examples, for our set of benchmark programs. Finally, in Section 4.4 we discuss some related work, and in Section 4.5 we give conclusions.

4.1 Motivating Examples

In this section we present two small program examples that motivate the types of communication optimizations that we consider. These examples also illustrate the various tradeoffs that need to be considered when applying the communication optimizations.

Let us first consider the simple function `distance` given in Figure 4.1(a). This function takes a pointer to a structure `Point`, and returns the distance of the point from the origin. As no special locality information is given for the parameter `p`, the compiler must assume that each indirect reference via `p` is potentially a remote operation. Each program example shows remote operations underlined, and in Figure 4.1(a), it is clear that there are four remote reads via `p`. In order to apply the communication optimizations, each program is first simplified so that each statement has at most one remote read or write. Figure 4.1(b) shows the simplified version of `distance`.

One goal of communication optimization is to move the reads earlier in the program. For example, the four remote reads in Figure 4.1(b) can be moved to the beginning of the function. Moving remote reads earlier has several advantages. First, because the remote operations are split-phase, by issuing the remote read as early as possible, one allows communication to overlap with the computation following the read. Secondly, moving remote reads can also expose some opportunities for discovering redundant communication. Figure 4.1(c) shows that two remote reads are all that are necessary. Finally, by moving remote reads together, communication may be pipelined or blocked. Figure 4.1(d) shows one such transformation for blocking. In this case the entire structure is moved to a local struct (`bcomm1`) by one remote operation (`blkmov`), and then local reads are made with respect to the local struct. The choice of whether to use blocked operations depends on two factors: (1) the relative cost of pipelined scalar reads vs. block reads, and (2) the extra overhead of reading spurious fields of a struct (i.e. the fields required may not be contiguous in memory).

Figure 4.2 illustrates an example where both remote reads and remote writes are optimized. Note that the two remote reads are moved earlier in the program, whereas the two remote writes are moved later in the program. In the case of remote reads, moving the operations earlier in the program has two advantages. First, it allows overlapping communication with computation, and further it exposes opportunities for pipelined and blocked communication. Thus, it is always a good idea to move remote reads earlier. However, in the case of remote writes there are two conflicting goals. Moving remote writes earlier may improve the overlap of communication and

```
double distance(Point *p)
{ double dist_p;
  dist_p = sqrt(( p->x  *  p->x ) +
                ( p->y  *  p->y ));
  return(dist_p);  }
```

(a) original C code

```
double distance(Point *p)
{ double dist_p, temp1, temp2, ..., temp7;
  temp1 =  p->x ; temp2 =  p->x ;
  temp3 = temp1 * temp2;
  temp4 =  p->y ; temp5 =  p->y ;
  temp6 = temp4 * temp5;
  temp7 = temp3 + temp6;
  dist_p = sqrt(temp7);
  return(dist_p);  }
```

(b) SIMPLE C code

```
double distance(Point *p)
{ double dist_p, comm1, comm2, temp3,..., temp7;
  comm1 =  p->x ;   comm2 =  p->y ;
  temp3 = comm1 * comm1;
  temp6 = comm2 * comm2;
  temp7 = temp3 + temp6;
  dist_p = sqrt(temp7);
  return(dist_p);
}
```

(c) Collecting and Moving Reads

```
double distance(Point *p)
{ double dist_p, temp3, temp6, temp7;
  Point bcomm1;
  blkmov(p,&bcomm1,sizeof(Point)) ;
  temp3 = bcomm1.x * bcomm1.x;
  temp6 = bcomm1.y * bcomm1.y;
  temp7 = temp3 + temp6;
  dist_p = sqrt(temp7);
  return(dist_p);
}
```

(d) Blocking Reads

Figure 4.1: Optimizing Remote Reads

computation, but moving remote writes later may expose opportunities for blocking. Figure 4.2(c) shows the remote writes moved later, and this is a good idea if the blocking transformation in Figure 4.2(d) is performed. However, it is a bad idea if blocking should not be performed. Thus, we can see that there is a fine balance between the placement of remote write operations and the blocking transformations.

The two examples presented in this section show two simple applications of the communication optimizations. In general, one needs more sophisticated analyses that determine when it is safe to move communication, where to place the communication, and where to apply blocking.

4.2 The Algorithm

In this section, we describe the algorithm for communication optimization. It consists of an analysis phase followed by a transformation phase. The analysis phase, called *possible-placement analysis*, collects the set of remote reads and writes that can possibly be placed at each program point. The transformation phase, called *communication selection*, picks "the best" location for each remote read/write and performs the appropriate transformation, applying blocking when applicable.

The algorithm operates at the SIMPLE level. As described in Section 2.1, SIMPLE programs are composed of basic statements (assignments and function calls), statement sequences, conditionals (`if` and `switch`), and loops (`while`, `do`, and `for`). Each basic statement can have at most one remote operation (remote read or remote write). There is no irregular flow of control since SIMPLE programs have been automatically structured using goto-elimination [EH94]. Thus, we give our analysis rules in a structured form.

The algorithm makes use of advanced side-effect analysis. Each basic and compound statement is decorated with the set of locations read/written. In addition, statements involving pointers to the heap are decorated with heap read/write sets that have been computed using connection analysis [Ghi97, GH98].

4.2.1 Possible-Placement Analysis

Possible-placement analysis collects sets of remote communication expressions. Each *remote communication expression* is a 4-tuple $(p, f, n, Dlist)$ where p is a pointer variable, f is the field of a struct, n is an estimated frequency, and $Dlist$ is a set of basic

```
int scale_point(Point *p, double k)
{ p->x = scale( p->x ,k);
  p->y = scale( p->y ,k);
}
```

(a) original C code

```
int scale_point(Point *p, double k)
{ double temp1, temp2, temp3, temp4;
  temp1 = p->x ;
  temp2 = scale(temp1,k);
  p->x = temp2;
  temp3 = p->y ;
  temp4 = scale(temp3,k);
  p->y = temp4;
}
```

(b) SIMPLE C code

```
int scale_point(Point *p, double k)
{ double temp2, temp4;
  double comm1, comm2;
  comm1 = p->x ;
  comm2 = p->y ;
  temp2 = scale(comm1,k);
  temp4 = scale(comm2,k);
  p->x = temp2;
  p->y = temp4;
}
```

(c) Collecting and Moving Reads/Writes

```
int scale_point(Point *p, double k)
{ double temp2, temp4;
  Point bcomm1;
  blkmov(p,&bcomm1,sizeof(Point)) ;
  bcomm1.x = scale(bcomm1.x,k);
  bcomm1.y = scale(bcomm1.y,k);
  blkmov(&bcomm1,p,sizeof(Point)) ;
}
```

(d) Blocking Reads/Writes

Figure 4.2: Optimizing Remote Reads/Writes

statement labels. For convenience we sometimes write these tuples combining the first two components, so that $(p, f, n, Dlist)$ is sometimes written as $(p{\rightarrow}f, n, Dlist)$. A tuple can represent a read or a write.

The objective of possible-placement analysis is to compute the set of read/write tuples that could be *safely placed* at each program point.

Read Tuples

We say that a *read tuple* of the form $(p{\rightarrow}f, n, Dlist)$ can be *safely placed* just before statement S if:

1. $Dlist = [d_1, \ldots, d_n]$ is a list of statement labels for statements reading $p{\rightarrow}f$, i.e. statements of the form, $d_i : lhs = p{\rightarrow}f$;
2. statement S dominates all statements $d_i \in Dlist$; and
3. the value of $p{\rightarrow}f$ is the same just before statement S as it is at all statements $d_i \in Dlist$.

Consider the example of a safely placed read tuple T in Figure 4.3(a). Tuple T indicates that the read of `p->f` can be safely placed just before statement S, and this value is used by the three statements, `d1`, `d2` and `d3`. Note that tuple T satisfies all three conditions to make it safely placed: (1) `d1`, `d2` and `d3` are all reading `p->f`; (2) statement `S` dominates all of `d1`, `d2` and `d3`; and (3) the value of `p->f` is the same at statement `S` as it is at `d1`, `d2`, and `d3`. Figure 4.3(b) shows a typical transformation that can be done using this tuple. In this case one remote read of `p->f` is performed just before statement S, and then three local reads are used at `d1`, `d2` and `d3`.

Write Tuples

We say that a *write tuple* of the form $(p{\rightarrow}f, n, Dlist)$ can be *safely placed* just after statement S if:

1. $Dlist = [d_1, \ldots, d_n]$ is a list of statement labels for statements writing $p{\rightarrow}f$, i.e. statements of the form, $d_i : p{\rightarrow}f = rhs$;
2. statement S postdominates all statements $d_i \in Dlist$;

```
{  /* T = (p->f,2,[d1,d2,d3]) */
  S:
     if ( ... )
       { ...
 d1:     x = p->f;
       }
     else
       { ...
 d2:     y = p->f;
       }
     ...
 d3: z = p->f;
     ...
}
```

(a) T is safely placed before S

```
{      local_temp = p->f;
  S:
     if ( ... )
       { ...
 d1:     x = local_temp;
       }
     else
       { ...
 d2:     y = local_temp;
       }
     ...
 d3: z = local_temp;
     ...
}
```

(b) example transformation

Figure 4.3: Example of a Safely Placed Read Tuple

3. the value of $p{\rightarrow}f$ is the same just after statement S as it is at all statements $d_i \in Dlist$; and
4. there is a definition of $p{\rightarrow}f$ on all paths from the beginning of the start node to statement S.

Consider the example of a safely placed write tuple T in Figure 4.4(a). Tuple T indicates that the write of `p->f` can be safely placed just after statement S, and this value was written by two statements, `d1` and `d2`. Note that tuple T satisfies all four conditions to make it safely placed: (1) `d1` and `d2` are both writing `p->f`; (2) statement `S` postdominates both `d1` and `d2`; (3) the value of `p->f` is the same at statement `S` as it is at `d1` and `d2`; and (4) `p->f` is defined on all paths to `S`.

Note that a tuple for `p->g` could not be safely placed at statement `S` because `p->g` is not defined on the `else` branch of the conditional statement, and thus it does not meet the fourth criteria. Note that the safe placement of writes has this fourth criteria, whereas the safe placement of reads does not. Intuitively this is because it is safe to make spurious reads that are never used, but it is not safe to make spurious writes.

Figure 4.4(b) shows a typical transformation that can be done using this tuple. In this case one remote write of `p->f` is performed just after statement S, and then two local writes are used at d1 and d2. Note that even though this transformation is safe, it will not necessarily be performed because it is delaying the write without removing any redundancy. In the case of remote reads it is always a good idea to move them earlier, even if there is no redundancy. However, in the case of remote writes it is only useful to delay the write if some redundancy is found, or if a blocked write can be used instead of several related single writes.

```
{     if ( ... )
        { ...
  d1:     p->f = x;
          p->g = y;
        }
      else
        { ...
  d2:     p->f = z;
        }
      ...
   S:
      ...
   /* T = (p->f,1,[d1,d2]) */
}
```

(a) T is safely-placed after S

```
{     if ( ... )
        { ...
  d1:     local_temp = x;
          p->g = y;
        }
      else
        { ...
  d2:     local_temp = z;
        }
      ...
   S:
      ...
      p->f = local_temp;
}
```

(b) example transformation

Figure 4.4: Example of a Safely Placed Write Tuple

Computing RemoteReads and RemoteWrites

Possible-placement analysis computes two sets, RemoteReads and RemoteWrites, for each program point, defined as follows:

RemoteReads(S): The set of read tuples that may be safely placed just before statement S.

RemoteWrites(S): The set of write tuples that may be safely placed just after statement S.

These sets are collected using structured analyses. RemoteReads are computed via a backwards analysis, whereas RemoteWrites are computed via a forwards analysis. Each analysis is completed with a single traversal of the structured SIMPLE representation, no iteration is required. Thus, the analyses are quite efficient. The structured analysis rules are given in Figures 4.5 and 4.6.

In Figures 4.5 and 4.6, an auxiliary set is used for propagation. It contains all the tuples that can be safely propagated from the statement. These tuples may or may not be included in the remote set of the statement. For remote reads, the propagation set and the remote set are the same, reflecting our earliest placement policy for remote reads. However, only some tuples from the propagation set are added to the remote write set of a statement, following our policy of delaying remote writes only for blocking or redundancy elimination. We will explain more later.

The first rule in Figure 4.5 shows the driving rule, collectCommSet, that selects the appropriate rule depending on the kind of the statement. The simplest rule, for basic statements, is collectCommSetBasic. A basic remote statement of the form $S_i : lhs = p \rightarrow f$ generates a RemoteRead tuple $(p \rightarrow f, 1, Si)$. A statement of the form $S_i : p \rightarrow f = rhs$ generates a RemoteWrite tuple of the form $(p \rightarrow f, 1, Si)$.

The last two rules in Figure 4.5, collectCommReadsSeq and collectCommWritesSeq, demonstrate how the tuples are propagated through statement sequences. In Figure 4.5, `commReadSet` and `commWriteSet` are the `RemoteReads` and `RemoteWrites` for each statement, while `currReadSet`, `predReadSet`, `currWriteSet` and `predWriteSet` are the corresponding propagation sets. Let us first focus on collectComReadsSeq, the rule for propagating RemoteReads earlier in the program. In this case RemoteRead tuples are propagated from S_n to S_1. At each step i, the remote read set commReadSet(S_i) and the propagation set currReadSet have already been calculated for the program point just before S_i. Statement S_{i-1} is analyzed to find the sets that it generates predReadSet and commReadSet(S_{i-1}), and then all tuples from currReadSet that are not killed by S_{i-1} are added to predReadSet and commReadSet(S_{i-1}), giving the propagation set valid at the program point just before S_{i-1}.

Determining the kill set is actually the tricky part of the algorithm and requires relatively detailed side-effect analysis. A *read tuple* of the form $(p \rightarrow f, n, Dlist)$ is propagated from statement S_i to statement S_{i-1}, if at statement S_{i-1}:

1. the base variable p is not killed; and
2. the reference $p \rightarrow f$ is not written **through an alias**;

It is obvious that if S_{i-1} writes to pointer p (either directly or via an alias), then p is no longer pointing to the same structure, and all tuples with p as the first item must be killed. However, one must also kill all tuples of the form $(p \rightarrow f, *, *)$ if S_{i-1}

```
/* computation of communication read/write sets for a statement */
fun collectCommSet(stmt, accessType) =
  case (type(stmt)) of
    <Basic> => return (collectCommSetBasic(stmt, accessType));
    <Seq>    => if (accessType == READ) return(collectCommReadsSeq(stmt));
                  else return(collectCommWritesSeq(stmt)); /* must be WRITE */
    <Loop> => return(collectCommSetLoop(stmt, accessType));
    <If>     => return(collectCommSetIf(stmt, accessType));
     ...              /* Other stmts, switch...*/

fun collectCommSetBasic(stmt, accessType) =
  if ( (accessType == READ) && isRemoteAccess(rhs(stmt)))
         commReadSet(stmt) = (<basevar(rhs(stmt)),field(rhs(stmt)),1,label(stmt)>);
         return commReadSet(stmt);
  else if ((accessType == WRITE) && isRemoteAcess(lhs(stmt)))
         commWriteSet(stmt) = (<basevar(lhs(stmt)),field(lhs(stmt)),1,label(stmt)>);
         return commWriteSet(stmt);
  return EMPTY;

fun collectCommReadsSeq(stmtSeq) =
  [s_n:s_1] = getStmts(stmtSeq);   currReadSet = collectCommSet(s_n, READ);
  foreach stmt s_i in [s_n:s_2] /* backward propagation from s_n to s_1 */
    predReadSet = collectCommSet(s_i−1, READ); /* predecessor */
    foreach commExpr (p,f,n,d) in currReadSet
      if (varWritten(p, s_i−1)) /* base variable p itself is written */
        continue; /* commExpr cannot be propagated above s_i−1 */
      if (accessedViaAlias(p, f, d, s_i−1, WRITE))
        continue; /* p−>f possibly accessed via an alias, say t−>f: can't be propagated */
      addToSets((p,f,n,d),commReadSet(s_i−1),predReadSet);/*save and propagate */
      currReadSet = predReadSet;
  return(currReadSet);

fun collectCommWritesSeq(stmtSeq) =
  [s_1:s_n] = getStmts(stmtSeq);   currWriteSet = collectCommSet(s_1, WRITE);
  foreach stmt s_i in [s_1:s_n−1] /* forward propagation from s_1 to s_n */
    succWriteSet = collectCommSet(s_i+1, WRITE);  /* successor */
    foreach commExpr (p,f,n,d) in currWriteSet
      if (varWritten(p, s_i+1)) /* base variable p itself is written */
        continue; /* commExpr cannot be propagated above s_i+1 */
      if (accessedViaAlias(p,f,d,s_i+1,READ) || accessedViaAlias(p,f,d,s_i+1,WRITE))
        continue; /* p−>f possibly written via an alias, say t−>f: can't be propagated */
      addToSet((p,f,n,d), succWriteSet); /* propagate further on */
      if (contain((p,f1,n1,d1), commWriteSet(s_i+1)))
        addToSet((p,f,n,d), commWriteSet(s_i+1)); /* add to remote write set */
      currWriteSet = succWriteSet;
  return(currWriteSet);
```


Figure 4.5: Possible Placement Analysis - Basic Rules

```
/* computation of communication read/write sets for an if stmt */
fun CollectCommSetIf(ifStmt, accessType) =
  ifSet = commIfSet(ifStmt, accessType) = EMPTY;
  thenSet = collectCommSet(ifStmt.thenpart, accessType);
  elseSet = collectCommSet(ifStmt.elsepart, accessType);
  if (accessType == READ)
   foreach commExpr (p,f,n,d) in thenSet /* merge then set */
     n = adjustFrequency(n, ifStmt); addToSets((p,f,n,d), ifSet, commReadSet(ifStmt));
   foreach commExpr (p,f,n,d) in elseSet  /* merge else set */
     n = adjustFrequency(n, ifStmt); addToSets((p,f,n,d), ifSet, commReadSet(ifStmt));
  else /* if (accessType == WRITE) */
   foreach commExpr (p,f,n,d) in thenSet
     if ((p,f,n1,d1) exists in elseSet)/* Same expression is written in else part */
       n = adjustFrequency(n, ifStmt);    addToSet((p,f,n,d), ifSet);
       n1 = adjustFrequency(n1, ifStmt);  addToSet((p,f,n1,d1), ifSet);
   return (ifSet); /* commExprs that can be moved above or below the if stmt */

/* computation of communication read/write sets for a loop */
fun collectCommSetLoop(loopStmt, accessType) =
  loopSet = commLoopSet(loopStmt, accessType) = EMPTY;
  bodySet = collectCommSet(loopStmt.body, accessType);
  if (accessType == READ)
   foreach commExpr (p,f,n,d) in bodySet
    if (varWritten(p, loopStmt) || accessedViaAlias(p, f, d, loopStmt, WRITE))
      continue; /* cannot move it above the loop */
    n=adjustFrequency(n,loopStmt);addToSets((p,f,n,d),loopSet,commReadSet(loopStmt));
  else /* if (accessType == WRITE) */
    if (executesOnce(loopStmt))
     foreach commExpr (p,f,n,d) in bodySet
       if ( varWritten(p, loopStmt) || accessedViaAlias(p, f, d, loopStmt, READ)
           || accessedViaAlias(p, f, d, loopStmt, WRITE))
         continue; /* cannot move it below the loop */
       n = adjustFrequency(n, loopStmt);
       addToSets((p,f,n,d), loopSet, commWriteSet(loopStmt));
   return(loopSet); /* commExprs that can be moved above/below the loop */

fun adjustFrequency(freq, stmt) =
  if (isLoopStmt(stmt)) return (freq * 10); /* moving commExpr out of a loop */
  if (isIfStmt(stmt)) return (freq / 2);/* moving commExpr out of an if statement */
  if (isSwitchStmt(stmt))
    n = numberOfCaseStmts(stmt); /* number of case stmts involved */
    return (freq / n); /* moving commExpr out of a switch statement */
  return (freq); /* return original value */
```

Figure 4.6: Possible Placement Analysis - Compound Rules

writes to $p{\rightarrow}f$ **via an alias**. Note that a direct write via $p{\rightarrow}f$ should not be killed because in the best case we want to replace all reads and writes via pointer p with accesses to a local structure.

Fig 4.7 shows an example, where each statement is decorated with its RemoteReads. Observe that the tuple $(p{\rightarrow}y, 1, S4)$ is killed when being propagated from $S2$ to $S1$ for the violation of condition 1. Under the assumption that p and t are aliased, $p{\rightarrow}x$ may be written through $t{\rightarrow}x$ at statement $S2$, therefore the tuple $(p{\rightarrow}x, 1, S3)$ is also killed when being propagated from $S3$ to $S2$, according to condition 2. We use our connection analysis with anchor handles to distinguish between direct reads and aliased reads [GH98].

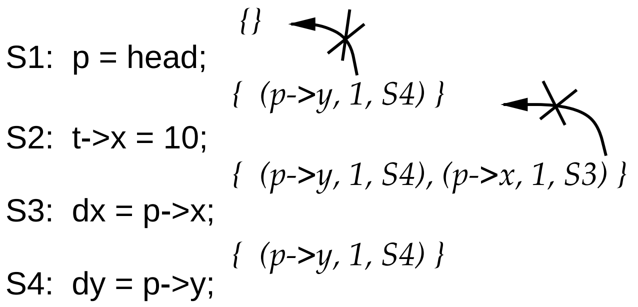


Figure 4.7: Assuming p and t Are Aliased

The rule collectCommWritesSeq for propagating RemoteWrites forwards is similar to collectCommReadsSeq. In this case, the statement sequence is processed from S_1 to S_n. A *write tuple* of the form $(p{\rightarrow}f, n, Dlist)$ is propagated from statement S_i to statement S_{i+1}, if at statment S_{i+1}:

1. the base variable p is not killed; and
2. the reference $p{\rightarrow}f$ is not **read** or written though an alias;

The crucial difference between the read and write propagations is that tuples of the form $(p{\rightarrow}f, *, *)$ must be killed if S_{i+1} reads or writes $p{\rightarrow}f$ via an alias. The other difference is that, the tuples from the propagation set are added to the commWriteSet of S_{i+1}, only if S_{i+1} already contains some remote write accesses of p. Intuitively, we will delay remote writes to S_{i+1}, only if there appears some opportunity for either block writing or redundancy elimination. In other words, we will not trade pipelining remote writes for the possibility of overlapping remotes with the computation.

Fig 4.8 shows an example, with the RemoteWrites displayed at each statement. We notice that $(p{\rightarrow}x, 1, S1)$ is added to the RemoteWrites of S_2 for the potential of block writing p at S_2, as the RemoteWrites of S_2 contains another tuple of p $(p{\rightarrow}y, 1, S2)$. From S_2 to S_3, $(p{\rightarrow}x, 1, S1)$ is killed on account of a possible alias read of $p{\rightarrow}x$ via $t{\rightarrow}x$. Although $(p{\rightarrow}y, 1, S2)$ is valid at $S3$, we do not add it to the RemoteWrites of $S3$, as $S3$ does not access any tuples of p. However, the tuple $(p{\rightarrow}y, 1, S2)$ can be further propagated and deposited at S_4. Finally, at S_5, all tuples of p are killed, as a result of the update of the variable p.

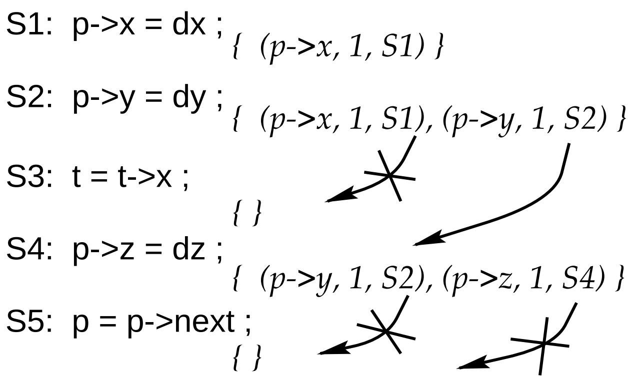


Figure 4.8: Assuming p and t Are Aliased

Figure 4.6 gives the rules for handling loops and conditionals. In the case of `if` and `switch` statements each alternative is analyzed to give the propagating set and the remote set generated by each alternative. For the case of RemoteReads analysis, all tuples from the propagation sets of all alternatives are propagated out of the conditionals. This is because we are optimistically propagating reads as early as possible, since it is safe to read spurious fields, and never use their value.[1] However, we must be more conservative since it is certainly not safe to propagate spurious writes. Thus, for RemoteWrites we only propagate tuples that occur in all alternatives. Notice that these tuples are not added to the RemoteWrites of the conditionals for being conservative. To reflect the conditional nature of the control flow, we adjust the frequencies in the tuples. Our simple scheme simply divides the current frequency by the number of alternatives, although clearly either static or dynamic branch prediction information would be very valuable here. When moving tuples out of conditionals, tuples referring to the same location are merged by summing their adjusted frequencies and taking

[1]By moving tuples out of conditionals and loops we may move a pointer dereference to a spot in the program that will cause a dereference that would not have occurred in the original program. In the transformation stage we ensure that it is safe to issue such a dereference.

the union of their definition sets.

We handle loop statements by analyzing the loop body first, collecting and propagating the remote sets to the top or bottom of the loop body. We then propagate all tuples that can not be killed by the loop out of the loop. At the top of a loop body, a valid RemoteRead tuple of the form $(p{\rightarrow}f, n, Dlist)$ is promoted ahead if p is not modified by the loop, and $p{\rightarrow}f$ is not written via an alias in the loop. Similarly, at the bottom of a loop, a valid RemoteWrite tuple $(p{\rightarrow}f, n, Dlist)$ is pulled after the loop, if p is not updated by the loop statement, and $p{\rightarrow}f$ is neither read nor written via an alias in the loop. The decision is made based on the Read/Write information associated with the loop statement, thus avoiding many iterations through the loop body. All tuples that are safely propagated out of the loop are added to the RemoteReads or RemoteWrites of the loop statement, for the potential benefit of redundancy elimination. Their frequencies are increased, corresponding to the expected number of times the loop will execute. In the case of an unknown loop count, they can be adjusted according to the profile information. In our implementation, we adopt a simple strategy of multiplying by 10. This simple strategy works well for our benchmark set.

To demonstrate the backward propagation of RemoteReads, consider the example program in Figure 4.9. This program traverses a list pointed to by `p`, looking for a point that is within `epsilon` distance to the point pointed to by `t`. The last such point is pointed to by `close`. After the loop the differences between the `x` and `y` fields of `t` and `close` are computed.

The backward propagation proceeds as follows. First, consider the outer statement sequence $S1; ...; S8$. The analysis starts with $S8$ which has no remote accesses, and therefore generates the empty set. Statement $S7$ generates $(t{\rightarrow}y, 1, S7)$, statement $S6$ generates $(close{\rightarrow}y, 1, S6)$, and so on until statement $S3$ where the set $\{(t{\rightarrow}y, 1, S7)$, $(close{\rightarrow}y, 1, S6)$, $(t{\rightarrow}x, 1, S4)$, $(close{\rightarrow}, 1, S3)\}$ has been calculated.

Following the same sequence propagation rules for the loop body (statements $S9 \ldots S15$), we obtain the RemoteReads $\{(p{\rightarrow}next, 1, S15), (t{\rightarrow}y, 1, S12), (t{\rightarrow}x, 1, S11),$ $(p{\rightarrow}y, 1, S10)$, $(p{\rightarrow}x, 1, S9)\}$ at $S9$. Now we consider propagating them out of the loop. Since p is written in the loop body, all tuples with p must be killed, and thus only $\{(t{\rightarrow}y, 1, S12), (t{\rightarrow}x, 1, S11)\}$ are generated by the loop. The loop also writes the pointer $close$, and so it kills the tuples $(close{\rightarrow}y, 1, S6)$ and $(close{\rightarrow}x, 1, S3)$ that were valid at $S3$. Combining the remaining tuples at $S3$, $\{(t{\rightarrow}x, 1, S4), (t{\rightarrow}y, 1, S7)\}$, with the frequency-adjusted tuples generated by the loop $\{(t{\rightarrow}x, 10, S11), (t{\rightarrow}y, 10, S12)\}$, gives the set $\{(t{\rightarrow}x, 11, S11 : S4), (t{\rightarrow}y, 11, S12 : S7)\}$ at $S2$. This set propagates to $S1$ as well.

```
S1: p = head; {(t→x, 11, S11 : S4), (t→y, 11, S12 : S7)}
S2: while (p!=NULL) {(t→x, 11, S11 : S4), (t→y, 11, S12 : S7)}
    S9:  { ax = p->x ;
              {(p→next, 1, S15), (t→y, 1, S12), (t→x, 1, S11), (p→y, 1, S10), (p→x, 1, S9)}
    S10:   ay = p->y ; {(p→next, 1, S15), (t→y, 1, S12), (t→x, 1, S11), (p→y, 1, S10)}
    S11:   bx = t->x ; {(p→next, 1, S15), (t→y, 1, S12), (t→x, 1, S11)}
    S12:   by = t->y ; {(p→next, 1, S15), (t→y, 1, S12)}
    S13:   dist = f(ax,ay,bx,by); {(p→next, 1, S15)}
    S14:   if (dist < epsilon) close = p; {(p→next, 1, S15)}
    S15:   p = p->next ; {(p→next, 1, S15)}
         }
S3: cx = close->x ;   {(t→y, 1, S7), (close→y, 1, S6), (t→x, 1, S4), (close→x, 1, S3)}
S4: tx = t->x ;       {(t→y, 1, S7), (close→y, 1, S6), (t→x, 1, S4)}
S5: diffx = cx - tx;  {(t→y, 1, S7), (close→y, 1, S6)}
S6: cy = close->y ;   {(t→y, 1, S7), (close→y, 1, S6)}
S7: ty = t->y ;       {(t→y, 1, S7)}
S8: diffy = cy - ty;  {}
```

Figure 4.9: Example Propagation of RemoteRead Sets

Fig 4.10 gives an example for propagating remote write tuples forwards. The example generates a random point within a circle, and records the closest x and y axis distance to the center of the circle. In the example, each statement is decorated with its RemoteWrites.

The propagation of RemoteWrites starts at $S1$. As $S1$ is a compound statement, we proceed further to its body(statements $S4 ... S10$). A tuple $(new{\rightarrow}x, 1, S4)$ is generated at $S4$, then is propagated and merged into the RemoteWrites of $S5$. The RemoteWrites of $S5$ also contains $(new{\rightarrow}y, 1, S5)$ it generates. Both tuples are then propagated all the way out of the loop, due to the fact that the loop does not update new, and there is no alias read or written for these tuples in the loop. However, these tuples do not appear in any RemoteWrites of statements $S6 ... S10$, as none of these statements contains a write access to new, (there are no opportunities for block writing or redundancy elimination). As a result, the RemoteWrites of these statements remain empty. Nevertheless, these tuples $(new{\rightarrow}x, 10, S4)$, and $(new{\rightarrow}y, 10, S5)$ are added to the RemoteWrites of $S1$, for the potential of loop redundancy elimination. Their frequencies have been increased up to 10 to reflect the propagation from the loop. From $S1$, these tuples continue to be propagated until the end. They are not added to any RemoteWrites of statements $S2$ and $S3$ for the same reason. Thus, the RemoteWrites of these statements remain empty.

```
S1: do {
    S4:  new->x = getRandom(range); {(new→x, 1, S4)}
    S5:  new->y = getRandom(range); {(new→x, 1, S4), (new→y, 1, S5)}
    S6:  ax = orig->x; {}
    S7:  Dx = new->x - ax; {}
    S8:  ay = orig->y; {}
    S9:  Dy = new->y - ay; {}
    S10: dist = Dx * Dx + Dy * Dy;  {}
       } while (dist > radius*radius );  {(new→x, 10, S4), (new→y, 10, S5)}
S2: if (minx > Dx)  {}
    S11:  minx = ax;
S3: if (mixy > Dy)  {}
    S12:  miny = ay;
```

Figure 4.10: Example Propagation of RemoteWrite Sets

4.2.2 Communication Selection

After possible-placement analysis, each statement is associated with a set of remote communication expressions (RCEs), which can be safely placed before/after the statement. Based on this information we have developed some heuristics for placing the communications, and for selecting the correct kind of communications.

In compiling with split-phase remote memory operations it is usually beneficial to place these operations as early as possible. This allows some overlap of communication with computation. Thus, we follow an earliest placement policy for placing remote reads. This is achieved via a top-down traversal of the SIMPLE representation. At the beginning of the traversal a hash table is generated that is used to contain all remote memory operations that have already been selected. Each remote memory operation is a triple (p, f, d), also written $(p{\rightarrow}f, d)$, where p is a pointer, f is a field, and d is the label of the statement containing the memory reference. Initially the hash table is empty. At each statement, S_i, the RemoteReads set is examined, and any tuples already in the hash table are removed from the RemoteReads set. For each remaining access of the form $(p{\rightarrow}f, n, Dlist)$, a decision is made whether or not to to insert the remote read at this point. If the frequency n is one or more and it is safe to place a dereference to `p` at this point in the program, then this is a candidate for inclusion.[2] If the tuple is a candidate, then an entry for each statement label in

[2]There are several ways of ensuring that the dereference is valid. One method is to check that there exists some dereference to `p` on all program paths starting at S_i. Another method is to use a *nilness* analysis to determine which pointers are definitely not nil. In our run-time system we also

Dlist is made into the hash table.

After all the candidates at a program point are selected we then determine if some of them should be blocked. This choice depends upon the target architecture. For example, for our target architecture, pipelining is better for two remote accesses, but blocked communication is better for three or more accesses. If the structure being read is very large compared to the number of fields actually required, then the tradeoff shifts slightly towards pipelined communication. We use a cost model based on the number of words needed, and the size of the block to determine the best choice. Note that if a pointer p is only used for reads, then a block move is safe, even if we read spurious fields.

Consider the example in Figure 4.11(a) (the same program as used in Figure 4.9 to illustrate possible-placement analysis). In placing the communication, the program is processed starting at statement $S1$. At this point there are two entries for pointer t, each with a frequency count of 11, and it is safe to dereference t at this point. Thus, both of these tuples are selected as candidates. Since there are only two reads, they are pipelined (as shown in Figure 4.11(b)). Entries for $(t \rightarrow x, S11)$, $(t \rightarrow x, S4)$, $(t \rightarrow y, S12)$, and $(t \rightarrow y, S7)$ are made in the hash table. At statement $S2$ there will be no remaining remote tuples (after removing all entries in the hash table), and so nothing is done. At statement $S9$ there are three remaining tuples, all referencing pointer p, and thus a blocked communication is selected. For statements $S10$ to $S15$ there are no remaining tuples, and the next non-empty list is at statement $S3$. At statement $S3$ there are two pipelined reads via pointer *close*.

Handling the insertion of RemoteWrites is somewhat more difficult. In this case we must balance the cost of delaying some writes in order to expose opportunities for blocked writes. Further, it is not safe to write spurious fields in a block move. We handle these situations by introducing special RemoteFill tuples. These tuples ensure that all fields in a structure will be read before a blocked write is inserted.

We select remote write tuples in a bottom-up traversal, in order to expose the opportunity for blocked writes and redundancy elimination. Like the selection of remote reads, a hash table is used to hold the remote write tuples selected. At the beginning of the traversal, the hash table is empty. At each statementS_i, the RemoteWrites is examined, and any tuples already in the hash table are removed. With the updated RemoteWrites, we first consider the possibility of blocked writes, then of redundancy elimination. Notice that we do not consider pipelining remote writes at the cost of delaying them. This is ensured by the way we propagate and

have the option of issuing a remote operation to a potentially-invalid address. In this case one could speculatively issue the remote operation, even for an invalid address.

$$\{(t{\rightarrow}x, 11, S11:S4), (t{\rightarrow}y, 11, S12:S7)\}$$

```
S1: p = head;
S2: while (p!=NULL)
```

$$\{(p{\rightarrow}next, 1, S15), (p{\rightarrow}y, 1, S10), (p{\rightarrow}x, 1, S9)\}$$

```
    S9: { ax = p->x ;
    S10:  ay = p->y ;
    S11:  bx = t->x ;
    S12:  by = t->y ;
    S13:  dist = f(ax,ay,bx,by);
    S14:  if (dist < epsilon) close = p;
    S15:  p = p->next ;
        }
```

$$\{(close{\rightarrow}y, 1, S6), (close{\rightarrow}x, 1, S3)\}$$

```
S3: cx = close->x ;
S4: tx = t->x ;
S5: diffx = cx - tx;
S6: cy = close->y ;
S7: ty = t->y ;
S8: diffy = cy - ty;
```

(a) selected RemoteRead sets

```
S1a:comm1 = t->x ;
S1b:comm2 = t->y ;
S1: p = head;
S2: while (p!=NULL){
    S9a:blkmov(p,&bcomm1,sizeof(POINT));
    S9: ax = bcomm1.x;
    S10:ay = bcomm1.y;
    S11:bx = comm1;
    S12:by = comm2;
    S13:dist = f(ax,ay,bx,by);
    S14:if (dist < epsilon) close= p;
    S15:p = bcomm1.next;
        }
S3a:comm3 = close->x ;
S3b:comm4 = close->y ;
S3: cx = comm3;
S4: tx = comm1;
S5: diffx = cx - tx;
S6: cy = comm4;
S7: ty = comm2;
S8: diffy = cy - ty;
```

(b) after transformation

Figure 4.11: Applying Communication Selection

select the remote write tuples. When a blocked write is not chosen, we then proceed with only those tuples that are written by the current statement, and those that are propagated right out of the loop for redundancy elimination.

The placement of block writes proceeds in two steps. First, we select all variable candidates for blocked writes. At each statement S_i, after removing the hashed tuples from its RemoteWrites, we then check the updated RemoteWrites. If the number of remote write tuples for a variable exceeds a threshold, the variable will be selected as a candidate for a blocked write. The setting of a threshold depends on the size of a structure node, and the cost of a block write operation on a particular architecture. In EARTH-MANNA we set it to three. Once these candidates are selected, we then decide whether or not to blockwrite it for each of them. Figure 4.12 provides the algorithm for determining whether or not to blockwrite a variable at a statement. The basic idea is that we will only blockwrite the variables whose non-updated fields are already fetched to the local memory. In other words, we issue safe blocked writes, and will not generate any new remote reads because of these blocked writes. To examine the condition, a Remote-Fill set is introduced. A Remote-Fill set of a variable at S_i is a set that contains all the tuples of the variable that do not occur in the RemoteWrites of S_i, i.e. the complement set of the RemoteWrites for a variable at a statement. A Remote-Fill set is initially generated for a variable candidate, and is then propagated backwards. During the propagation, any tuples that have been covered by the RemoteReads of a statement are removed from the Remote-Fill set. The process continues until one of the following conditions is satisfied: (1) the Remote-Fill set is empty, we return with a success for a block write, (2) there is only one access in Remote-Fill set, and it is directly written by the statement, we also return with a success, (3) either the variable is written, or one of its tuples in Remote-Fill set is written via an alias, we return with a failure, or (4) S_i is the beginning of a function, we then return with a failure.

To illustrate the algorithm for placing block writes, let us look at the example in Figure 4.13. In this example, each statement is decorated with the RemoteWrites computed using the rules described in the previous section. The process of placing remote write tuples starts from $S13$, whose RemoteWrites set is empty. We thus proceed to $S12$. As there are more than three tuples of p in its RemoteWrites, p is selected as a candidate for a blocked write. We then consider its RemoteFill set. Assuming the example covers all the fields of p, the RemoteFill set for p at $S12$ will be a nonempty set ($p \rightarrow label$). This RemoteFill set is then propagated backwards to check if it is covered by any previous remote reads. The RemoteFill set remains unchanged until it reaches $S7$, whose RemoteReads contains a tuple $(p \rightarrow label, 1, S7)$. Thus ($p \rightarrow label$) is removed from RemoteFill set, and the set becomes empty. We return with a yes for placing a blocked write for p at $S12$. All the selected tuples

```
/* decide whether to blovk write a variable or not*/
fun isBlockwrite(stmt, var) =
  RemoteFill = ^ RemoteWrites(stmt, var);
  if (RemoteFill == Empty) return(1);
  while ( ! is_function_head(stmt))
    if (is_branch_head(stmt))  /* move out of a branch/switch */
        stmt = BranchStmt;
        RemoteFill = RemoteFill − commReadSet(stmt);
        if (RemoteFill == Empty) return(1);
    else (is_loop_head(stmt))  /* move out of a loop */
        stmt = LoopStmt;
        RemoteFill = RemoteFill − commReadSet(stmt);
        if (RemoteFill == Empty) return(1);
        if (varWritten(var, stmt) || fieldWritten(RemoteFill, stmt))
             return(0);
    else  /* sequence of statements */
        stmt = get_predecessor(stmt);
        RemoteFill = RemoteFill − commReadSet(stmt);
        if (RemoteFill == Empty) return(1);
        if (fieldSureWritten(RemoteFill, stmt)) return(1);
        if (varWritten(var, stmt) || writtenViaAlias(RemoteFill, stmt) )
             return(0);

  return(0);
```

Figure 4.12: Selection Analysis - Compound Rules

of p are then hashed. From $S12$, we continue the process for other statements. The RemoteWrites set for these statements are either originally empty, or become empty after removing the hashed tuples, so no more remote writes are placed. Figure 4.14 shows the code after both the read and write placement transformations.

```
S1: l = head; {}
S2: while (l!=NULL) {}
    S3: { p = l->patient; {}
    S4:   seed = p->seed; {}
    S5:   rand = my_rand(seed); {}
    S6:   p->seed = rand; {(p→seed, 1, S6)}
    S7:   label = p->label; {}
    S8:   if ( rand > 0.1 || label == 0)  {(p→seed, 1, S6), (p→status, 1, S14 : S15)}
       S14:    p->status = assessing; {(p→status, 1, S14)}
          else
       S15:    p->status = waiting; {(p→status, 1, S15)}
    S9:   time = p->time; {}
    S10:  p->time=time+10; {(p→seed, 1, S6), (p→status, 1, S14 : S15), (p→time, 1, S10)}
    S11:  left = p->tleft; {}
    S12:  p->tleft = left - 1;
       {(p→seed, 1, S6), (p→status, 1, S14 : S15), (p→time, 1, S10), (p→tleft, 1, S12)}
    S13:  l = l->next; {}
      }
```

Figure 4.13: Applying Write Communication Selection

Figure 4.15 provides another example for placing remote writes. The process starts from $S3$, and continues to $S1$, as the RemoteWrites of both $S3$ and $S2$ are empty. At $S1$, we first consider possible blocked writes. As the RemoteWrites contains only two tuples of *new*, no variable is selected as a candidate. We then consider the possibility of redundancy elimination. Due to the fact that both tuples are moved right after the loop, we decide to place these tuples at $S1$. All these tuples are then added to the hash table. After that, the RemoteWrites for the remaining statements become empty, so no more placement is done. The final code is shown in Figure 4.16.

4.3 Experimental Results

We perform a series of experiments on the suite of benchmarks described in Chapter 2.4, to evaluate the effect of communication optimizations. Similar to the locality

```
S1:  l = head;
S2:  while (l!=NULL)
     S3a:{ comm1 = l->next;
     S3:    p = l->patient;
     S4a:  blkmov(p, &comm2, sizeof(PATIENT));
     S4:    seed = comm2.seed;
     S5:    rand = my_rand(seed);
     S6:    comm3.seed = rand;
     S7:    label = comm2.label;
     S7a:   comm3.label = comm2.label;
     S8:    if ( rand > 0.1 || label == 0)
        S14:     comm3.status = assessing;
            else
        S15:     comm3.status = waiting;
     S9:    time = comm2.time;
     S10:   comm3.time = time + 10;
     S11:   left = comm2.tleft;
     S12:   comm3.tleft = left - 1;
     S12a: blkmov(&comm3, p, sizeof(PATIENT));
     S13:   l = comm1;
         }
```

Figure 4.14: Applying Write Communication Selection

```
S1: do {  {(new→x,1,S4),(new→y,1,S5)}
    S4:  new->x = getRandom(range); {(new→x,1,S4)}
    S5:  new->y = getRandom(range); {(new→x,1,S4),(new→y,1,S5)}
    S6:    ax = orig->x; {(new→x,1,S4),(new→y,1,S5)}
    S7:    Dx = new->x - ax; {(new→x,1,S4),(new→y,1,S5)}
    S8:    ay = orig->y; {(new→x,1,S4),(new→y,1,S5)}
    S9:    Dy = new->y - ay; {(new→x,1,S4),(new→y,1,S5)}
    S10:   dist = Dx * Dx + Dy * Dy;  {(new→x,1,S4),(new→y,1,S5)}
       } while (dist > radius * radius);
S2:     if ( minx > abs(Dx))  {}
S11:       minx = abs(Dx);
S3:     if ( mixy > abs(Dy))  {}
S12:       miny = abs(Dy);
```

Figure 4.15: Applying Communication Selection

```
S1a: comm1 = orig->x;
S1b: comm2 = orig->y;
S1:  do {
     S4:  comm3 = getRandom(range);
     S5:  comm4 = getRandom(range);
     S6:  ax = comm1;
     S7:  Dx = comm3 - ax;
     S8:  ay = comm2;
     S9:  Dy = comm4 - ay;
     S10: dist = Dx * Dx + Dy * Dy;
         } while (dist > radius * radius );
S1c: new->x  = comm3;
S1d: new->y  = comm4;
S2:  if ( minx > abs(Dx))
     S11:       minx = abs(Dx);
S3:  if ( mixy > abs(Dy))
     S12:       miny = abs(Dy);
```

Figure 4.16: Applying Communication Selection

experiments in section 3.4, we use the same *simple* version of each benchmark as a basis, apply the communication optimizations, and compare the performance with and without communication optimizations. In the following sections, we analyze the results we get. We refer to the optimized programs as *optimized* versions.

Dynamic Counts for Communication

In Figure 4.17, we compare the number of remote communication operations for the *simple* and *optimized* versions. The number under each benchmark name gives the total number of communication operations in millions. Each benchmark has two bars, the left bar represents the number of communication operations performed by the *simple* version, normalized to 1. The right bar shows the number of remote operations performed by the *optimized* version. In each bar we indicate the proportion that are read-data, write-data and blkmovs.[3] It is clear that in all cases the total number of communication operations reduces. The number of read-data and write-data operations reduce because of redundant communication elimination and blocking. The number of blkmov operations increases because some individual read-data/write-data operations were combined into new blkmov operations.

[3]Note that the unoptimized versions contain some blkmovs because the compiler inserts blkmovs for assignments to entire `structs`.

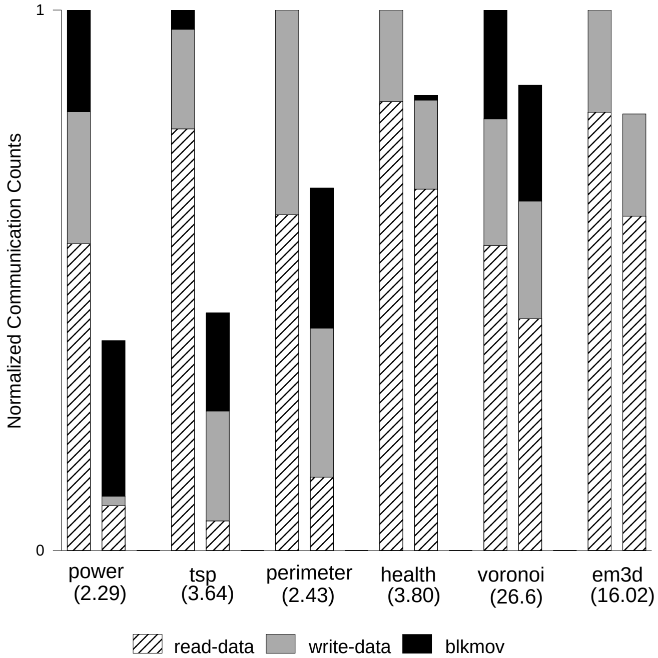


Figure 4.17: Improvement on Dynamic Communication Counts

Although the data in Figure 4.17 gives us some insight into the number of communication operations performed, it does not indicate the total communication volume (i.e. total number of bytes read/written remotely). We can expect some reduction in volume due to the removal of redundant reads/writes, but we can also expect some additional volume because of blocking since a blkmov operation is used to read/write the entire structure, and not just those fields which are actually needed. Figure 4.18 provides some insight into traffic volume.

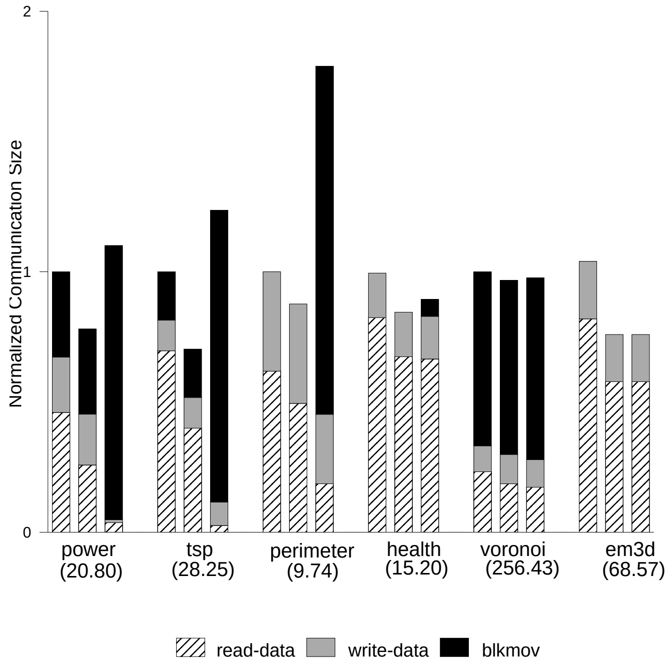


Figure 4.18: Improvement on Dynamic Communication Size

In Figure 4.18 there are three bars for each benchmark corresponding to the following cases:

simple: the left bar represents the *simple* version where no communication optimization was performed;

nonblocked: the middle bar represents an *optimized* version where communication is moved and redundant communication is eliminated, but no blocking is performed; and

full: the right bar represents the fully optimized version where blocking is included.

The height of the bar represents the number of bytes read/written/block-moved, normalized to 1 for the simple case. By comparing the height of the left (simple) and middle (nonblocked) cases we can see how much redundant communication was eliminated. Note that five of the benchmarks (`power`, `tsp`, `perimeter`, `em3d` and `health`) do have significant reductions, whereas one benchmark (`voronoi`) does not have much reduction.

By comparing the height of the the middle (nonblocked) and right (full) bars we can see how much extra communication is added due to blocked communication. When blocking is included we use one blocked operation to read or write a complete struct. Often not all of the fields of the struct will actually be needed, so even though there is a decrease in the number of communication operations, there can be an increase in the number of bytes communicated. We see from Figure 4.18 that this is indeed the case, and for most the benchmarks there is an increase in the total number of bytes communicated. In some benchmarks, such as `perimeter`, this increase is quite noticeable. However, as we show in the next section, the blocking does give performance improvement, even though total volume goes up.

Performance Improvement

The ultimate measure of success of our optimization is runtime speed improvement. In Table 4.1 we provide the performance improvement achieved for each benchmark.

The first data column gives the sequential execution time for each benchmark. This time is measured for a purely sequential version of the benchmark running on one node of the MANNA machine. In these versions all data is local, there is no parallelism, and there are no calls to any EARTH runtime operations. Thus, this is a truly sequential program, with no extra overhead. The next two columns give the times for the simple and the fully optimized parallel versions of the benchmark, for 1, 2, 4, 8 and 16 processors. This is followed by two columns that give the speedup of the *simple* and *optimized* versions over the sequential version. Finally, the last column gives the performance improvement due to communication analysis. Note that communication analysis gives some performance improvement for all benchmarks. In general the performance improvement increases as the number of processors increase. For 16 processors we observe a performance improvement of 1.7% for the `em3d` benchmark to 16% for the `perimeter` benchmark. We found that these were quite satisfactory improvements since we were trying to improve the performance of benchmarks which already exhibited reasonable locality, and thus they did not have excessive communications. If we started with benchmarks with poor locality, we would expect communication optimization to give more improvement (since there is more communication to optimize), but we would not expect good overall performance on a distributed-memory architecture.

Since our measurements of communication volume showed that blocked communication reduced the number of communication operations, but sometimes increased total communication volume, we were interested in determining if blocking really did improve overall performance, or if most of the performance improvement is due to moving communication, and reducing redundant communication. Table 4.2 gives the execution time for the 16 processor case. The first column gives the execution time for the simple, unoptimized benchmark. The second large column gives the time for the nonblocked optimized case, and the percentage improvement over the simple case. The third large column gives the time for the fully optimized case, and the percentage improvement over the simple case. The last column shows the difference in improvement between the fully optimized and the nonblocked case. A positive number in this column for five benchmarks indicates the amount of performance improvement that is due to blocking.

It is important to realize that the performance improvement for the nonblocked case comes from two sources: (1) reducing redundant reads and writes, and (2) moving remote operations earlier so they can overlap with computation. Our results from Figure 4.18 would seem to indicate that quite a bit of redundant communication is found, and so we expect a large part of the improvement is due to eliminating these redundant operations. However, at least part of the improvement is also due to scheduling the operations earlier. We cannot really measure this impact because it is really an indirect impact on the thread generator which is a different component of the compiler.

It is interesting to note that blocking does have a significant effect on three of the benchmarks: (`tsp`, `perimeter` and `voronoi`). Thus, for an architecture that has blocked communication, blocking seems to be worth doing. Further, our analysis does give 1.7% to 14% performance improvement even without blocking. This means that architectures without blocked communication would still get a reasonable amount of benefit from our optimizations.

Example optimizations for individual benchmarks

We have reviewed the experimental results, and the generated optimized code for each benchmark in order to get some more detailed observations. Below we discuss

Benchmark		Seq. C (sec)	Simple EARTH-C (sec)	Opt EARTH-C (sec)	Simple Speedup	Opt Speedup	Opt vs. Simple (%impr)
powerI	1 proc	62.4	66.95	66.20	0.93	0.94	1.13
	2 procs		35.79	35.18	1.74	1.77	1.70
	4 procs		17.92	17.64	3.48	3.54	1.56
	8 procs		9.07	8.95	6.88	6.97	1.32
	16 procs		4.72	4.62	13.22	13.51	2.12
powerII	1 proc	62.4	67.76	66.76	0.92	0.93	1.48
	2 procs		35.96	34.41	1.74	1.81	4.31
	4 procs		18.58	17.58	3.36	3.55	5.38
	8 procs		9.77	9.12	6.39	6.84	6.65
	16 procs		5.23	4.86	11.93	12.84	7.07
tsp	1 proc	23.8	26.95	26.26	0.88	0.91	2.56
	2 procs		14.02	13.56	1.70	1.76	3.28
	4 procs		7.71	7.33	3.09	3.25	4.93
	8 procs		4.67	4.29	5.10	5.55	8.14
	16 procs		3.77	3.32	6.31	7.17	11.94
health	1 proc	143.72	144.09	144.05	0.99	0.99	0.03
	2 procs		84.25	80.72	1.71	1.78	4.19
	4 procs		48.32	44.78	2.97	3.21	7.33
	8 procs		42.74	37.69	3.36	3.81	11.82
	16 procs		33.88	28.84	4.24	4.98	14.88
perimeter	1 proc	5.45	7.32	6.75	0.74	0.81	7.79
	2 procs		3.67	3.35	1.49	1.63	8.72
	4 procs		2.16	1.94	2.52	2.81	10.19
	8 procs		1.12	0.98	4.87	5.56	12.50
	16 procs		0.75	0.63	7.27	8.65	16.00
voronoi	1 proc	3.98	7.27	6.78	0.55	0.59	6.74
	2 procs		3.23	2.85	1.23	1.40	11.76
	4 procs		2.52	2.13	1.58	1.87	15.48
	8 procs		1.31	1.17	3.04	3.40	10.69
	16 procs		1.04	0.88	3.83	4.52	15.38
em3d	1 proc	16.26	29.15	26.89	0.56	0.60	7.75
	2 procs		21.79	20.66	0.75	0.79	5.19
	4 procs		10.95	10.38	1.48	1.57	5.21
	8 procs		5.94	5.87	2.74	2.77	1.18
	16 procs		3.48	3.42	4.67	4.75	1.72

Table 4.1: Performance Improvement from Communicaiton Optimizations

Benchmark	Simple (sec)	Nonblocked (sec)	Nonblocked (% decr.)	Full (sec)	Full (% decr.)	Diff. (Full-Nonblocked)
powerI	5.23	4.90	6.30%	4.86	7.07%	0.77%
powerII	5.23	4.90	6.30%	4.86	7.07%	0.77%
tsp	3.77	3.49	7.42%	3.32	11.93%	4.51%
health	33.88	28.90	14.70%	28.84	14.88%	0.18%
perimeter	0.75	0.70	6.67%	0.63	16.00%	9.33%
voronoi	1.04	0.92	11.54%	0.88	15.38%	3.84%
em3d	3.48	3.42	1.72%	3.42	1.72%	0.00%

Table 4.2: Performance for 16 Processors

some details about each benchmark.

Power: This benchmark implements the power system optimization problem. It uses a four-level tree structure with different branching widths at each level. There are two *simple* versions for `power`. `PowerI` is a version in which major computation functions in the program are invoked using the `@OWNER_OF` primitive, while in PowerII, these functions are called without any call site specifications, thus depending on the runtime system to distribute them across the architecture.

Communication optimization reduces the number of remote communications by 61% for both versions of the benchmark, as the optimizations are independent of the call site information. The main benefit comes from blocking. It is computation-intensive benchmark, with functions operating on a particular node of the tree. These functions typically read the fields from a node into scalars at different points, perform computation, and write the values back into the field.

Since all field accesses are with respect to a given node, communication optimization is able to place all the read accesses early, and all the write accesses late in the function. Subsequently, these accesses are blocked, as they correspond to fields within a specific node. An example code fragment from this benchmark is shown in Figure 4.19(a), which illustrates this observation.

Although the amount of remote communications reduced is the same for both versions of the benchmark, their effect on the performance improvement is different, with an average 1.5% for `powerI`, and 5.0% for `powerII`. This difference comes from the fact that, with major function calls in `powerI` invoked with the `OWNER_OF` primitive, the remote accesses reduced in `powerI` are mainly *pseudo remote* accesses, which are less expensive than the actual remote accesses, as is the case for `powerII`. Nevertheless, we find that, with the communication optimizations, the performance of `powerII`, a version with less programmer effort, comes close to that of `powerI`, a version which requires more programmer effort.

Tsp: This benchmark solves the traveling salesman problem using a divide-and-conquer approach based on a close-point algorithm. This algorithm first searches a suboptimal tour for each subtree(region) and then merges subtours into bigger ones. The tour found is built as a circular linked list sitting on top of the root nodes of subtrees.

This benchmark is irregular in nature and performs a significant time in data accesses. We are able to obtain up to 11.94% speed-up for it. The benchmark mainly benefits from redundant communication elimination and communication pipelining. As illustrated in Figure 4.19(b), remote communication expressions with respect to the pointer `t` are pulled out of the loop, and the two `blkmov` operations that were moved out of the loop can be pipelined.

Perimeter: This benchmark computes the perimeter of a quad-tree encoded raster image. The unit square image is recursively divided into four quadrants until each one has only one point. The tree is then traversed bottom-up to compute the perimeter of each quadrant.

Similar to `tsp`, this benchmark is irregular, and each computation requires accesses to tree nodes which may not be physically close to each other. Thus it is a communication-intensive benchmark, and the benefits of communication optimization are more visible for it. We are able to achieve up to 16% speed-up over the *simple* version. The main optimization applied for this benchmark is also blocking. An example code fragment from `perimeter` is shown in Figure 4.20(a). Here the `blkmov` replaces three remote read operations. The optimization is applied within a recursive function, where the program spends most of the time. Hence substantial speed-up is obtained.

Health: This benchmark simulates the Colombian health-care system using a 4-way tree. Each village has four child villages, and a village hospital, treating patients from the villages in the same subtree. At each time step, the tree is traversed, and patients, once assessed, are either treated or passed up to the parent tree node. The 4-way tree is evenly distributed among the processors and only top-level tree nodes have their children spread among different processors.

```
Compute_Branch(br, theta_R, ....)
{ ...
  blkmov(br,&bcomm7, sizeof(branch));
  if ((next != 0)) {
   { bcomm7.D.P= (temp_258 + temp_259);
     bcomm7.D.Q= (temp_260 + temp_261);
   }
  else
   { bcomm7.D.P = tmp.P;
     bcomm7.D.Q = tmp.Q;
   }
  temp_263 = bcomm7.R;
  temp_262 = (temp_263 * temp_263);
  temp_266 = bcomm7.X;
  temp_265 = (temp_266 * temp_266);
  ...
  bcomm7.alpha = (a / temp_313);
  bcomm7.beta = (b / temp_315);
  temp_317 = bcomm7.D;
 blkmov(&bcomm7, br, sizeof(branch));
  ...
}
```

(a) Power

```
tree choose_min(t,a,head,prev_min,prev_dist)
{blkmov(t, &bcomm6, sizeof(tree));
 blkmov(a, &bcomm7, sizeof(tree));
  ax = bcomm6.x;
  ay = bcomm6.y;
  bx = bcomm7.x;
  by = bcomm7.y;
  temp_84 = (ax - bx);
  temp_87 = (ay - by);
  ...
  tmp = bcomm7.next;
  while (tmp != b)
  {blkmov(tmp, &bcomm8), sizeof(tree));
    ax = bcomm8.x;
    ay = bcomm8.y;
    bx = bcomm6.x;
    by = bcomm6.y;
    ...
    tmp = bcomm8.next; }
  ...
}
```

(b) Tsp

Figure 4.19: Benchmark Codes Illustrating Communication Optimization

```
int R_sum_adjacent(p, q1, q2, size)
 { ...
  blkmov(p,&bcomm,sizeof(quad_struct)) ;
  temp_110 = bcomm.color;
  if ((temp_110 == 2))
  { switch (q1)
    { case 0: p1 = bcomm.nw;
              break;
      case 1: p1 = bcomm.ne;
              break;
      ...}
    switch (q2)
    { case 0: p2 = bcomm.nw;
              break;
      case 1: p2 = bcomm.ne;
              break;
      ...}
    x = R_sum_adjacent(p1, q1, q2, size_1);
    y = R_sum_adjacent(p2, q1, q2, size_1); }
  else
  { temp_112 = bcomm.color;
    if ((temp_112 == 1)) ...}
}
```

(a) perimeter

```
void check_patients_inside(village, list)
{ ...
  comm6 = (*village).hosp.free_personnel;
  while ( (list != 0) )
  { p = (*list).patient;
    comm1 = (*list).forward;
    comm5 = (*p).time_left;
    comm5 = comm5 - 1;
    (*p).time_left = comm5;
    temp_56 = comm5;
    if ((temp_56 == 0))
    { t = comm6;
      comm6 = (t + 1);
      l = (&(*village).hosp.inside);
      R_removeList(l, p);
      l = (&(*village).returned);
      ... }
    list = comm1; }
  (*village).hosp.free_personnel = comm6  ;
}
```

(b) health

Figure 4.20: Benchmark Codes Illustrating Communication Optimization

This benchmark has relatively few remote data accesses. Thus the speed-up obtained via communication optimization is not as significant. The communication optimizations applicable to this benchmark include pipelining, and redundancy elimination, as illustrated via the code fragment taken from it, shown in Figure 4.20(b).

Voronoi: The `voronoi` benchmark computes a Voronoi Diagram for a random set of points. The points are generated and stored in a binary tree, and the algorithm computes the Voronoi Diagrams of the two subtrees recursively, and merges them to form the final diagram.

The merge phase walks along the convex hull of the two sub-diagrams, alternating between in an irregular fashion, so the benchmark spends a significant time in data accesses. Therefore, we obtain up to 15.48% speedup by communication optimization. This mainly comes from redundant communication elimination and blocking.

Em3d: The `em3d` benchmark models the propagation of electromagnetic waves through objects in three dimensions. It does the computation on an irregular bipartite graph. At each time step, new waves for the E nodes are computed from a weighted sum of the neighbouring H nodes, and new waves for the H nodes are computed similarly from the E nodes.

The computation of new waves for E and H nodes invoke many data accesses. With communication optimizations, we achieve up to 7.7% speedup for this benchmark. The benefit mainly comes from redundant communication elimination.

4.4 Related Work

The closest related work is previous research at McGill on techniques to reduce communication overhead in parallel EARTH-C programs that use pointer-based dynamic data structures [TGHG97].

Tang et. al [TGHG97] use standard compiler optimizations like common subexpression elimination, loop- and location-invariant removal to eliminate redundant pointer dereferences, and examine the effect of these optimizations on the quality of threaded code produced by the EARTH-C compiler. The communication analysis presented in this chapter can also reduce redundant pointer dereferences (for example, when remote memory accesses are moved out of loops). However, in this chapter we are also concerned about the placement of the pointer dereferences, pipelining remote operations, and blocking of associated remote operations.

Another approach for compilation of dynamic data structure based applications on distributed memory machines was proposed by Carlisle and Rogers [CR95, Car96]. They propose the Olden runtime system, that uses a trade off between software caching of remote data and computation migration, depending on the data distribution and the amount of communication required. This decision is guided, in part, by programmer-specified *path-affinity* hints for recursive data structures. If accessing remote data is chosen, a software caching scheme is used to minimize the overhead of the communication. In our approach, the programmer makes the choice of whether or not to migrate computation using explicit constructs such as `@OWNER_OF` and `@HOME`, which enable the programmer to invoke functions at a given processor. Therefore, our communication optimization focuses on further reducing the overhead of communication that is considered necessary by the EARTH-C programmer, i.e., the communication that cannot be effectively substituted by computation migration. Instead of using a software caching scheme which requires runtime checks, our analysis decides at compile-time which pointer dereferences are reused, and which references can be blocked together. An interesting point would be to consider how the software caching scheme could benefit from our analysis.

Another form of communication optimization is prefetching. The most relevant work is compiler-based prefetching for recursive data structures by Luk and Mowry [LM96]. This work was directed towards reducing the memory latency time for superscalar processors. The basic idea is to automatically insert prefetch instructions for pointer references using three different schemes, including a scheme whereby nodes pointed to by some pointer p are greedily prefetched (for example, prefetching the nodes pointed to by $p \rightarrow left$ and $p \rightarrow right$ as soon as $p \rightarrow value$ is accessed). Our analysis is not trying to prefetch along a traversal chain, rather we are concentrating on moving pointer dereferences early, and on blocking related dereferences together (i.e. we would read the complete node pointed to by p at the same time, we would not speculatively dereference any of those fields.) Further, in our case we are concerned with distributed memory parallel processors, and so we have a different cost model. In our case it is always a good idea to move accesses early (our results are written to memory and not a cache), and, in general, it is too expensive for us to speculatively prefetch pointer references.

For array-based scientific computations, a significant amount of work has been done on communication optimization for distributed memory compilation [ASD95, AL93, CGC96, vHK94]. The approach proposed by Chakrabarti et. al [CGC96] is the most recent. They propose a global scheduling approach for communication optimization of F90/HPF programs.

Our strategy is similar to their approach in the scheduling aspect. In their method they calculate an earliest and latest point for each remote read. Their earliest point is similar to our earliest point since in both cases we chose the earliest single dominator point. Their latest point captures the idea that a read should be moved to outermost loops, if possible. We do not compute a latest point explicitly, but get this effect using our frequency estimate, which favours placing reads/writes in outer loops. However, our frequency estimate is also useful for dealing with partial redundancies and moving reads/writes out of conditional statements. For placing the communications we follow the earliest placement policy in order to maximize overlap of communication and computation, whereas in their model they give more weight to reducing communication volume, and so they may not chose the earliest location. Their strategy is useful when aggregating messages from different arrays, whereas our strategy is best when aggregating field references to the same structure.

A major difference is that they consider only *read* remote communications, while we optimize both *read* and *write* remote accesses. Furthermore, we are handling dynamically-allocated pointer data structures and recursive programs, while their work is targeted towards array-based scientific programs with nested loops, where compiler optimizations traditionally work very well.

Agrawal et. al [ASD95] indicated that interprocedural partial redundancy elimination can be beneficial for eliminating redundant communication for array-based scientific applications. This observation also applies to programs that use dynamic data structures. In our *tsp* benchmark, one of the pointer parameters passed to the function `distance` remains invariant across several calls to the function, and all the field accesses with respect to this pointer can be placed before the first call by interprocedural partial redundancy elimination. Therefore, we would benefit more from an interprocedural communication optimization. Currently, we achieve part of this effect via function inlining.

Krishnamurthy and Yelick [KY95] present communication optimizations in the context of compiling explicitly parallel programs to Split-C programs. Their optimizations include message pipelining by converting remote read/writes into their split-phase analogues, eliminating acknowledgment traffic, and reusing values from remote memory accesses. Their source language includes shared scalar variables and distributed arrays, but does **not** include global pointers. The emphasis of their work is an optimization framework that handles explicitly parallel programs that may read/write the same memory locations, and thus they must correctly handle interfering parallel computations. In contrast, we have a simpler parallel programming model, where our parallel threads do not interfere on ordinary remote memory accesses. However, we **do** allow global pointers, and in fact the focus of our work is optimizing programs using global pointers. Presumably the strengths of these two approaches could be combined to handle both the more general parallel programming model and global pointers.

For shared-memory models, transformations similar to communication optimization, can be used to reduce the synchronization overhead of the program [DR97]. Diniz and Rinard [DR97] present an approach that focuses on coalescing multiple critical sections that acquire and release the same lock multiple times, into a single critical region that acquires and releases the lock only once. To this end, they perform *lock movement* and *lock cancelation* transformations, which try to reduce the frequency with which the program acquires and releases locks. These transformations are similar in spirit to the redundant communication elimination optimization.

The elimination of redundant code has been examined quite rigorously for scalar computations. As an example, Knoop. et. al. optimize the computation by optimal code motion [KRS94]. They issue the computation as late as possible to avoid unnecessary register pressure while maintaining computational optimality. They focus on scalar variables, while we are interested in pointer dereferences. Further their algorithms treat each computation independently, while we consider the interaction between tuples in determining the best placement, and we consider the costs of different communication strategies to decide between pipelining and block-moving.

4.5 Summary

In this chapter we have presented a communication analysis framework that is composed of *possible-placement analysis* and *communication selection*. The fundamental idea is that we wish to move remote read operations as early as possible in order to allow for overlap between communication and computation, and to expose redundant communication. Further, we wish to pipeline or block remote reads and writes when beneficial.

We presented *possible-placement* analysis as a structured analysis on the SIMPLE representation of the EARTH-McCAT compiler. In this analysis RemoteRead tuples are propagated optimistically upwards, and RemoteWrite tuples are propagated conservatively downwards. In order to determine when it is safe to propagate tuples, both stack and heap read/write sets are required.

The *communication selection* transformation uses the results of *possible-placement analysis*. The goal of the analysis is to locate the earliest point for remote reads, and to generate either pipelined or blocked communication. For remote writes, the communication is delayed if this enables blocked communication.

We implemented these techniques in the EARTH-McCAT compiler, and experimented with a set of benchmarks. These results show that the optimizations improve performance up to 16% over the unoptimized programs. We performed additional experiments to measure the amount of redundant communication found, and to investigate the impact of the blocking part of our optimization.

Chapter 5

Combining Locality and Communication Optimizations

In the previous two chapters, we presented two techniques to reduce the communication overhead: locality techniques and communication optimization techniques. Locality techniques reduce remote communications by detecting when they can be safely assumed local, thus replacing them with local references, while communication optimization techniques reduce communication overhead by moving communications ahead, eliminating redundant communications and blocking or pipelining communications. A set of experiments have been conducted independently to evaluate each of these techniques. The experimental results show that both techniques effectively improve the performance.

As both techniques are targeted for communication elimination on distributed memory architectures, it would be interesting to study the joint effect of these techniques. To this end, we performed another set of experiments on the benchmarks suite with both techniques enabled. The experimental results are presented and analyzed in this chapter. These results demonstrate that locality techniques and communication optimization techniques are both useful, and although there is some overlap between the two techniques, the performance improvement is best when both techniques are applied.

5.1 Experimental Results

We conducted a set of experiments in order to evaluate the combined effect of locality techniques and communication optimizations. Like the experiments in the previous chapters, we take a *simple* version of each benchmark as a basis, apply both locality and communication optimization techniques. We apply the static locality analysis first, and then invoke communication analysis to optimize the reduced set of remote communications. In the following sections, we analyze these results, and compare them with the results in the previous sections. We will refer to the combined version as *combined* where both locality and communication analyses are applied.

5.1.1 Overall Performance Improvement

Table 5.1 summarizes the performance improvement from these techniques. The *simple* column provides the execution time for the *simple* version on EARTH-MANNA in units of seconds (the same data as in column 1 of Table 3.2, Chapter 3). The *localized* column reports the percentage speedup obtained over the *simple* version by static locality techniques (the same data as in column 4 of Table 3.2, Chapter 3). The *comm-opt* column gives the percentage speedup obtained over the *simple* version by communication optimizations (the same data as in column 6 of Table 4.1. Chapter 4). Finally, the last *combined* column shows the percentage speedup over the *simple* version when both techniques are applied. We provide data for benchmark runs on 1, 8 and 16 nodes.

The data in Table 5.1 shows that the locality techniques improve the performance for all benchmarks, and it particularly dominates the performance improvement for benchmarks `powerI` and `em3d`. This indicates that there exists much data locality in our benchmark suite as a result of a good data distribution from the programmer, and this locality is effectively detected and exploited by our locality techniques. Communication optimization techniques compete well with static locality techniques for benchmarks `tsp`, `perimeter` and `health`, and it outperforms the locality techniques for benchmarks `powerII` and `tsp`. The *combined* version achieves the best performance for all benchmarks. This best performance improvement sometimes occurs as an accumulation of the performance improvement for each kind of techniques individually for some benchmarks, for example, `powerII` and `tsp`, while in other benchmarks, it appears better than ones from each of the techniques, but less than their sum. Usually, the overall performance improvement depends on the degree of the overlapping between the remote communications being optimized by each kind of technique.

Benchmark		Simple EARTH-C (sec)	Localized vs. Simple (%impr)	Comm-opt vs. Simple (%impr)	Combined vs. Simple (%impr)
powerI	1 proc	66.95	3.99	1.13	4.02
	8 procs	9.07	4.19	1.32	4.19
	16 procs	4.72	4.45	2.12	4.66
powerII	1 proc	67.76	1.83	1.48	3.20
	8 procs	9.77	1.54	6.65	7.88
	16 procs	5.23	1.53	7.07	7.65
tsp	1 proc	26.95	1.96	2.56	6.79
	8 procs	4.67	2.78	8.14	11.35
	16 procs	3.77	3.45	11.94	13.00
health	1 proc	144.09	0.17	0.03	1.87
	8 procs	42.74	14.72	11.82	15.86
	16 procs	33.88	15.76	14.88	16.17
perimeter	1 proc	7.32	14.89	7.79	15.85
	8 procs	1.12	16.96	12.50	18.75
	16 procs	0.75	21.33	16.00	22.67
voronoi	1 proc	7.27	14.86	6.74	15.41
	8 procs	1.31	15.27	10.69	16.03
	16 procs	1.04	16.35	15.38	17.31
em3d	1 proc	29.15	21.78	7.75	29.43
	8 procs	5.94	26.77	1.18	28.96
	16 procs	3.48	24.14	1.72	26.44

Table 5.1: The Overall Performance Improvement Results

5.1.2 Improvement on Individual Benchmarks

Table 5.1 gives us a general idea about the performance improvement from both techniques. To gain further insight on the interaction between these two optimization techniques, we collect the dynamic counts of remote communications for the *combined* version of each benchmark, and compare them with the counts of the *localized* and *comm-opt* versions, the versions that each kind of technique is applied individually. Below we analyze these dynamic counts and the percentage performance improvement for each benchmark in detail.

We categorize the benchmarks into two categories: the first category includes the benchmarks where the performance for the *combined* version is an accumulation of the performance from each kind of technique, and the second category contains the rest of benchmarks in the suite.

For each benchmark in a category, we provide a bar graph, associating the performance improvement with the reduction on the number of remote communications. There are two sets of bars for each graph, with four bars in each set, corresponding respectively to the four versions of each benchmark. These four versions are in the order of *simple*, *localized*, *comm-opt* and *combined*. The left set of bars represents the total execution time on 16 nodes for each version, with the leftmost bar for the *simple* version normalized to 1. The right set of bars indicates the dynamic counts of remote communications. These counts are the same independent of the number of the processors used. Further, each bar in the right set is composed of three parts, with each part indicating the proportion that are read-data, write-data and blkmovs. The leftmost bar in the right set of the *simple* version is also normalized to 1.

Benchmarks Showing Accumulated Benefit

This category contains benchmarks whose performance from both techniques is a sum of the performance from each individual technique. The benchmarks belonging to this category are `powerII`, `tsp` and `em3d`. In `em3d`, static locality techniques dominate the overall performance, while in `powerII` and `tsp`, communication optimization techniques outperform the locality techniques.

Let us first look at Figure 5.1 for powerII benchmark. In both set of bars, the *combined* version is almost the same height as the *comm-opt* one, which indicates the fact that the performance improvement for the *combined* version mainly comes from the communication optimization techniques. The right set of bars further reveals that this benefit is a result of both redundancy elimination and blocking operations.

Nevertheless, comparing to the *comm-opt* version, both read-data and blkmov parts for the *combined* version are slightly shorter, indicating the small effect of locality techniques on the *combined* version. Finally, as `power` is a computation intensive benchmark, we observe that nearly 60% reduction on the number of remote communications only leads to 7% performance improvement for the *combined* version on 16 nodes.

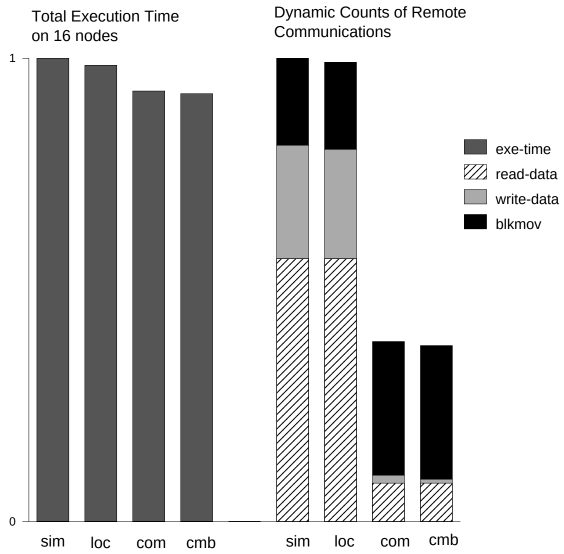


Figure 5.1: Performance Improvement for PowerII

`Tsp` is another benchmark where the *combined* version enjoys the improvement from both techniques, as shown in Figure 5.2. The *localized* bars in both sets show that the 5% performance improvement from locality techniques comes from the reduction on the number of both remote reads and writes. The *comm-opt* bars indicate that redundancy elimination and blkmov operations techniques are able to achieve about 11% performance speedup. As a result, the locality techniques contribute to the reduction on the number of remote writes for *combined* version, while both techniques may help reduce its number of remote reads.

`Em3d` is a benchmark where there are no blkmov operations involved for all three versions, as shown in Figure 5.3. We observe that, comparing with the *simple* version, the *localized* version enjoys a reduction on the number of both remote reads and

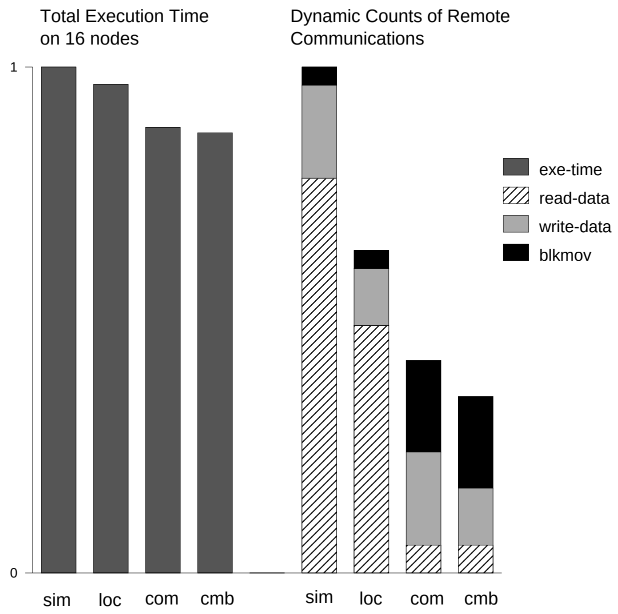


Figure 5.2: Performance Improvement for Tsp

remote writes, while the improvement for the *comm-opt* version comes mainly from redundancy elimination of remote reads. Therefore, the *combined* version inherits the reduction on the number of remote writes from locality techniques, and its reduction on the number of remote reads is a result of both techniques.

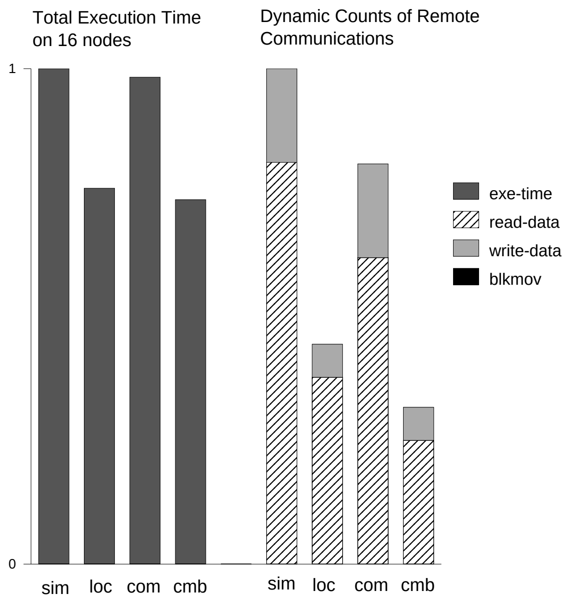


Figure 5.3: Performance Improvement for Em3d

Benchmarks Showing Overlapping Benefit

This category summarizes the benchmarks where there is a large overlap between the remote communications being optimized by locality techniques and communication optimization techniques. As a result, the overall performance is better than the performance from each technique, but not as great as the accumulation of both techniques. This category includes benchmarks `powerI`, `perimeter`, `health` and `voronoi`.

Figure 5.4 shows the improvement for the `powerI` benchmark. We notice that the significant reduction on the number of remote communications for the *combined* version has a small effect on its overall performance, due to the fact that power is a computation intensive benchmark. Furthermore, the right set of bars indicates that the benefit for the *combined* version comes mainly from locality techniques, which successfully reduce most remote reads and writes.

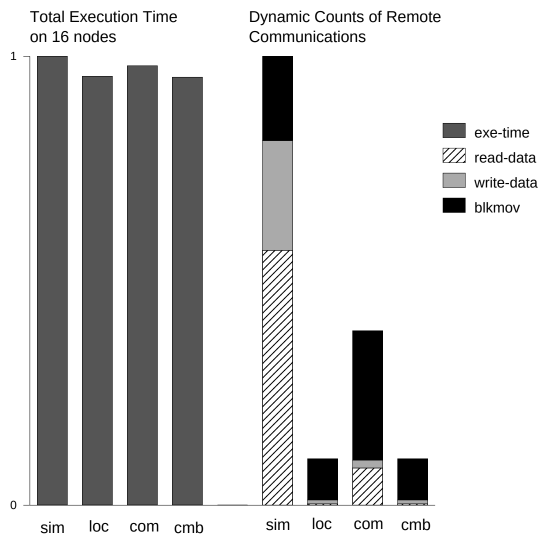


Figure 5.4: Performance Improvement for PowerI

Unlike `power`, the reduction on the dynamic counts of remote communications has a big effect on the performance improvement on `perimeter`, as shown in Figure 5.5. This is because `perimeter` spends many time on data accesses. We observe that the overall improvement for the *combined* version comes from a joint effort of both techniques, with the locality techniques playing an important part. Notice that though the total counts of remote communications for the *comm-opt* version is less than that of the *localized* version, its effect on the performance improvement is less obvious, as the *comm-opt* contains more blkmov operations, which are more expensive in general. The *combined* version inherits the benefit of blkmov operations from the *comm-opt* version, but the number of blkmov operations has been reduced by the locality techniques.

The left set of bars in Figure 5.6 for `health` benchmark shows that the three versions achieve almost the same amount of performance improvement. A look at

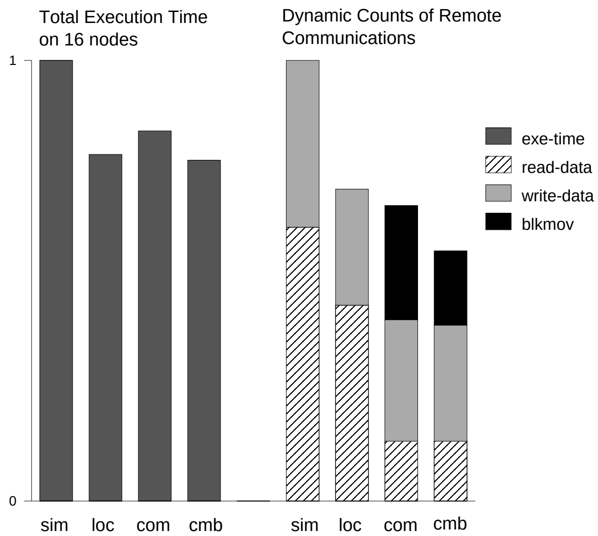


Figure 5.5: Performance Improvement for Perimeter

the right set of bars tells us that the benefit for all versions comes mainly from the elimination of remote reads, with only a few blkmov operations from communication optimization techniques in the *comm-opt* version and the *combined* version.

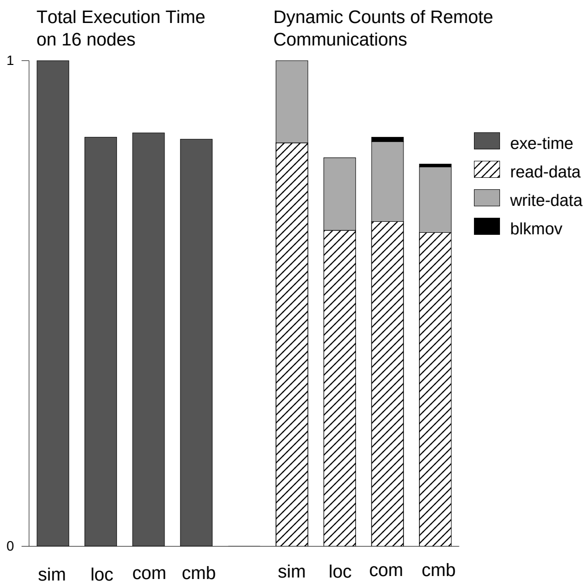


Figure 5.6: Performance Improvement for Health

The bar graph in Figure 5.7 for the `voronoi` benchmark resembles the `health` graph, in general. The left set of bars shows that the performance improvement for the *combined* version is a result of both techniques, with the locality techniques performing slightly better. Further the right set of bars indicates that the benefit mainly comes from the reduction of remote reads. Unlike `power`, the reduction on the remote communications has a significant effect on the overall performance, as the benchmark spends a considerable amount of time in data accesses.

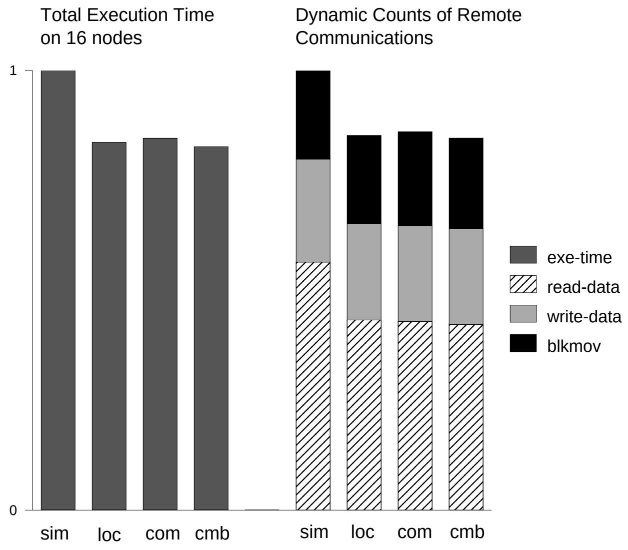


Figure 5.7: Performance Improvement for Voronoi

5.2 Summary

In summary, we find that the *combined* version, when both locality techniques and communication optimization techniques are applied, achieves the best performance for all the benchmarks. This demonstrates that these two techniques are orthogonal in general, as they use different kind of techniques to optimize different sets of remote communications. The static locality analysis uses a type-inference based technique to optimize those implicitly local references, while communication techniques optimize the remote communications by moving remote communications ahead, reducing redundant communications and pipelining or blocking them. Therefore, a good strategy for reducing remote communication overhead is to first use static locality analysis to handle all cases where the memory references must be local, and then use the communication techniques to optimize the remaining remote references. Nevertheless, some of the remote communications being optimized by the communication techniques may actually be implicitly local, as indicated by the bar graphs in section 5.1. This overlapping on the communications being optimized determines the overall performance improvement we can achieve for the *combined* version. As a result, for some benchmarks, the overall performance occurs as an accumulation of the improvement from each individual technique, while in other cases it appears better than the improvement from each individual technique, but less than their aggregation.

Chapter 6

Conclusions and Future Work

In this thesis we have presented effective techniques for reducing communication overhead on distributed memory architectures, and provided detailed experimental evaluation of the benefits achieved. The experiments were performed using the EARTH-McCAT compiler. We implemented the techniques in the EARTH-McCAT compiler framework and evaluated their effectiveness on the EARTH-MANNA architecture. Below we briefly summarize the main contributions and the salient points of this thesis.

We first presented a static locality analysis for parallel C programs, based on type-inference techniques (Section 3.2). These techniques use both locality information provided by the programmer and other program features like malloc sites to infer additional locality and to automatically specialize functions. We also introduced two potential dynamic techniques (fine-grain and coarse-grain), where dynamic locality checks are introduced into the program (Section 3.3).

We evaluated the effectiveness of the static locality analysis for a set of benchmarks and found that it can eliminate a significant number of pseudo-remote accesses and provide speedups ranging from a modest 4% up to 80% over the original parallel program (Section 3.4.1). For several benchmarks, this speedup also comes very close to the speedup obtained by an advanced hand-coded version. Furthermore, we found that the locality analysis reduced the burden on the programmer, and allowed us to develop shorter, more general benchmarks.

In order to compare the static techniques with dynamic techniques, we also provided experimental results comparing the static locality analysis, fine-grain dynamic checks, coarse-grain dynamic checks, and a combined approach (Section 3.4.2). We found that the fine-grain dynamic checks rarely provided any benefit, and often produced a degradation in performance due to too much overhead. The coarse-grain dynamic checks did give some benefit, but not as much as the static locality analysis. Further, it was quite difficult to find the best places to insert dynamic checks. The best overall technique was a combination of the static analysis and coarse-grain dynamic checks. This demonstrates that one should first use static techniques that introduce no run-time overhead, and then use dynamic techniques on the remaining interesting parts of the program.

For the communications that are not potentially local or cannot be detected local by a compiler, we also provided a set of communication optimization techniques (Section 4.2). The communication analysis framework is composed of *possible-placement analysis* and *communication selection*. The fundamental idea is that we wish to move remote read operations as early as possible in order to allow for overlap between communication and computation, and to expose redundant communication. Further, we wish to pipeline or block remote reads and writes when beneficial.

We presented *possible-placement* analysis as a structured analysis on the SIMPLE representation of the EARTH-McCAT compiler (Section 4.2.1). In this analysis RemoteRead tuples are propagated optimistically upwards, and RemoteWrite tuples are propagated conservatively downwards. In order to determine when it is safe to propagate tuples, both stack and heap read/write sets are required.

The *communication selection* transformation uses the results of *possible-placement analysis* (Section 4.2.2). The goal of the analysis is to locate the earliest point for remote reads, and to generate either pipelined or blocked communication. For remote writes, the communication is delayed if this enables blocked communication.

We also implemented these techniques in the EARTH-McCAT compiler, and experimented with a set of benchmarks (Section 4.3). These results show that the optimizations improve performance up to 16% over the unoptimized programs. We performed additional experiments to measure the amount of redundant communication found, and to investigate the impact of the blocking part of our optimization.

We did find significant redundant communication in five of the six benchmarks. It should be noted that even though finding redundant computation can be considered a standard optimization in many compilers, our communication algorithm is finding redundant pointer accesses to dynamically-allocated data structures, which is a problem that is not normally addressed. This part of our algorithm could be applied to sequential programs. However, we also aggressively move computations as early as possible in order to provide additional benefit for the thread scheduler. This part of the optimization would only be useful in the sequential setting if data prefetching is useful.

Blocking communication reduces the number of communication operations, but can potentially increase the communication volume. However, blocking does give a significant part of our performance improvement in three of the six benchmarks.

In order to evaluate the combined effect of both locality techniques and communication optimization techniques, we perform another set of experiments on the benchmark suite (Section 5). The experimental results we get show that the performance improvement is the best when both techniques are applied (Section 5.1). This proves the fact the locality techniques and communication techniques are orthogonal in general. Therefore, to achieve a maximum performance, a good strategy would be to first apply the locality techniques to identify those references that must be local, then use communication techniques to optimize those remaining remote references. We also observe that the total performance improvement depends on the overlapping of the communications being optimized by each individual technique, as in some cases the overall performance can be as good as the accumulation of the improvement from each technique, while in other cases it is less than that.

Above, we have summarized the main contributions of this thesis. These techniques presented in the thesis fits into the overall framework of the EARTH-McCAT compiler project where we are developing techniques to automatically generate low-level threaded programs from high-level parallel C programs. In the thesis we have focused on the techniques of one component - reducing the communication overhead. By adding this component to the compiler we get one step closer to closing the gap between compiler-generated threaded programs and hand-coded threaded programs.

6.1 Future Work

There are three major directions for future work. First, we plan to continue the study of locality techniques, develop flow-sensitive locality propagation techniques, that can exploit the locality of a pointer, even if it is local within only a specific section of the program. Based on this locality information, we can extend the specialization techniques to automatically generate a specialized section of code. Encouraged by the experimental results from the dynamic locality techniques, we would also like to automate the process of inserting coarse-grain dynamic checks based on compiler information. For example, we can use extended SSA numbers [LH98] to determine when one locality test can apply to a group of memory references (using the SSA property that specifies that two pointer references with the same primary SSA number must point to the same memory location). We would also like to do some research on extending locality analysis to approximate the probability of a static memory reference being local. This would also aid in the placement of dynamic checks.

Next, we would like to improve the communication optimizations by adding techniques for finding the best organization for fields within each `struct`. By placing those fields that are accessed remotely located close to one another, we can select only part of struct to block move, and thus further improve the efficiency of the blocked communication.

In an aim to close the gap between the compiler generated programs and hand-coded programs, we have focused on one important part in the thesis - reducing communication overhead. Another equally important area would be automatic parallelization. We plan to investigate in this area in the future. For example, we can incorporate the parallel loop detection techniques in [RGZ98] into EARTH-McCAT, and study how to use the shape information from the shape analysis [GH96b] to automatically detect parallel functions.

Bibliography

[AL93] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 126–138, Albuquerque, New Mexico, June 23–25, 1993. *SIGPLAN Notices,* 28(6), June 1993.

[ASD95] Gagan Agrawal, Joel Saltz, and Raja Das. Interprocedural partial redundancy elimination and its application to distributed memory compilation. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation* [?], pages 258–269. *SIGPLAN Notices,* 30(6), June 1995.

[BGSP94] U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat. Latency hiding in message-passing architectures. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 704–709, Cancún, Mexico, April 26–29, 1994. IEEE Computer Society.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* [?], pages 207–216. *SIGPLAN Notices,* 30(8), August 1995.

[Car96] Martin C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines.* PhD thesis, Princeton University Department of Computer Science, June 1996.

[CDG+93] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, Portland, Oregon, November 15–19, 1993. ACM SIGARCH and IEEE Computer Society.

[CGC96] Soumen Chakrabarti, Manish Gupta, and Jong-Deok Choi. Global communication analysis and optimization. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation* [?], pages 68–78. *SIGPLAN Notices,* 31(6), June 1996.

[CR95] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* [?], pages 29–38. *SIGPLAN Notices,* 30(8), August 1995.

[DR97] Pedro Diniz and Martin Rinard. Synchronization transformations for parallel computing. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 187–200, Paris, France, January 15–17, 1997.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation* [?], pages 242–256. *SIGPLAN Notices,* 29(6), June 1994.

[EH94] Ana M. Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 229–240, Toulouse, France, May 16–19, 1994. IEEE Computer Society Press.

[GH96a] Rakesh Ghiya and Laurie J. Hendren. Connection Analaysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 24(6):547–578, December 1996.

[GH96b] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–15, January 1996.

[GH98] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Conference Record of POPL'98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 121–133, San Diego, California, January 19–21, 1998.

[Ghi97] Rakesh Ghiya. *Putting pointer analysis to work.* PhD thesis, McGill University, Montréal, Québec, November 1997.

[Gho97] Shereen Ghobrial. Supporting shared variables and atomic functions in the earth-c language. Master's thesis, McGill University, Montréal, Québec, May 1997.

[HDE+92] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 406–420, New Haven, Connecticut, August 3–5, 1992. Springer-Verlag. Published in 1993.

[Hen91] F. Henglein. Efficient type inference for higher-order binding-time analysis. In *Functional Programming and Computer Architecture*, pages 448–472, 1991.

[HMT+96] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Guang R. Gao, and Laurie J. Hendren. A study of the EARTH-MANNA multithreaded system. *International Journal of Parallel Programming*, 24(4):319–347, August 1996.

[HTZ+97] Laurie J. Hendren, Xinan Tang, Yingchun Zhu, Shereen Ghobrial, Guang R. Gao, Xun Xue, Haiying Cai, and Pierre Ouellet. Compiling C for the EARTH multithreaded architecture. *International Journal of Parallel Programming*, 25(4):305–337, August 1997.

[JH94] Justiani and Laurie J. Hendren. Supporting array dependence testing for an optimizing/parallelizing C compiler. In Peter A. Fritzson, editor, *Proceedings of the 5th International Conference on Compiler Construction, CC '94*, number 786 in Lecture Notes in Computer Science, pages 309–323, Edinburgh, Scotland, April 7–9, 1994. Springer-Verlag.

[KRS94] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.

[KY95] Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel programs with explicit synchronization. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation* [?], pages 196–204. *SIGPLAN Notices,* 30(6), June 1995.

[LH98] Christopher Lapkowski and Laurie Hendren. Extended SSA numbering: Introducing SSA properties to languages with multi-level pointers. In *Proceedings of the 7th International Conference on Compiler Construction*, pages 128–143, Lisbon, Portugal, April 1998.

[LM96] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Cambridge, Massachusetts, October 1–5, 1996. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society. *Computer Architecture News,* 24, October 1996; *Operating Systems Review*, 30(5), December 1996; *SIGPLAN Notices,* 31(9), September 1996.

[MF98] Keith H. Randall Matteo Frigo, Charles E. Leiserson. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 54–67, Montréal, Québec, June 17–19, 1998. *SIGPLAN Notices,* 33(6), June 1998.

[MGH+96] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xin-Min Tian. Polling Watchdog: Combining polling and interrupts for efficient message handling. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 178–188, Philadelphia, Pennsylvania, May 22–24, 1996. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 24(2), May 1996.

[Ran98] Keith H. Randall. *Cilk: Efficient Multithreaded Computing.* PhD thesis, MIT, Boston, Massachusetts, June 1998.

[RCRH95] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[RD96] M. Rinard and P. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation* [?], pages 54–67. *SIGPLAN Notices,* 31(6), June 1996.

[RGZ98] Laurie J. Hendren Rakesh Ghiya and Yingchun Zhu. Detecting parallelism in C prgorams with recursive data structures. In *Proceedings of the*

1998 International Conference on Compiler Construction, pages 159–173, March 1998.

[SKSY94] Mitsuhisa Sato, Yuetsu Kodama, Shuichi Sakai, and Yoshinori Yamaguchi. EM-C: Programming with explicit parallelism and locality for the EM-4 multiprocessor. In Michel Cosnard, Guang R. Gao, and Gabriel M. Silberman, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '94*, pages 3–14, Montréal, Québec, August 24–26, 1994. North-Holland Publishing Company.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg, Florida, January 21–24, 1996.

[TGHG97] Xinan Tang, Rakesh Ghiya, Laurie J. Hendren, and Guang R. Gao. Heap analysis and optimizations for threaded programs. In *Proceedings of the PACT'97*, pages 14–25, San Francisco, November 1997. North-Holland Publishing Company.

[vHK94] Reinhard von Hanxleden and Ken Kennedy. Give-N-Take — a balanced code placement framework. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation* [**?**], pages 107–120. *SIGPLAN Notices,* 29(6), June 1994.