University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Heap Analysis and Optimizations for Threaded Programs

Xinan Tang
Rakesh Ghiya†
Laurie J. Hendren†
Guang R. Gao

CAPSL Technical Memo 11
November 7, 1997

Appeared	in the Proceedings of the
	1997 International Conference on
	Parallel Architectures and Compilation Techniques
	Novermber 11-15, 1997

†School of Computer Science, McGill Unviersity, Montréal, Québec, Canada

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

Traditional compiler optimizations such as loop invariant removal and common subexpression elimination are standard in all optimizing compilers. The purpose of this paper is to present new versions of these optimizations that apply to programs using dynamically-allocated data structures, and to show the effect of these optimizations on the performance of multithreaded programs.

In this paper we show how heap pointer analyses can be used to support better dependence testing, new applications of the above traditional optimizations, and high-quality code generation for multithreaded architectures. We have implemented these analyses and optimizations in the EARTH-C compiler to study their impact on the performance of generated multithreaded code. We provide both static and dynamic measurements showing the effect of the optimizations applied individually, and together. We note several general trends, and discuss the performance tradeoffs, and suggest when specific optimizations are generally beneficial.

KEYWORDS: heap analysis, heap optimizations, thread generation, multithreaded architectures, and compiling for parallel architectures.

Contents

1 Introduction **1**

2 EARTH-C Compiler Environment **2**
2.1 EARTH-C Language 3
2.2 Heap Analysis 4
2.3 Thread Generation in the EARTH-C compiler 7

3 Thread Partitioning and Optimizing Transformations **8**
3.1 Heap-based Optimizations 9
3.2 Thread Partitioning 9

4 Experimental Results and Analysis **12**
4.1 Benchmarks and EARTH-Manna system 12
4.2 Static Program Measurements 14
4.3 Runtime Performance 16
4.4 Performance Analysis 19
4.4.1 The Effect of Multithreading 20

5 Related Work **21**

6 Conclusion and Future Work **22**

List of Figures

1 EARTH-C compiler environment 3
2 Parallel Structures in EARTH-C 3
3 Sequential Structures in EARTH-C 3
4 Snapshots and related connection information 4
5 Example for Read-Write Sets 5
6 Analyses and Transformations in the EARTH-C compiler 7
7 Location Invariants 9
8 Code Segment Using the Optimizations 10
9 Useful Optimizations for Thread Generation 10
10 Optimization Effect on Thread Length 15

List of Tables

1 Data Structure and Running Parameters 13
2 Program Feature Statistics 15
3 Optimization Statistics 16
4 Timing Improvement After Optimizations 17
5 Improvement (%) on Remote Read-Write 18
6 Improvement (%) on Threads Generated 19
7 r_p changes with the thread and processor counts 21
8 Time Improvement (%) on Em3d 21

1 Introduction

Compiler analyses and optimizations are key components in traditional optimizing compilers for sequential languages. In fact, it is the presence of these optimizations that allows compiled code to compete with hand-written assembler code. When compiling programs written in parallel languages it is desirable to be able to perform similar optimizations to those applied to sequential programs and thus inherit the same power of these optimizations. However, analyses and optimizations for parallel programs pose different problems from those for sequential programs.

In this paper we address the problem of applying analyses and transformations designed for heap-allocated data structures. In particular, we integrate new kinds of standard optimizing transformations into our EARTH-C compiler that translates a parallel dialect of C, EARTH-C [9], to a low-level multithreaded language, Threaded-C [10, 14]. The overall philosophy of the EARTH-C project is that we wish to present a high-level language to the programmer, and have the compiler apply appropriate analyses and transformations in order to automatically produce efficient low-level threaded programs. In particular, we wish to support irregular applications that use dynamically-allocated data structures.

Although the compiler for our target low-level language supports standard optimizing transformations for scalars and arrays, we have found that new techniques are required to fully support our target applications that use dynamically-allocated structures heavily. Thus, we have developed further high-level analyses and transformations that concentrate on dependence analysis and optimization of memory references to the heap. In this paper we present: (1) new forms of dependence testing for heap-allocated structures, (2) new versions of standard transformations including *common sub-expression elimination*, and *loop-invariant removal*, (3) a new type of transformation that we call *location-invariant removal*, and (4) a thread generation strategy making full use of the new dependence analysis and transformations. Sharper dependence testing for heap-directed pointers provides more flexibility to the compiler's thread generator, and thus leads to better thread generation. Furthermore, the application of standard transformations to heap-allocated structures reduces the number of memory accesses via pointer indirections. In parallel applications, pointer indirections to heap-allocated data structures often refer to memory locations that are remote, especially for distributed-memory machines. In general, these optimizations can lead to a significant reduction in network traffic, and an overall improvement in performance. For the multithreaded program execution model, these optimizations may bring additional benefit by reducing the number of threads. This is accomplished by merging parallel statements detected by the heap analysis into one thread or producing no threads at all if the statements have already been eliminated (or moved out of loops) by some transformations.

In standard compilers, optimizations usually have advantages as well as possible negative side-effects on performance. For example, eliminating common subexpression may reduce the computation time for certain common expressions but it may also increase the register pressure. However, if the same optimization is applied to eliminate common remote accesses for parallel programs it will reduce the number of messages sent across the network, hence lowering the communication cost. Furthermore, reducing remote accesses results in further reduction on the number of threads generated, eliminating unnecessary thread-switching overhead. The saving on the communication time as well as reducing the cost of thread switching for parallel threaded programs is far more than the cost introduced by the register pressure, hence it is more beneficial to apply this optimization to parallel programs than their sequential counterparts.

However, optimizations for parallel programs may not always improve performance. Furthermore, optimizations for parallel programs have many other performance factors to consider such as task distribution, data locality, load balancing, scheduling overhead and other performance issues. Thus, another goal of this paper is to study the effect of the optimizations and to find some general principles for when to apply certain optimizations.

We have integrated the heap analyses and transformations into the EARTH-McCAT C compiler [8, 9], and studied the effect of the optimizations on the performance of a collection of benchmark programs. We provide both static and dynamic measurements to show which optimizations give performance improvement, and we suggest some general guidelines on when to apply different optimizations on multithreaded architectures. Somewhat surprisingly, we found that multithreading can not only hide communication latencies but it also sometimes hides the effect of some of our optimizations.

The remainder of the paper is organized as follows. In Section 2, we summarize the EARTH-C compiler environment, especially the thread generation strategies and the heap analysis techniques. In Section 3, we present the optimizations which are useful for thread generation. In Section 4, we analyze our experimental data on a suite of irregular applications. Finally, related work is reviewed in Section 5 and future work and conclusions are given in Section 6.

2 EARTH-C Compiler Environment

EARTH-C is a parallel C dialect with simple extensions to express: *control parallelism, data locality*, and *collective communication*. The EARTH-C compiler takes either C or EARTH-C programs as input, performs thread partitioning based on data-flow analyses such as side-effect (read-write) analysis and heap dependence analysis provided by the McCAT compiler [8], and generates the Threaded-C code as output [10, 14]. The Threaded-C code contains lower-level thread primitives. The final executable code, which runs on the MANNA parallel architecture [2], is compiled by the Portland `pgcc` compiler by linking with the EARTH threaded runtime library. The component structure of the EARTH-C compiler is shown in Figure 1. In this section, we introduce the three parts most related to this paper: the EARTH-C language, heap analysis, and thread generation.

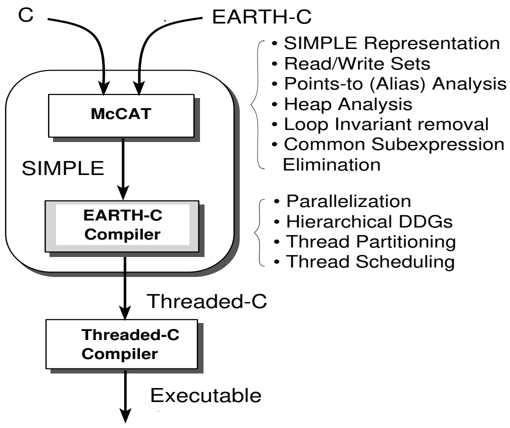

Figure 1: EARTH-C compiler environment

2.1 EARTH-C Language

In general, the EARTH-C language was designed so that a programmer can take an ordinary C program, and the default assumptions will lead to a correct, although possibly inefficient, low-level threaded program. The programmer can then add information such as coarse-grain parallelism and data locality to the program to enhance performance. A detailed description of the EARTH-C language has been given previously [9], and a very brief summary is given here.

{^ $S_1; S_2; \ldots; S_n$ ^}	forall(...) { body }
(a) Parallel Statement Sequence	(b) Forall Loop

Figure 2: Parallel Structures in EARTH-C

There are three kinds of statement blocks in EARTH-C, which can represent either an *ordinary*, *parallel*, or *strictly sequential* statement sequence. In addition, there is a general type of `forall` loop that can express loop parallelism. Figure 2(a) represents a parallel statement sequence in which statement $S_1, S_2, \ldots, S_n$ can be executed in any order, or in parallel. On the contrary, Figure 3(a) is a *strictly-sequential* statement sequence in which the sequential execution order of statements must be obeyed. Figure 3(b) shows an *ordinary* C statement sequence in which the execution order of statements can be reordered with respect to the data dependencies. An abstract parallel loop is listed in Figure 2(b), in which each instance of loop body can be run in parallel.

{! $S_1; S_2; \ldots; S_n$!}	{ $S_1; S_2; \ldots; S_n$ }
(a) Strictly Sequential	(b) Ordinary

Figure 3: Sequential Structures in EARTH-C

In addition to the ordinary C data types, EARTH-C provides *global variables*, *replicated global variables*, *shared variables*, and *local pointers*. *Global variables* have only one copy in the whole system. *Replicated globals* have one copy of the global variables per processor, and no consistency in maintained for those copies. Thus, updating a replicated global updates the copy on the current processor. *Shared variables* are provided for parallel computations that need to access the same copy of a variable. Shared variables must be accessed via atomic functions, thus the atomic access of these *shared variables* is guaranteed. *Local pointers* are used to specify when pointer indirections must point to local memory. In EARTH-C, pointer indirections are assumed to be references to remote memory, unless the pointer has been explicitly declared to be a local pointer. The general rule is that two computations can be executed in parallel if they interfere only on replicated and shared variables. If the computations interfere on other variables, then it is not a valid EARTH-C program.

EARTH-C supports three types of functions: *remote*, *basic* and *atomic*. A function declared in the usual manner in C is by default a *remote* function, it can be invoked on any processor, and it may contain remote memory accesses. In contrast, a function which is explicitly declared with the keyword `basic` is a C sequential function, and it will always be invoked on the stack of the current processor, and it may not access any remote memory, nor call any remote functions. Finally, a function explicitly declared with the keyword `atomic`, operates on a shared variable that is given as the first parameter, and it must not refer to remote memory, nor call any remote functions.

2.2 Heap Analysis

To accurately disambiguate memory references via pointer indirections, our compiler presently uses two pointer analyses: (i) points-to analysis [5] and connection analysis [6]. Points-to analysis computes points-to pairs of the form *(ptr, target)*. It accurately estimates the set of possible target locations for pointers that point to named memory locations (typically allocated on the stack). However, for heap-directed pointers it just provides the conservative information that they point to a symbolic location called *heap*.

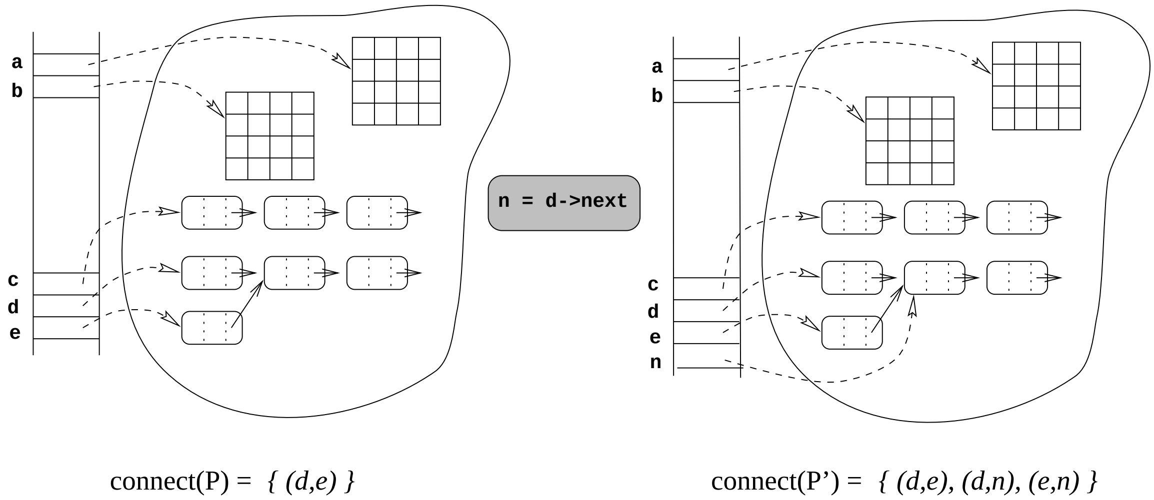

Figure 4: Snapshots and related connection information

As our study focuses on applications which primarily use heap-allocated data structures, we need more accurate information for heap pointers. To this end, we perform a simple heap analysis called connection analysis. Connection analysis computes connection relationships between heap pointers at each program point. Two heap pointers are *connected* if they possibly point to objects belonging to the same heap data structure. They are *not connected* if they *definitely* point to objects belonging to disjoint data structures. For example, in Figure 4 at program point P (left hand side), only pointers `d` and `e` point to the same data structure, giving the connection set, Connect(P) $= \{(d, e)\}$. After executing the statement `n = d->next`, `n` also points to the same data structure as `d`, giving the new connection set, Connect(P') $= \{(d, e), (n, d), (n, e)\}$.

Note that it is the *negative connection information* that is really useful. For example, in Figure 4 the pair (a, b) is not in the set of connection relationships, so any operation on array `a` must be distinct from any operation on array `b`. Similarly any access to the list pointed to by `c` must be disjoint from any access to the list pointed to by `d`. Complete details about connection analysis can be found in [6].

```
typedef struct point
{ double x;
  double y;
  struct point *next;
} Point;

Point pointA, pointB;
```

```
void main()
{
  Point *s, *t;
  s = alloc_point();
  t = alloc_point();
  sum(&pointA, &pointB);
  flip(s, t);
}
```

```
void flip(Point *a, Point *b)
/* a gets the reflection of b */
{
  S: a->x = b->y;
  R: /* tmp = a; a = b; b = tmp; */
  T: a->y = b->x;
}

void sum(Point *c, Point *d)
/* a gets the reflection of b */
{
  U: c->x = c->x + d->y;
  V: c->y = c->y + d->x;
}
```

Figure 5: Example for Read-Write Sets

In order to enable optimizing transformations to treat both direct and indirect memory references, our compiler uses a read-write set analysis, which computes the sets of variables read and written by each statement. Since alias (points-to) information can expose memory locations dereferenced by pointer variables in a statement, encoding such information into read-write sets enhances the accuracy of this analysis. For example in Figure 5, inside function `sum` we have the points-to information *(c, pointA), (d, pointB)*, this leads to the read-write sets for statements **U** and **V**:

Read(U)=(c,d,pointA.x, pointB.y), Write(U)=(pointA.x)
Read(V)=(c,d,pointA.y, pointB.x), Write(U)=(pointA.y)

Inside function `flip`, both pointers are heap-directed. The points-to information *(a, heap), (b, heap)* gives very conservative read-write sets:

$$Read(S) = Write(S) = Read(T) = Write(T) = \{heap\}$$

To get sharper read-write information in this case, we can use connection information (Connect(S) = Connect(T) ={}). However, encoding of connection information into read-write sets is not so obvious. For example, one cannot simply say that Write(S) = (`a->x`) and Read(T) = (`b->x`), and conclude that there is no flow dependence from `S` to `T` as pointers `a` and `b` are *not connected* at statement `S` as well as at `T`.

Consider the case when statement `R` in function `flip` is uncommented and pointers `a` and `b` get swapped. The connection sets remain unchanged, but the references `a->x` at `S` and `b->x` at `T` now access the same memory location. Thus even if `a` and `b` are not connected (i.e. disjoint) at both `S` and `T`, the incarnation of pointer `a` at `S` can still have the same target as that of pointer `b` at `T`. Thus we need to capture connection relationships between specific incarnations of heap pointers.

To this end, we introduce special new names, *anchor handles*, of the form `ptr@Stmt` which capture the connection relationships of a pointer at a given statement [7]. These anchor pointers are included in connection analysis by introducing ghost copy assignments of the form `ptr@Stmt = ptr` while analyzing the given statement. With this modification the connection sets (C) for function `flip` with statement `R` uncommented, are as follows:

$$\begin{aligned} C(S) &= \{(\mathrm{a,a@S}),(\mathrm{b,b@S})\ \} \\ C(T) &= \{(\mathrm{a,a@T}),(\mathrm{b,b@T}),(\mathrm{b,a@S}), \\ &\quad\ (\mathrm{a,b@S}),(\mathrm{a@S,b@T}),(\mathrm{b@S,a@T})\} \end{aligned}$$

We can now infer correct dependence information based on the fact that `a@S` is connected with `b@T` at statement `T`. Further, using anchor handles, we can encode read-write set information for heap pointers in the same fashion as for stack pointers, as follows:

Read(S)={`b@S->y`}, Write(S)={`a@S->x`}
Read(T)={`b@T->x,a@S->x`}, Write(T)={`a@T->y,b@S->y`}

Once we have computed both stack and heap read-write sets, it is straightforward to provide accurate dependence information even for statements involving stack or heap pointers. Similarly, we can include pointer expressions in standard compiler optimizations like common subexpression elimination and loop invariant removal. In section 3 we discuss the impact of better pointer dependence information on generating more efficient threaded programs. Since the focus of this paper is on heap-based programs, in other sections we use more specific terms like heap analysis and heap dependence tests interchangeably with pointer analysis and pointer dependence tests.

2.3 Thread Generation in the EARTH-C compiler

As shown in Figure 6, the input to the EARTH-C thread generator is a SIMPLE AST representation [21]. In SIMPLE, complicated expressions have been simplified into three-address format while the high-level control structures are still retained. For the EARTH-C compiler, the SIMPLE representation is further simplified so that each assignment statement contains at most one *remote* variable. A *remote* variable is a variable which may access memory of another processor. For instance, global variables, and references through non-local pointers are all remote variables. This simplification process is required because only one remote variable is allowed in the low-level EARTH remote Read/Write operations.

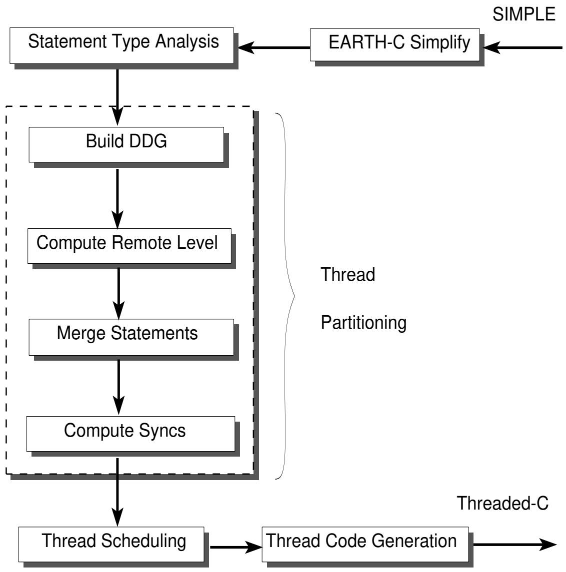

Figure 6: Analyses and Transformations in the EARTH-C compiler

After simplification, the EARTH-C statement analysis assigns a type for each statement. For the purpose of this paper, we assume there are only two types: *local* and *remote*. If a statement contains a remote variable, then it is a remote statement; otherwise, it is a local statement. The EARTH-C statement type is used in thread partitioning and scheduling.

The EARTH-C thread generator relies on a data dependence graph (DDG) to partition threads. DDGs for a program are built in a hierarchical fashion. Compound statements like conditionals or loops are treated as one node within their own hierarchical level. Going into a compound statement, another layer of DDG is exposed. The more precise the DDG, the more parallelism may be revealed, and thus fewer constraints are placed on the thread generator.

More accurate heap analysis can be used to build more precise DDGs, and this can lead to better thread generation.

Once a DDG is built, the thread partitioner first computes the *remote* level for every node in the DDG[1]. The remote level of a node is the maximum number of remote nodes along any path from the root of the DDG to the node itself. The remote level of a node represents the minimum number of thread boundaries from the root to the current node. Merging nodes with the same remote level into one thread is the partition strategy used in the EARTH-C compiler [9]. Even though other advanced thread partitioning algorithms have also been studied [22], we found that when the size of DDG is small (less than 50), this remote-level-based thread partitioning algorithm achieves competitive partitions as compared to other advanced partition algorithms.

Heap analysis can help building more precise DDGs so that two independent dereferences through pointers may be assigned the same remote level, and thus merged into one thread. One of the advantages of such merging is to make remote accesses pipelined.

The last step in thread partitioning is to compute the synchronization necessary among threads. We use a very simple strategy: the number of synchronization signals of next thread is the number of remote statements of its previous threads.

After threads have been formed, the thread scheduler rearranges the execution order within a thread. The most important heuristic used in the EARTH-C compiler is: hoisting remote statements as high as possible within a hierarchical block. In doing so, we can not only pipeline remote operations but also provide an opportunity for local statements to cover remote operations. More precise dependence tests can lead to more opportunities for hoisting remote statements.

Finally, the statements marked as remote are translated to lower-level *split-phased* Read (Write) operations and thread boundaries are created by inserting the thread creation and synchronization primitives.

3 Thread Partitioning and Optimizing Transformations

In this section we study how the connection-based heap dependence information enables generation of more efficient threaded code. In section 3.1 we describe some scalar optimizations on heap expressions which can reduce the number of remote data accesses for the threaded program. In section 3.2 we describe how the thread generator itself benefits from improved heap dependence information.

[1] The *remote* level is also called the *earliest thread number* in [9]

3.1 Heap-based Optimizations

Since every indirect memory access translates into a remote access for threaded programs (unless the dereferenced pointer is explicitly declared as `local`), we focused on optimizations that can eliminate redundant indirect references. To this end, we extended two very useful scalar optimizations, loop invariant removal (LIR) and common subexpression elimination (CSE), to heap expressions in programs. We also implemented another optimization we call *location invariant removal* (LIR-1). Any pointer reference that accesses the same memory location in all iterations of a loop is considered to be *location invariant*. A location invariant access can be replaced by a scalar access, under some constraints. This optimization is similar in spirit to the scalar replacement technique proposed for array references [3]. Consider the loop in Figure 7(a). In this loop the pointer reference `r->item` is location invariant as the pointer `r` is not written inside the loop. We can replace all accesses to `r->item` with a scalar, say `temp_0`, by initializing `temp_0` as `r->item` and then assigning it to `r->item` outside the loop (Figure 7(b)). However, we also need to check that the reference `p->item` does not access the same location as the reference `r->item` in any iteration (for example when `r` and `p` are not connected this situation will not arise). Detailed algorithms for *safely* performing these optimizations using stack and heap read-write sets, are presented in [7].

```
while (p != NULL)
{ S: r->item = r->item + p->item;
  T: p = p->next;
}
```
(a)

```
temp_0 = r->item;
while (p != NULL)
{ S: temp_0 = temp_0 + p->item;
  T: p = p->next;
}
r->item = temp_0;
```
(b)

Figure 7: Location Invariants

Opportunities for the above optimizations frequently arise in heap-based programs: typically for address calculations of heap-embedded arrays, and due to the frequent use of macros to access heap data. In Figure 8, we show an example from our benchmarks to demonstrate this. It shows a loop from the *health* benchmark, where both loop and location invariants are found (annotated as LIR and LIR-1). The location invariant reference is a scalar structure field which is conditionally updated in every loop iteration.

3.2 Thread Partitioning

A more precise heap analysis helps the thread generator to do some optimizations during the thread partition phase. In Figure 9, we demonstrate three major optimizations: (i) *merging and*

```
/* LIR-1 */
invar_17 = (*village).hosp.free_personnel;
/* LIR */
invar_18 = (&(*village).hosp.waiting);
while ( (list != 0) )
{ i = invar_17; p = (*list).patient;
  if ((i > 0))
  { t = invar_17;       /* LIR-1 */
    invar_17 = (t - 1); /* LIR-1 */
    l = invar_18;       /* LIR   */
    L_removeList(l, p);
    /* ... statements deleted */
  }
  list = (*list).forward;
}
/* LIR-1 */
(*village).hosp.free_personnel = invar_17;
```

Figure 8: Code Segment Using the Optimizations

pipelining remote accesses, (ii) *exploiting function-level parallelism*, and (iii) *reducing remote messages*. The EARTH-C fragments are listed in the upper part (first and third rows) in Figure 9, and the corresponding generated Threaded-C code in the lower part (second and fourth rows), illustrating the optimization effect. Note that `GET_SYNC` and `DATA_SYNC` are the corresponding low-level EARTH read and write primitives. They are executed in a split-phase manner, and such a split-phase constraint requires that issuing a remote operation and accessing the value returned must be separated in two different threads. `END_THREAD` is an EARTH primitive which ends the current thread and switches the control to another ready thread.

<table>
<tr><td><pre>
  y = 1;
  p->x = a;
  q->x = b;
</pre></td><td><pre>
  x = 1;
  f(p->next);
  g(q->next);
  h(x);
</pre></td></tr>
<tr><td><pre>
  DATA_SYNC(a,&p->x);
  DATA_SYNC(b,&q->x);
  y = 1;
  END_THREAD();
THREAD_n:
</pre>(a) Pipelining</td><td><pre>
  TOKEN(f,p->next);
  TOKEN(g,q->next);
  x = 1;
  h(x);
  END_THREAD();
THREAD_n:
</pre>(b) Parallelizing</td></tr>
<tr><td><pre>
  a = p->x;
  b = p->x;
</pre></td><td><pre>
  t = p->x;
  a = t;
  b = t;
</pre></td></tr>
<tr><td><pre>
  GET_SYNC(&p->x,&a);
  GET_SYNC(&p->x,&b);
  END_THREAD();
THREAD_n:
</pre></td><td><pre>
  GET_SYNC(&p->x,&t);
  END_THREAD();
THREAD_n:
  a = t;
  b = t;
</pre></td></tr>
<tr><td colspan="2">(c) Reducing Messages</td></tr>
</table>

Figure 9: Useful Optimizations for Thread Generation

First, the goal of *merging and pipelining remote accesses* is to group remote operations into one thread. As Figure 9(a) shows, without a precise DDG, the EARTH-C compiler has to assume that both the statements write to the *heap* and there is a possible output dependence between `p->x` and `q->x`. However, with more precise dependence information from connection analysis, we can determine that `p` and `q` point to disjoint memory locations, and thus there is no output dependence between the statements. Thus, the two statements can be merged into one thread, as shown in the second row of Figure 9(a). The advantages of this optimization are three-fold: (1) reducing the thread switching cost by reducing the number of threads generated; (2) allowing the thread generator to hoist remote operations up and issue them earlier to cover latencies; (3) merging remote operations together to pipeline the issuing of remote operations. In the lower part of Figure 9(a), two remote `DATA_SYNC` operations are grouped into one thread, moved to the beginning of the thread, and their execution is pipelined. Further, the *local* statement $y = 1$ is scheduled to be executed after the issuing of two remote statements to cover latencies. Although the effect of overlapping such a small local statement is minimal, this technique can also be used to overlap larger local computations such as local loops, and calls to basic functions.

Second, the goal of *exploiting function-level parallelism* is to expose parallel function calls. Given the program fragment in Figure 9(b), assume functions f and g are *remote*, function h is *local*, and h is independent of the other two calls. We may be able to determine that the two remote function calls can be executed in parallel if connection information says that p and q point to disjoint data structures. Once the thread generator decides these remote function calls can be run in parallel, they will be merged into one thread and issued with the EARTH `TOKEN` primitive in pipeline and much earlier, as shown in the lower part of Figure 9(b). These tokened function invocations are mapped to a processor by the runtime load balancer. If some processors are really idle, the newly exposed function-level parallelism may lead to a better load balance by exploiting the idle processors. As the second row of Figure 9(b) shows, two parallel functions are grouped into one thread. They are started in parallel with the `TOKEN` primitive and issued earlier then the local function h. Clearly, latencies caused by the remote functions will be covered by the local function h and local statement $x = 1$.

However, there can be some negative side-effects of this transformation. The processors on which the newly discovered parallel functions will run are decided by the runtime load balancer. If a function instance is invoked at a processor which does not own much of the data the function needed, we risk losing data locality for the function-level parallelism. Further, if no processor is idle at that moment, the newly discovered parallelism may not be used at all.

Third, the goal of *reducing the number of messages* is to eliminate remote statements. When the CSE and LIR optimizations can be applied, they may eliminate extra remote statements and thus reduce the number of messages sent across the network. In Figure 9(c), the left column gives the original program before optimization, the right column shows the program after optimization. After the CSE optimization is applied, the number of remote accesses is reduced from 2 to 1 because one extra `GET_SYNC` statement is eliminated. This optimization may also increase data locality since it is highly likely that when t is read second time it is already in cache. The reduction in communication becomes even bigger if a remote access is loop/location invariant and it has been moved out of the loop body.

4 Experimental Results and Analysis

We have implemented the heap analysis based optimizations in the EARTH-C compiler. The main transformations are: *thread merging, remote operation hoisting and pipelining, common sub-expression elimination*, and *location and loop invariant removal.* We report our experiments on four irregular benchmarks using the multithreaded EARTH-MANNA machine. Our major experimental results are summarized below.

First, static program measurements indicate that our optimizations have a significant effect. The average degree of DDGs is reduced modestly and the average thread length is increased dramatically. Further, We find significant opportunities for applying CSE, and LIR to heap references (see Section 4.2).

Second, the optimizing transformations show a significant speedup over the unoptimized programs for certain benchmarks. Among all optimizations, the heap-based common sub-expression elimination and loop invariant removal play a major role in enhancing performance. (see Section 4.3).

Third, there are many other parallel performance factors that may affect the optimization effect. Data locality, parallelism exploitation, load balancing, communication cost to name a few. A significant static improvement may lead to only a modest improvement in execution time of the threaded program, and we may not have the same performance improvement at runtime as the improvement on static program features (see Section 4.4).

Fourth, multithreading can hide the communication latencies as well as the optimization effect on reducing communication messages (see discussions in 4.4.1).

In the following, we first introduce the benchmarks used, then describe the multithreaded emulator: EARTH-MANNA. After listing both *static* and *runtime* statistics of threaded programs, we analyze the optimization effects on performance as well as examine the influence of other parallel performance factors on optimizations.

4.1 Benchmarks and EARTH-Manna system

We have implemented four Olden [4, 15] benchmarks: *health*, *em3d*, *perimeter*, and *mst* to test the effect of our optimizations.

Health simulates the Colombian health-care system using a 4-way tree [12]. 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. The 4-way tree is distributed as evenly as possible among the processors for certain levels.

Em3d is a computational problem based on an irregular bipartite graph containing nodes representing electric and magnetic field values (E nodes and H nodes, respectively) [13]. At each time step, new values for the E nodes are computed from a weighted sum of the neighboring H nodes, and then the same is done for the H nodes. The main computation loop consists of each processor walking down its list of nodes, reading the values from neighbors and using them to update the current node.

Perimeter computes the perimeter of a quad-tree encoded raster image [18]. 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.

Mst computes the minimum spanning tree of a graph using Bentley's algorithm [1]. The vertices of the graph are divided evenly among the processors. At each step, each processor nominates one vertex to be added to the spanning tree, and the closest vertex to the partial spanning tree is selected. This process repeats until all vertices have been searched. One of the main data structures used is the hash table which stores adjacent edges for each node.

Benchmark	Main Data Structure	Parameters
health	4-way tree	6 levels and 100 iterations
em3d	adjacent list	32K nodes and 50% of neighbors are remote
perimeter	quad-tree	Maximum 11 levels
mst	hash table and linked list	512 nodes

Table 1: Data Structure and Running Parameters

We have implemented a "naive" version of these four benchmarks in EARTH-C. In the "naive" version, we mainly expose the parallelism with the EARTH-C parallel constructs to the EARTH-C compiler, and make some attempts to map computation to the processor owning the data. We do not fine-tune the program for the best locality. Thus, these benchmarks reflect the kind of programs that are easy for the programmer to develop.

Table 1 lists the main data structures and the running parameters for each benchmark. Because the input to *mst* is a complete graph, the graph size is limited by the memory capacity of each processor node. In each run, we measure the total program execution time including the tree (graph) building phase. Thus, the runtime improvement we achieved is for the whole program instead of some "kernel".

Our experiments have been done using a multithreaded emulator built on top of the MANNA parallel machine [2]. Each MANNA node consists of two Intel i860 XP CPUs, clocked at 50MHz, 32MB of dynamic RAM and a bidirectional network interface capable of transferring 50MB/s in each direction. The two processors on each node are mapped to the EARTH Execution Unit (EU) and Synchronization Unit (SU). The EARTH EU executes active threads while the SU takes care of thread scheduling, thread synchronization, and remote communication. The EARTH runtime system supports efficient remote operations. Sequentially, loading a remote word takes about 7μs, calling a remote function can be performed in 9μs, and spawning a new remote thread takes about 4μs. When issued in a pipeline these operation take only one third of these times. Thus, placing many independent remote operations into one thread and issuing them in a pipeline may reduce the total execution time. Moreover, reducing remote operations can further eliminate significant overhead incurred.

4.2 Static Program Measurements

In this subsection we evaluate the static properties of the threaded code generated after applying our optimizing transformations. The following set of static program statistics have been collected:

Average Degree of DDG (ADD): For each DDG in the program, we calculate its average degree as the number of edges divided by the number of nodes. The ADD is simply the *mean* of the average degrees of all DDGs within a program.

Average Thread Length (ATL): A thread is *statically* defined as starting from the beginning of a hierarchical block or starting immediately after an END_THREAD, and continuing until the next END_THREAD or end of a block. Please note that because conditional and compound statements are allowed within one thread, static ATL may be different from runtime ATL. However, static ATL can be used as an indicator of thread size.

Maximal Thread Length (MTL): This is simply the largest thread length found over the whole program.

Remote Statement Ratio (RSR): To calculate the remote statement ratio for a DDG, we divide the number of remote statements by the total number of statements. The RSR for a program is the *mean* of the RSRs for all DDGs within the program.

The average thread length (ATL) for a program is the *mean* of the thread lengths of all threads within the program. Since static ATL is computed hierarchically by the level-by-level method, sometimes it does not accurately reflect the real improvement of optimizations, particularly when loop invariant removal drastically changes the structure of the program. For example consider the program in Figure 10.

In Figure 10(a), there are two threads at the hierarchical level of the loop body, the first containing $N-1$ local statements, and a remote statement; and the second containing N local statements. The ATL for the loop body is $((N-1+1)+N)/2 = N$. However, assume the

(a). Before	(b). After

<pre>for (i=0; i<10; i++) {
  N-1 local statements;
  remote-stmt;
  END_THREAD ;
THREAD_1:
  N local statement;
}</pre>
	<pre> remote statement;
	  END_THREAD;
	THREAD_1:
	  for (i=0; i<10; i++) {
	    N-1 local statements ;
	    N local statements ;
	  }</pre>

Figure 10: Optimization Effect on Thread Length

remote statement is a loop invariant and is moved out of loop body, as illustrated in Figure 10(b). Then at the hierarchical level of the *for* loop, there are only two statements: the remote statement and the loop, and each of them composes one thread itself. Thus, ATL for this level is: $(1+1)/2 = 1$. Therefore, our current counting method on ATL may sometimes not accurately reflect the real effect of the optimizations.

Heap (connection) analysis gives more precise data dependence information for pointer variables. We expect that this analysis will reduce the average degree of DDGs (ADD) and thus provide more opportunity to merge parallel statements into a thread, therefore increasing the average thread length and reducing the total number of threads generated by the compiler. Moreover, common sub-expression elimination and loop invariant removal transformations may eliminate some remote statements, thus further reducing the number of threads generated. At the same time, fewer remote statements should reduce the remote statement ratio (RSR). The static measurements for our benchmarks are given in Table 2 and 3.

Benchmark	Versions	ADD	ATH	MTL	RSR
health	before	1.45	1.67	128	0.52
	after	1.33	2.17	97	0.47
	Imp(%)	8.28	29.94	24.22	9.62
em3d	before	1.70	3.56	99	0.36
	after	1.67	5.02	87	0.35
	Imp(%)	1.76	41.01	12.12	2.78
perimeter	before	1.40	1.83	77	0.35
	after	1.32	2.10	55	0.34
	Imp(%)	5.71	14.75	28.57	2.86
mst	before	1.23	3.08	33	0.32
	after	1.19	2.94	29	0.33
	Imp(%)	3.25	-4.55	12.12	-3.13

Table 2: Program Feature Statistics

Table 2 lists program feature statistics for two versions of programs: *before* and *after* optimizations are applied. The *Imp* rows show the percentage improvement in these program statistics of optimized code over un-optimized one. As shown in Table 2, except for *mst* benchmark, other three benchmarks have positive improvement on various program statistics. The negative numbers for *mst* benchmark are due to the fact that some remote loop invariants are moved out of loops, and so our measurement is not a true reflection of the improvement to the program.

From Table 2, we can see that the optimizations have a significant impact on the static program features. The largest improvements are seen in the *health* and *em3d* benchmarks. For *health*, the average DDG degree is reduced by 8.28% and the average remote ratio by 9.62%. For *em3d*, the average thread length is increased by 41.01%. The heap analysis and optimizations have a significant impact on the threaded code generated.

Benchmark	CSE	LIR	LIR-1	Sum	PAR
health	16	29	20	65	N
em3d	22	32	0	54	N
perimeter	9	0	0	9	Y
mst	10	8	2	20	N

Table 3: Optimization Statistics

Table 3 gives statistics on optimizations themselves. The *CSE* column shows the number of common sub-expressions eliminated. The column labeled *LIR* lists the number of loop invariants moved out of loop bodies, and the column labeled *LIR-1* gives the number of location invariants. The *Sum* column is the sum of all optimizations applied. The last column *PAR* shows whether or not new function-level parallelism is detected by the heap analysis. Among four benchmarks, this type of parallelism is detected only for the *perimeter* benchmark.

4.3 Runtime Performance

Statically, there was a significant improvement on the threaded code after the optimizations have been applied. Can these static numbers translate directly into the improvement on runtime performance? Our experiments show that the optimizations will generally improve execution performance. However, runtime performance can only partially match the static improvement measurements.

Table 4 lists the percentage improvement in optimized program execution time over unoptimized program, running on 1, 2, 4, 8, and 16 processors. We use the relative improvement rate, r_p, to measure the execution-time improvement on p processors. Let T_p^{unopt} and T_p^{opt} be the total execution times for un-optimized and optimized programs respectively. Then, r_p is defined as:

$$r_p = (T_p^{unopt} - T_p^{opt})/T_p^{unopt}$$

The bigger r_p is, the more effect on execution time the optimizations have. Thus, r_p reflects the execution-time improvement on p processors.

We also list the individual optimization effects for r_p in Table 4. The *-O* notation is used to represent optimization level in our experiment. The optimization levels are as follows:

O0 level optimizations use the heap analysis to improve thread partitioning. The transformations include: exposing more parallelism, merging parallel statements into one thread, hoisting remote statements earlier in threads, and pipelining remote operations.

O1 level optimizations apply common sub-expression elimination (CSE), copy propagation, loop invariant removal (LIR), and location invariant removal.

O2 means enabling all optimizations, i.e., *O0* $+$*O1*.

Overall, common sub-expression elimination and loop invariant removal dominate the optimization effect on execution time. For example, the optimization effect to *mst* comes almost completely from the *O1* optimization. Revealing more function-level parallelism by the *O0* optimization may deteriorate performance. For example, with 1 or 4 processors, the *perimeter* performance is slightly reduced by the *O0* optimizations.

We also notice, as with many compiler optimizations, the overall optimization effect may not be the sum of the individual effects. For example, r_{16} for perimeter with *O0* and *O1* optimization levels are 4.54% and 7.62% respectively. However, r_{16} with *O2* is only 8.39%, which is almost 4% less than the sum of two r_{16} numbers for *O0* and *O1*.

Benchmark	$(-O)$	r_p(%) for different nodes				
		1	2	4	8	16
health	0	0.01	0.07	0.16	0.47	0.83
	1	1.00	1.03	1.00	1.15	0.87
	2	1.01	1.08	1.11	1.36	1.54
em3d	0	4.98	3.76	2.86	2.99	1.57
	1	24.03	21.30	21.62	19.79	15.17
	2	26.40	23.33	23.28	21.37	15.93
perimeter	0	-0.03	0.06	-0.52	1.83	4.54
	1	1.64	1.64	4.05	4.42	7.62
	2	1.61	1.72	3.15	5.60	8.39
mst	0	0.01	0.01	0.02	0.06	0.13
	1	6.34	6.22	6.27	7.27	7.60
	2	6.35	6.24	6.30	7.33	7.73

Table 4: Timing Improvement After Optimizations

For the overall performance, we see a varied range of improvement rates: from slight increase 1.54% to significant 15.93% (16-processors). In the following, we report the reasons for either improvement or deterioration. The detailed performance analysis is given in the next section.

Health: The optimizations have little impact on *health*. The reason is that *health* benchmark has high data locality and thus there is fairly little room for further improvement with our optimizations. Even so, we see a small improvement, and this improvement increases as more processors are used.

Em3d: This benchmark is an example where r_p decreases as more processors are used. This "diminishing" effect is because it is a communication intensive benchmark where multithreading really seems to work. This is because even though communication cost increases as more processors are used, multithreading appears to hide communication latencies, and thus shadow the optimization effect. A more detailed analysis of this effect is given in Section 4.4.1.

Perimeter: The *perimeter* benchmark demonstrates the impact of revealing more function-level parallelism. Table 3 tells us that heap analysis detects more function-level parallelism for *perimeter*. However when a small number of processors are used, we only see very small r_p

with *O2* optimizations, and even negative numbers with *O0* optimizations. The reason is that abundant parallelism may be detrimental to performance when only limited processor resources are available. A large number of processors are needed to cope with the newly discovered parallelism. That is why we see r_p increasing when more processors are used.

Mst: The optimizations have modest impact on *mst*, and we see a modest improvement rate for *mst*, 7.73% for 16-processors. Since many remote loop invariants are moved out of loop-bodies, we might expect to see a significant improvement on runtime performance. However, *mst* also exploits data locality well which makes the potential optimization effect less visible.

Table 5 and 6 list two runtime statistics for the optimizations: the percentage reduction of the number of of remote Read-Write operations and total threads executed. In order to collect these figures we instrumented the generated threaded-C programs to count the number of remote read/writes and the number of threads executed. We report the improvement of the optimized version over the unoptimized version.

The percentage of the improvement in reduced remote Read-Write operations is defined as:

$$Imp_{rw} = (RW_{before} - RW_{after})/RW_{before}$$

RW_{before} and RW_{after} are the number of remote Read-Write operations executed for two version of threaded programs, *before* and *after* optimizations are applied. Imp_{rw} represents the reduction in reducing the communication cost.

The percentage of the reduction in total number of threads is defined as:

$$Imp_{th} = (TH_{before} - TH_{after})/TH_{before}$$

TH_{before} and TH_{after} are the total number of threads executed respectively. Imp_{th} represents saving in the thread switching cost.

Benchmark	Imp_{rw}				
	1	2	4	8	16
health	0.00	0.69	0.69	0.71	0.71
em3d	32.67	32.67	32.67	32.67	32.67
perimeter	3.17	3.17	3.17	3.17	3.17
mst	98.82	98.43	97.67	96.17	93.28

Table 5: Improvement (%) on Remote Read-Write

Table 5 gives the improvement of the optimized programs in the number of remote Read-Write operations over un-optimized ones, collected at runtime. This gives a good estimate of improvement in communication time for the optimized versions. As explained before, the optimizations have little impact on *health* benchmark because it has high data locality, and there are very few opportunities for further optimization. On the other hand, the optimization has the greatest impact on *mst* benchmark. The number of remote Read-Write operations is reduced by over 93%. The main reason is that the remote operations eliminated are loop invariants in key loops. The *perimeter* benchmark has shown a moderate reduction in communication since

Benchmark	Imp_{th}				
	1	2	4	8	16
health	0.00	4.93	4.93	5.05	5.05
em3d	38.63	38.63	38.63	38.63	38.62
perimeter	1.67	1.67	1.67	1.68	1.68
mst	99.41	99.02	98.25	96.75	93.87

Table 6: Improvement (%) on Threads Generated

there are no loop invariants detected in this benchmark. The *em3d* benchmark has a significant reduction in communication because it is a communication intensive benchmark.

Table 6 lists the reduction achieved in total number of threads executed at runtime. Because these numbers reflect the saving on thread generated, they also represent the saving on thread switching cost. *Perimeter* has the least reduction because the number of optimizations applied is the least in Table 2; *mst* has the greatest due to the LIR optimization; *health* has a modest reduction because high data locality shadows the effect of optimizations; and *em3d* has a significant reduction in thread switching cost because the number of optimizations applied is the second largest in Table 2.

However, other parallel performance factors may influence the effect of the optimizations. We have already seen the modest improvement in execution time even though there are relatively large reductions in communication and thread switching cost.

4.4 Performance Analysis

In the previous sections we have seen that our optimizations can lead to relatively large reductions in the total number of remote Read-Write operations, and we can reduce the number of threads, thus reducing the overhead of thread switching. However, these improvements do not necessarily predict the overall improvement in execution time. We may see only moderate improvement in execution time, even in the presence of substantial optimizations. We believe that there are four major reasons for this.

First, good data locality may provide little room for further optimization. If a program exploits data locality well, optimizations in reducing remote operations becomes less important. For example, the *health* benchmark has high data locality. In *health*, there are only $(4^3-1)/3$ remote villages on 16 processors, compared to $(4^7-1)/3$ total villages. References to villages that happen to reside on a local processor will be quite efficient, and thus optimizing them is not so important. Thus, saving on visiting remote villages will be hidden by the total visiting time on the whole village tree. This results in little effect of the optimizations on program performance.

Another example is in *mst* benchmark. With the LIR optimizations, there is a tremendous saving in remote communication, 93.87% on 16 processors (see Table 5). We would expect to see great saving in execution time too. However, this effect is overshadowed by the exploitation of data locality. *Mst* exploits data locality extremely well by using hash tables to store node's adjacent edges. These hash tables behave like cache, reducing the remote read time significantly.

Second, exposing excess parallelism may decrease performance. Once the number of function instances at one processor is greater than one, the runtime load balancer may decide to run some function instances on other processors. This may have two adverse impacts on performance: (1) it may destroy data locality if the load balancer decides to run the function-call instance on a processor which does not own much of needed data; (2) the discovered parallelism will be wasted if there are no processors idle. We observed the negative impact of revealing more function-level parallelism for *perimeter* benchmark, when it is run on a small number of processors.

Third, communication latencies may shadow other optimization effects. As more and more processors are used, the communication latencies become longer as distance between two processors increases, and contention increases too. The saving on the thread switching cost by $O0$ optimizations may be made less significant as the communication cost predominates. *Em3d* is a communication intensive benchmark. In our test case for *em3d*, 50% of a node's neighbors are remote, and thus half of the neighbor visiting time is spent on remote communication. Even though the saving on the thread switching cost is significant, 38.83% on average (see Table 6), the relative improvement on execution time is reduced from 26.40% to 15.93% when the number of processors used is increased from 1 to 16 because the communication cost is also increased.

The above explanation may be good enough for being unable to see the clear effect on the reduction in the number of thread switching. Nevertheless, people wonder why the saving on communication is "diminishing" too for such a communication intensive benchmark? Part of the answer is due to the influence of multithreading. Because multithreaded architectures can hide communication latencies, optimization effect on reducing communication traffic may be shadowed by the multithreading ability to hide latencies.

4.4.1 The Effect of Multithreading

To examine the impact of multithreading further, we experimented with several versions of *em3d*. Tables 7 and 8 show the multithreading impact on optimizations and performance. The number of top-level threads is set to 16, 32, and 64 respectively. These threads are then distributed evenly to processors. We measure how r_p and execution times change with the variation of thread numbers. Table 7 reports r_p with respect to different thread numbers and processor counts. Table 8 gives the percentage of the execution-time improvement compared to the execution times of 16 threads with $O2$ optimizations.

On a single processor node, all threads are actually assigned to one physical processor. Therefore, the more top-level threads we have, the greater overhead is due to thread switching. This results in a more visible optimization effect, r_1 increases from 25.68% to 28.68% when threads are up from 16 to 64. However, the switching overhead slows down the program execution. Thus, we see some negative improvement in execution time for some 32-threads

Threads	$r_p(\%)$				
	1	2	4	8	16
16	25.68	23.17	23.26	21.37	18.8
32	26.40	23.33	23.28	21.37	15.97
64	28.68	26.10	23.34	21.40	15.96

Table 7: r_p changes with the thread and processor counts

and 64-threads cases. The same argument is applied to the cases when 2 and 4 processors are employed.

Threads	Percentage of Time Improvement				
	1	2	4	8	16
16	0.00	0.00	0.00	0.00	0.00
32	-2.35	-0.77	0.03	4.71	10.20
64	-9.61	-9.06	-0.74	5.21	10.95

Table 8: Time Improvement (%) on Em3d

When 16 processors are used, each processor has 1, 2, and 4 top-level threads respectively for the 16-threads, 32-threads and 64-threads cases. The least opportunity for multithreading is the 16-threads case (one thread per processor) and it has the greatest of r_p 18.8% (Figure 7) since optimizations on reducing communication have the biggest impact. Nevertheless, it does not exploit the latency hiding ability of multithreading and its execution time is the longest since other two cases have the positive improvement in execution time against the 16-threads case. On the contrary, when there are multiple threads per processor, such as 32-threads and 64-threads cases, the optimization effects are less visible than the 16-threads case and they are shadowed by the latency hiding ability of multithreading. *em3d* clearly shows that multithreading can hide communication latencies as well as optimization effects on reducing communication traffic.

5 Related Work

Because compiling for multithreading architectures is a relatively new field, research on this subject is limited. For a complete coverage, please refer to [17, 19]. There is relatively little research which has been done on studying the optimization impact on threaded programs, especially the heap-based optimization transformations we studied in this paper. Three of the most related works to this paper are: (1) Krishnamurthy and Yelick's work on using explicit synchronization provided by the programmer to optimize Split-C programs [11]; (2) Roh et. al's work on evaluating the impact of optimizations on threaded code generation [16]; and (3) Sohn et. al's work on identifying the capability of overlapping computation with communication [20].

Krishnamurthy and Yelick use dataflow analysis for synchronization, barriers, and locks to eliminate unnecessary spurious cycles caused by accessing shared variables from different processors. They use more precise ordering information to overlap communication, eliminate redundant messages, and to pipeline messages. Their analysis is restricted to a subset of Split-C, to avoid full blown alias analysis on global pointers, and thread partitioning is done by the programmer. On the contrary, our approach is completely automated, applies to normal C and EARTH-C programs, and it heavily relies on our pointer and heap analyses.

Roh et. al utilize local and global optimizations such as dead code elimination and common subexpression elimination to optimize the code generated from a top-down thread generation strategy. Their experimental results are based on a multithreaded simulator and small kernel loops. In contrast, our benchmarks are based on irregular applications that use dynamically-allocated data structures intensively. Our measurements were collected for the whole program, which runs on a real multithreaded system.

Sohn et. al study the capability of overlapping computation with communication on different distributed-memory machines. Their focus is more on performance analysis than thread partitioning in terms of hiding communication latencies. We emphasize using compiler-based automatic thread-partitioning techniques to hide communication latencies.

In this paper, we use the advanced heap analysis techniques such as connection analysis to optimize the threaded code and study the interactions between high-level transformations and thread generation. Furthermore, we concentrate on irregular applications using dynamically-allocated data structures. These kinds of programs are typically difficult to analyze and transform. We have applied our relatively advanced pointer and heap analyses, and we have analyzed the performance impact of each optimization and show its performance gain or loss for our suite of irregular pointer-intensive based benchmarks, running on a real multithreaded platform.

The heap analysis techniques and optimizations presented in this paper can be applied to any C program. Actually, sequential programs also benefit from these optimizations [7]. The preprocessing and partitioning techniques used in our thread partitioner can also be applied to other *non-preemptive* multithreaded architectures, especially, the transformations designed to take advantage of more accurate heap analysis. We believe that the general trends revealed in our experiment on optimizing communication for multithreaded architectures hold for other similar multithreaded systems.

6 Conclusion and Future Work

In this paper we have described a set of new optimization transformations based on the advanced heap analysis and some thread generation strategies to exploit such an advanced analysis. We have also evaluated the optimization impact on thread generation and runtime performance, and identified some key factors influencing optimization effects. Our experiment shows that optimizations may have a significant impact on performance for certain applications. Positive as well as detrimental optimization effects have been observed in our experiments, and optimization tradeoffs have been studied.

Although we achieve quite reasonable speedups on most applications, our studies show that data locality, excess parallelism, increasing communication cost, and multithreading may overshadow some of the optimization effects. Therefore, we need to develop high-level analyses to capture these performance features and impart this knowledge to the thread generator. In the mean time, the interactions between the thread generator and the influence of the parallel performance factors need to be investigated further and more effective transformations to exploit performance need to be developed.

The current thread partitioning algorithm is based on a very simple cost model, i.e., every remote operation takes "infinite" time and thus thread merging is always profitable. We plan to implement a more sophisticated thread-partition algorithm based on more refined program cost model in the EARTH-C compiler and evaluate its performance in real applications.

Acknowledgements

We would like to thank NSERC (Natural Sciences and Engineering Research Council), Micronet (both in Canada), and NSF-MIPS (NSF-MIPS-9707125) for funding our research. Many people have given assistance in developing the EARTH system, especially Olivier Maquelin, who developed the low-level threaded-C compiler, and the EARTH-MANNA runtime system which makes our experiments possible.

References

[1] J. Bentley. A parallel algorithm for constructing minimum spanning trees. *Journal of Algorithms*, 1, 1980.

[2] 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.

[3] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 53–65, White Plains, New York, June 20–22, 1990. *SIGPLAN Notices,* 25(6), June 1990.

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

[5] 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, Orlando, Florida, June 20–24, 1994. *SIGPLAN Notices,* 29(6), June 1994.

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

[7] Rakesh Ghiya and Laurie J. Hendren. Dropping anchors: Putting heap analysis to work. ACAPS Technical Memo 107, School of Computer Science, McGill University, Montréal, Québec, May 1997.

[8] 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.

[9] 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.

[10] 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.

[11] 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, La Jolla, California, June 18–21, 1995. *SIGPLAN Notices,* 30(6), June 1995.

[12] G. Lomow, J. Cleary, B Unger, and D. West. A performance study of Time Warp. In *Proceedings of the SCS Multiconference on Distributed Simulation*, 1988.

[13] N.K. Madsen. Divergence preserving discrete surface integral methods for maxwell's curl equations using non-orthogonal unstructured grids. Technical Report 92.04, RIACS, February 1992.

[14] Olivier C. Maquelin, Herbert H. J. Hum, and Guang R. Gao. Costs and benefits of multithreading with off-the-shelf RISC processors. In Seif Haridi, Khayri Ali, and Peter Magnusson, editors, *Proceedings of the First International EURO-PAR Conference*, number 966 in Lecture Notes in Computer Science, pages 117–128, Stockholm, Sweden, August 29–31, 1995. Springer-Verlag.

[15] 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.

[16] Lucas Roh, Walid A. Najjar, Bhanu Shankar, and A. P. Wim Böhm. An evaluation of optimized threaded code generation. 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 37–46, Montréal, Québec, August 24–26, 1994. North-Holland Publishing Company.

[17] Lucas J. Roh. *Code Generation, Evaluation, and Optimizations in Multithreaded Executions.* PhD thesis, Computer Science Department, Colorado State University, Fort Collins, Co 80523-1873, 1995.

[18] H. Samet. Computing perimeters of regions in images represented bu quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3(6), November 1981.

[19] Klaus Erik Schauser. *Compiling Lenient Languages for Parallel Asynchronous Execution.* PhD thesis, Department of Electrical Engineering and Computer Science, Division of Computer Science, University of California, Berkeley, California, 1995.

[20] A. Sohn, J. Ku, Y. Kodama, M. Sato, H. Sakane, H.Yamana, S. Sakai, and Y. Yamaguchi. Identifying the capability of overlapping computation with communication. In *Proceedings of the PACT'96*, Boston, Massachusetts, October 1996. North-Holland Publishing Company.

[21] Bhama Sridharan. An analysis framework for the McCAT compiler. Master's thesis, McGill University, Montréal, Québec, September 1992.

[22] Xinan Tang, Jian Wang, Kevin B. Theobald, and Guang R. Gao. Thread partitioning and scheduling based on cost model. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 272–281, Newport, Rhode Island, June 22–25, 1997. SIGACT/SIGARCH and EATCS.