Automatic Compiler Techniques for Thread Coarsening for Multithreaded Architectures*

Gary M. Zoppetti Gagan Agrawal Lori Pollock
Department of Computer and Information Sciences
University of Delaware, Newark DE 19716
{zoppetti,agrawal,pollock}@cis.udel.edu

Jose Nelson Amaral Xinan Tang† Guang Gao
Department of Electrical and Computer Engineering
University of Delaware, Newark DE 19716
{amaral,tang,ggao}@capsl.udel.edu

Abstract

Multithreaded architectures are emerging as an important class of parallel machines. By allowing fast context switching between threads on the same processor, these systems hide communication and synchronization latencies and allow scalable parallelism for dynamic and irregular applications. Thread partitioning is the most important task in compiling high-level languages for multithreaded architectures. Non-preemptive multithreaded architectures, which can be built from off-the-shelf components, require that if a thread issues a potentially remote memory request, then any statement that is dependent upon this request must be in a separate thread.

When performing thread partitioning on codes that use pointer-based recursive data structures, it is often difficult to extract accurate dependence information. As a result, threads of unnecessarily small granularity get generated, which, because of thread switching costs, leads to increased execution time. In this paper, we present three techniques that lead to improved extraction and representation of dependence information in the presence of structured control flow, references through fields of structures, and pointer-based data structures. The benefit of these techniques is the generation of coarser-grained threads and, therefore, decreased execution time. Our experiments were performed using the EARTH-C compiler and the EARTH multithreaded architecture model emulated on both a cluster of Pentium PCs and a distributed memory multiprocessor. On our set of 6 pointer-based programs, these techniques reduced the static number of threads by 38%. Reductions in execution times ranged from 16% to 45% on the four programs we measured runtime performance.

*This research was supported by NSF Grant CCR-9808522. Author Agrawal was also supported in part by NSF CAREER award ACI-9733520.

†Currently affiliated with Chameleon Systems Inc., Sunnyvale CA.

ICS 2000 Santa Fe New Mexico USA

1 Introduction

In recent years, multithreaded multiprocessor architectures have been emerging as an important class of parallel architectures. The main advantage of multithreaded architectures is their ability to support fast context switching between threads on the same processor. By context switching to a different thread whenever a high latency operation (like a remote memory reference) is encountered, these systems mask communication and synchronization latencies. As a result, these systems allow scalable parallelism for dynamic and irregular codes, where the traditional static analysis methods are unable to perform communication optimizations (like aggregation and prefetching). Examples of multithreaded architectures include Tera corporation's MTA [2], the MIT Alewife [1], and EARTH (*Efficient Architecture for Running THreads*) MANNA, designed at the University of Delaware (and in previous work at McGill University) [12].

In compiling high-level languages for multithreaded architectures, the most important task is *thread partitioning*, i.e., partitioning the code in the high-level language into separate threads of execution. Based upon the thread execution model, multithreaded systems can be of two types. *Preemptive* systems perform a thread context switch after encountering a long latency operation, and enable the thread again when the operation has completed. Tera corporation's MTA is an example of a multithreaded system with preemptive threads. *Non-preemptive* systems execute a thread to completion when it is scheduled. The thread partitioning must be done in the following fashion: if a thread issues a potentially remote memory reference, then any statement that requires that memory reference must be placed in a separate thread. The EARTH design is an example of a multithreaded system with non-preemptive threads.

The main advantage of the non-preemptive model is that the multithreaded system can be constructed from *off-the-shelf* components. For example, the EARTH multithreading model has been successfully emulated on a cluster of Pentium PCs connected through fast Ethernet and a cluster of 4-processor SUN workstations connected through Myricom Myrinet. In contrast, a preemptive design like Tera's requires custom hardware, which can be expensive and time-consuming to construct.

A number of recent projects have addressed the problem of providing high-level imperative language and compiler support for multithreaded systems [3, 10, 16, 20]. In performing thread partitioning on a non-preemptive model,

there are two very important goals:

- Create a sufficiently large number of threads so that there is enough parallelism and communication latency can be easily masked on each node.
- Create threads of sufficient granularity so that the context switching cost is relatively small compared to the cost of performing the actual computations.

A non-preemptive thread execution model requires that a thread boundary be created between two statements such that the first involves a (potentially) remote operation and the second is data dependent upon the first. In the presence of pointer-based recursive data structures, which are frequently used by dynamic and irregular applications, it is difficult to extract precise information about which memory references are remote and where there may be data dependences. Our experience with the current version of the EARTH-C compiler [10, 30, 24, 29] shows that the compiler frequently generates threads of very low granularity in the presence of dynamic, pointer-based data structures. As a result, the execution time of compiler-generated code can be dominated by context switching costs and poor speedups may be observed.

In this paper, we present three novel techniques for improving the precision and representation of data dependence information between statements in codes with pointer-based recursive data structures and structured control flow. In the context of thread partitioning for non-preemptive multithreaded architectures, these techniques enable the generation of coarser threads, and therefore, more efficient execution.

These three techniques are

- Better representation of data dependences between different control blocks in the code, which enables generation of threads that may span different control blocks, even if there are remote references in these control blocks.
- Extracting and representing dependence information in the presence of references to fields of structures, which avoids spurious dependences between statements that read and write different fields of the same structure. This more precise dependence information leads to coarser threads.
- Use of *must alias* or *definite points-to* information for removing redundant remote references. If the program has fewer remote references, there are fewer dependences involving remote references and therefore, threads of higher granularity can be generated.

While the improved precision and representation of data dependences resulting from these techniques can be applicable while performing transformations for parallelism and locality on a variety of systems, the performance benefits are more pronounced in performing thread partitioning on a multithreaded architecture with non-preemptive threads.

We have used a set of 6 benchmarks with pointer-based data structures and/or dynamic control patterns. Our experiments were performed using the in-house version of the EARTH-C compiler [22] and the emulation of the EARTH model on both a cluster of Pentium PCs connected through fast Ethernet and a distributed memory multiprocessor. The overall percentage decrease (across all 6 benchmarks) in the number of static threads generated was 19.2% by using the technique for thread partitioning in the presence of structured control flow, an additional 15.6% by using the technique for disambiguating heap allocated structure references, and an added 3.6% from the use of must alias information. Reductions in execution times ranged from 16% to 45% on the four programs we measured runtime performance.

Overall, our techniques, in conjunction with the existing EARTH-C language and compilation techniques, and the support of the EARTH multithreaded system, enable dynamic and irregular codes written in a high-level language to be compiled for efficient execution on distributed memory parallel machines and clusters of workstations.

The rest of this paper is organized as follows. Multithreaded architectures and our target architecture and compilation environment are described in Section 2. A detailed description of the optimizations is given in Section 3. Experimental results are presented in Section 4. We compare our work with related work in Section 5 and conclude in Section 6.

2 Background

In this section, we describe the source language, the compilation framework, and the target environment of the compiler.

2.1 EARTH-C Language

EARTH-C is an extension of C which allows programmers to specify parallelism and directives for locality [11, 24]. Even though EARTH-C targets a multithreaded environment, thread boundaries, communication involving scalars, and synchronization between threads do not need to be explicitly specified.

The EARTH-C language gives a shared memory view to the programmers, but without supporting cache coherence. For any procedure, local variables and formal parameters are always available in the local memory. However, any dereference to a pointer is considered a potentially remote reference.

The EARTH-C compiler is responsible for correctly compiling any C code for correct execution; however, better performance can usually be achieved if the programmer uses additional directives for specifying parallelism and locality.

The directives in EARTH-C allow the programmer to specify: 1) Parallel/sequential sequences of statements, 2) Parallel loops, by using *forall*, 3) Data locality by declaring some memory references to be *local*, 4) Computation partitioning by declaring some functions to be *basic* (i.e., they should not be spawned on another processor), and 5) Bulk communication (to be used only in some cases) of data aggregates (arrays, structures, etc.).

2.2 EARTH-C Compiler

The work on the EARTH-C compiler was initiated by Laurie Hendren's group at McGill University [10, 11, 24, 30, 9, 29]. This compiler was based upon the McCAT compiler [7, 8], which is a general framework for analyzing and transforming C programs.

At the core of the EARTH-C compiler is the thread generation algorithm [10, 11, 22, 23]. It uses data dependence information in the form of a data dependence graph (DDG) to partition sequential code into threads. Accurate alias analysis information is a must for recognizing dependences between statements in C code. The underlying McCAT infrastructure has an extensive framework for recognizing aliases [7] in C code and propagating them interprocedurally using a context-sensitive analysis. After building the DDG, the compiler recognizes the largest possible sequence of statements such that remote data dependences are met

and therefore can be executed in a single thread. Threads must be generated subject to the split-phase constraint, i.e., the consumption of remote data must be in a different thread than the thread which requests the data. This enables a non-preemptive thread execution policy where communication is overlapped with local computation, thus masking communication latency.

Other more aggressive analyses and transformations already implemented in the EARTH-C compiler (for C and EARTH-C codes) include the following: determining local and non-local references by performing type propagation analysis [30]; using the results of heap and connection analysis for performing optimizations like loop invariant code motion and common subexpression elimination [24]; performing aggressive communication optimizations like message aggregation and eliminating redundant communication [29]; detecting task and loop-level parallelism in C codes [9]; and performing automatic thread partitioning in the form of a general thread partitioning framework [23].

2.3 Target Language and Environment for the Compiler

The target language of the EARTH-C compiler is *Threaded-C*. Threaded-C is a low-level language targeting any multithreaded system which supports a non-preemptive model of thread execution. Threaded-C extends C with directives to explicitly specify thread boundaries, communication, and synchronization between threads. Programming in Threaded-C is comparable to writing message-passing code on distributed memory parallel machines or coding in assembly language on uniprocessors. We can not realistically expect application programmers to program directly in Threaded-C.

As mentioned previously, the EARTH-C compiler was designed for multithreaded systems with *non-preemptive* thread execution. This is in contrast to the design of some other multithreaded systems, like Tera [2], where the thread execution is *preemptive*. A non-preemptive thread execution model can be easily implemented in software on a parallel cluster with off-the-shelf components.

3 Techniques for Improving Thread Granularity

In this section, we present three analyses: improving thread partitioning in the presence of structured control flow, disambiguating heap-allocated structure references, and using must alias information. These three techniques improve the extraction and representation of dependence information in the presence of pointer-based data structures and structured control flow. These techniques, in the context of compiling for multithreaded architectures, are specifically useful for improving the granularity of threads.

3.1 Thread Partitioning in the Presence of Structured Control Flow

As we described earlier in this paper, the main constraint in performing thread partitioning is that a thread boundary must be created between two statements such that the first involves a (potentially) remote operation and the second is data dependent upon the first. However, besides using data dependences and potentially remote operations, we also need to consider control dependences in the original program.

The current thread partitioning techniques deal with control dependencies by using a hierarchical representation, called a *Hierarchical Data Dependence Graph (HDDG)* [23]. A Data Dependence Graph (DDG) is a commonly used program representation. It has one node for each statement in the program. An edge between two nodes represents a data dependence. In the presence of structured control flow, a DDG is extended into a HDDG as follows. Two kinds of nodes are inserted, *simple* nodes representing assignments and *compound* nodes representing control blocks like loops and conditionals. Associated with each compound node is another DDG, which has one node for each simple and compound statement enclosed within this control block. Each compound node contains an edge incident to the DDG representing the node's body. Also, data dependent nodes are linked only at the same level. Therefore, if a statement $s1$, which is not enclosed by any control block, is data dependent with another statement $s2$, that is enclosed within a conditional c, then an edge is inserted between $s1$ and the compound node representing c.

The thread partitioning algorithm is recursively applied at different hierarchy levels (levels of control). Both the simple and compound nodes are classified as *local* or *remote*. A simple statement is *remote* if it can read or write any data that may reside on another processor. A compound statement is *remote* if any of the statements inside the corresponding control block can read or write any data that may reside on another processor. Within each hierarchical level, every statement is assigned a *thread number*. The key insight is that if a node is of remote type and the thread number associated with it is i, then the earliest thread number for any successor of this node can only be $i+1$. If a statement has multiple predecessors in the DDG which are remote, then the thread number of this statement is computed by taking the maximum thread number of all the remote predecessors and incrementing it by one. This can be easily modeled as a data flow problem on the DDG. Statements with the same thread number are grouped in the same thread. For each remote compound node, the thread partitioning algorithm is recursively invoked. If there are N nodes in the DDG at a certain level, the thread partitioner at this level takes $O(N^2)$ time.

This strategy has the advantage of simplicity and efficiency. For procedures with deeply nested loops and control flow, this approach allows a compact representation with potentially few statements in each of the DDGs. Since the list scheduling algorithm is an $O(N^2)$ algorithm, applying the algorithm successively with small values of N at each stage can be quite efficient.

However, in our experience with the current EARTH-C compiler, this technique results in several threads of very small granularity, which can significantly increase execution time. Therefore, we propose an alternative strategy, which requires more storage and a higher processing cost, but can result in more effective thread partitioning.

Graph Representation. We modify the HDDG to include data dependence edges between statements at different levels in the control hierarchy. If a statement $s1$ has a data dependence with another statement $s2$, an edge is inserted between $s1$ and $s2$, irrespective of the control blocks containing $s1$ and $s2$.

Partitioning Algorithm. The partitioning algorithm described earlier is now applied only once for the entire procedure with one important difference. The thread number of a statement is:

- At least one higher than the thread number of any remote statement that it is data dependent upon.
- At least equal to the thread number of any statement it is control dependent upon.

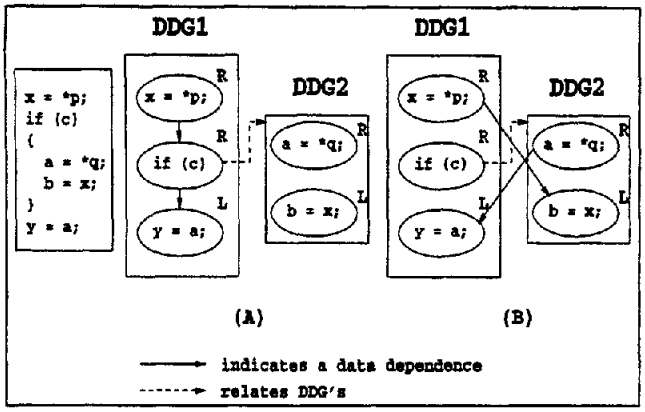

Figure 1: Earth-C Code (left), Original HDDG (middle), and the Modified HDDG (right).

By modeling the computation of thread numbers as a data flow problem, the thread number is calculated for all statements in the procedure. Again, statements with identical thread numbers are combined in the same thread.

Code Generation. Since thread partitioning has been performed across control blocks, one important difference is made while generating threads. Consider two statements $s1$ and $s2$ which are in the same thread. If the statements $s1$ and $s2$ are not in the same control block, the following algorithm is applied for code generation:

- Let the statement $s1$ be control dependent upon the nodes $c1_1, \ldots, c1_n$, such that $s1$ is most closely enclosed within $c1_1$. Similarly, let the statement $s2$ be control dependent upon the nodes $c2_1, \ldots, c2_m$, such that $s1$ is most closely enclosed within $c2_1$. Further, let us refer to $s1$ by $c1_0$ and let us refer to $s2$ by $c2_0$.
- Consider the following two sequences of nodes in the graph: $c1_0, c1_1, \ldots, c1_n$ and $c2_0, c2_1, \ldots, c2_m$. We find the lowest possible values of i and j such that $c1_i$ and $c2_j$ are in the same control block in the procedure.
- In generating the thread which includes $s1$ and $s2$, we also include the sequence of control statements $c1_1, \ldots, c1_i$ and $c2_1, \ldots, c2_j$.

Figure 1 contains EARTH-C code which when currently analyzed results in HDDG (A). An 'R' or 'L' by a statement node indicates that it is a remote or local node, respectively. Note that the flow dependences between `x = *p` and `b = x`, and `a = *q` and `y = a` are not captured. This results in the code on the left in Figure 2, which contains 4 threads. The `SYNC` in `T_2` is needed to enable `T_3` if `c` is false. HDDG (B) shows all dependences using inter-DDG edges. This allows a better thread partitioning, as the code on the right in Figure 2 contains only 2 threads.

3.2 Disambiguating Heap-Allocated Structure References

Multithreaded architectures are particularly targeted towards achieving scalable parallel performance from dynamic and irregular applications. Dynamic and irregular applications frequently use recursive data structures (lists, trees, etc.), which means frequent allocation of heap-based memory.

```
T_0:;
  x = *p; // remote read
  END_THREAD ();

T_1:; // fires after T_0
  if (c)
  {
    a = *q; // remote read
    b = x;
    END_THREAD ();

T_2:; // fires after T_1
  }
  SYNC (3);
  END_THREAD ();

T_3:; // fires after T_2
  y = a;
```

```
T_0:;
  x = *p; // remote read
  if (c)
  {
    a = *q; // remote read
  }
  END_THREAD ();

T_1:; // fires after T_0
  if (c)
  {
    b = x;
  }
  y = a;
  END_THREAD ();
```

Figure 2: Improving Thread Partitioning: an *unoptimized* code (left) produced from the HDDG in the middle of Figure 1 and an *optimized* code (right) produced from the HDDG on the right of Figure 1.

As previously stated, in the presence of aliasing, dependence information is difficult to compute because a read (or write) to a variable `x` implies a read or write to any variable which may be aliased with `x` at that program point. Computing points-to information or pointer-induced aliases is a well studied compilation problem [5, 7, 8, 14, 26]. The primary focus of all these techniques for computing pointer-induced aliases is on aliases between references to the stack. In other words, they mainly focus on named, stack-allocated variables of primitive data type, and pointers that point to stack locations. Computationally intensive dynamic and irregular applications frequently use heap-allocated structures in computations. Improving disambiguation of heap-allocated references is critical for having reasonably accurate dependence information for such codes.

The existing alias analysis techniques use a number of different approaches for computing points-to information between heap-allocated memory. The simplest approach is to abstract all heap allocated memory as a single stack location, referred to as *heap*. This approach is taken by Emami, Hendren and Ghiya [7] and is used in the current implementation of the EARTH-C compiler [10, 11, 30]. The advantage of this approach is its simplicity and efficiency; the alias analysis does not get any more expensive than without considering the heap. However, this results in many false dependences being incorporated into the data dependence graph (DDG), which is used to partition threads. For example, the sequential code on the left in Figure 3 demonstrates the problems that can arise without accurate heap alias information. Since the structure references `t->sz`, `t->next`, and `t->prev` can never be aliases, there are no WAW dependences among the three statements. (Note this is not the case if `t` points to a union.) The following `read` statement, however, is flow dependent on the previous three statements. If the heap is abstracted as a single location, the partitioned threaded segment in the middle will result. Eliminating the false WAW dependences allows us to merge the first three threads to obtain the better partition on the right. The following references, however, would cause a problem: `t->next` and `p->next`. These expressions are aliases if `t` and `p` point to the same location. However,

when the same structure is referenced within a function, it is likely it will be referenced through the same pointer.

Another simple technique used for treating heap references is to name the heap object by the location in the program where it is allocated [5]. This approach is again very simple and efficient. However, this simple approach is not sufficient to resolve two separate fields of the same structure. For example, this technique will note a write-after-write (WAW) dependence between the statements `t->next = 0` and `t->prev = 0`.

A number of other more aggressive techniques have been suggested for disambiguating references between heap objects [8, 14, 19, 26, 27]. These techniques are also very expensive. We discuss these techniques toward the end of this subsection.

We present a relatively simple and efficient technique for disambiguating heap-allocated structure references. Our technique is not significantly more expensive than considering all heap locations as a single location, *heap*, or just naming the heap object by the place in the program it is allocated. However, as shown later in our experimental results, the dependence information computed by our technique results in threads of significantly higher granularity as compared to the existing technique in the current EARTH-C compiler (based upon abstracting the entire heap as a single location *heap*).

Our proposed technique is as follows:

- Consider a statement of the form `t = malloc(size)`, where `t` is a pointer to a structure. Let the statement have a label `label`. For any field `f` of the structure referenced by `t`, we assume a memory location `(label,f)`. After such an allocation, the reference `t->f` points to `(label,f)`.
- If the statement with label `label` is inside a loop or recursive procedure call, we do not provide separate names to memory allocated in different executions of this statement.
- The abstract memory locations thus created are treated as stack locations and a points-to analysis algorithm like Emami's [7] is applied.

This method does not add significant complexity to the original alias analysis algorithm. But, this algorithm is successful in differentiating references like `t->prev` and `t->next` as separate fields of the same structure. Moreover, if one of the fields of a structure is a pointer which points to a field of another structure, this information can be accurately computed by the above technique.

We now briefly explain the existing aggressive techniques for extracting accurate dependence information between heap objects. Ghiya and Hendren present two techniques, *Connection Analysis* and *Shape Analysis*, for extracting important properties of recursive data structures [8]. The main idea behind Hendren's approach is to completely separate analysis of stack-based variables and heap-allocated objects and use sophisticated techniques for heap-allocated objects. Lam and Wilson [26] use the concept of *location sets*, which is a low-level representation that treats array references and fields of structures in a common framework. They specifically consider three tuples, which include the stride between consecutive elements. Landi and Ryder use the access paths as names of anonymous heap objects, and *k-limit* the access paths to have a finite set of object names in the presence of recursive structures. Our technique is essentially the same with the value of k set to 1.

```
p = &g1;                          p = &g1;
q = p; // aliases: g1, *q, *p     q = p; // aliases: g1, *q, *p
s1: *q = 10;                      q = 10;
s2: temp_16 = *p;                 temp_16 = 10;
s3: temp_17 = *q;                 temp_17 = 10;
s4: e_temp_0 = g1;                e_temp_0 = 10;
e_temp_1 = g2;                    e_temp_1 = g2;
print(e_temp_0, e_temp_1,         print(e_temp_0, e_temp_1,
  temp_16, temp_17);                temp_16, temp_17);
```

Figure 4: Using Must Alias Information: an *unoptimized* sequential code (left) and an *optimized* sequential code (right).

Our technique is considerably simpler than the above techniques. The main contribution of our work is a relatively simple and easy to implement extension of points-to techniques for stack locations. This extension results in significantly improved dependence information for structure references, and consequently, threads of significantly higher granularity.

3.3 Using Must Alias Information

The alias or points-to information available at any point in the program is of two types:

- *May alias* or *Possible Points-to* information which implies that two memory locations may be, but are not necessarily, aliases.
- *Must alias* or *Definite Points-to* information which implies that two memory locations are definitely aliases at that program point, irrespective of the execution paths taken in reaching that program point.

The initial work in the area of computing pointer-induced aliases focused on computing may aliases [14, 5]. One charactestic of may alias is that it can be computed correctly (though not necessarily accurately) by more efficient flow and context insensitive techniques. Computing reliable must alias information requires more detailed flow and context sensitive techniques. The first uniform treatment of must and may aliases was presented by Emami, Ghiya and Hendren [7].

We use the must alias or definite points-to information for reducing redundant communication, which subsequently results in improved thread granularity. To illustrate the benefits of this transformation, we first show a sequential C code. Consider Figure 4 which shows aliases reported as definite aliases by points-to analysis. The must alias information is not used in the segment on the left. Since g1, *q, and *p are must aliases, the write to *q in s1 implies the indirect memory references in statements s2, s3, and s4 are unnecessary and can be replaced with the constant 10, as in the sequence on the right. This can be modelled as a dataflow problem and availability of pointer dereferences at different program points can be computed. The details of the analysis are beyond the scope of this paper.

Figure 5 illustrates the transformation in a multithreading context. In EARTH-C, pointer dereferences result in remote reads or writes unless additional locality information is provided. The must alias information is not exploited in generating threads in the code shown on the left. However, the use of must alias information shows that the remote reads of *q and g1 are unnecessary. In addition, because *q

```
t- >sz = n;
t- >next = 0;
t- >prev = 0;
read (t);
```

```
T_0:;
  t- >sz = n;
  END_THREAD ();

T_1:; // fires after T_0
  t- >next = 0;
  END_THREAD ();

T_2:; // fires after T_1
  t- >prev = 0;
  END_THREAD ();

T_3:; // fires after T_2
  read (t);
  END_THREAD ();
```

```
T_0:;
  t- >sz = n;
  t- >next = 0;
  t- >prev = 0;
  END_THREAD ();

T_1:; // fires after T_0
  read (t);
  END_THREAD ();
```

Figure 3: Thread Partitioning in the Presence of Structure References: a sequential code (left), an *unoptimized* threaded code (middle), and an *optimized* code (right).

```
T_0:;
 p = &g1;
 q = p; // aliases: g1, *q, *p
 *q = 10; // remote write
 END_THREAD();

T_1:; // fires after T_0
 temp_16 = *p; // remote read
 END_THREAD();

T_2:; // fires after T_0
 temp_17 = *q; // remote read
 END_THREAD();

T_3:; // fires after T_0
 e_temp_0 = g1; // remote read
 END_THREAD();

T_4:; // fires after T_0
 e_temp_1 = g2; // remote read
 END_THREAD();

T_5:; // fires after T_1, T_2, T_3, T_4
 print(e_temp_0, e_temp_1,
   temp_16, temp_17);
 END_THREAD();
```

```
T_0:;
 p = &g1;
 q = p;
 *q = 10;
 temp_16 = 10;
 temp_17 = 10;
 e_temp_0 = 10;

T_4:; // fires after T_0
 e_temp_1 = g2;
 END_THREAD();

T_5:; // fires after T_4
 print(e_temp_0, e_temp_1,
   temp_16, temp_17);
 END_THREAD();
```

Figure 5: Using Must Alias Information: an *unoptimized* threaded code (left) produced from the *unoptimized* sequential code in Figure 4 and an *optimized* threaded code (right).

and *p are aliases, the remote read of *p can be eliminated. The threaded code that is generated after using must alias information is given on the right.

4 Experimental Results

In this section, we present the results demonstrating the usefulness of the techniques we have presented for improving thread granularity. Before presenting the measurements from our experiments, we describe the parallel configurations and the set of benchmarks used.

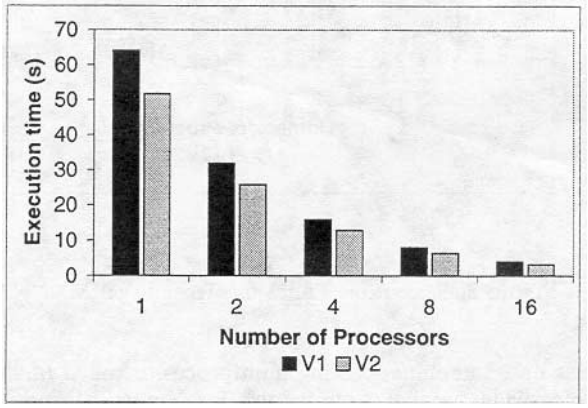

Figure 7: Execution Times for N-Queens on MANNA

4.1 Parallel Configurations

Our experiments have been performed in two EARTH multithreaded environments that support non-preemptive threads. EARTH, which stands for Efficient Architecture for Running THreads, was designed by Gao's group at the University of Delaware, and in earlier work at McGill University [12]. The basic idea is that every node in the EARTH system has an Execution Unit (EU), for executing threads, and a synchronization unit (SU), for handling thread synchronization and communication with remote processors. The SU may be implemented in hardware or software. The EU executes an active thread, which is initiated for execution when the EU fetches its *thread id* from the *ready queue*. The EU interacts with the SU and the network by placing messages in the *event queue*. The SU fetches these messages, plus messages from remote processors via the network. The SU also determines which threads are to be run and adds their thread ids to the ready queue. A thread is enabled when all the data required by the thread is available.

We have used two different implementations of the EARTH model in our experiments. The first system, which was also the first implementation of the EARTH design,

Benchmark	Description
N-Queens	Search for all legal queen configurations on a chess board
Treeadd	Add values at all nodes of a tree
Power	Optimization problem based upon a variable k-ary tree
Perimeter	Computes the perimeter of a quad-tree encoded raster image
Health	Simulates the Colombian health care system using a 4-ary tree
MST	Finds a minimum spanning tree for a graph

Figure 6: Benchmark Programs

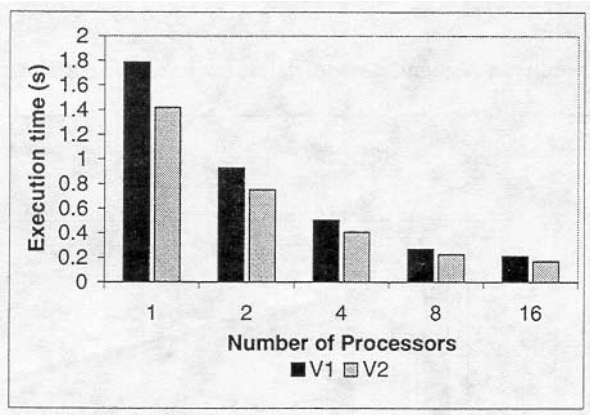

Figure 8: Execution Times for Treeadd on MANNA

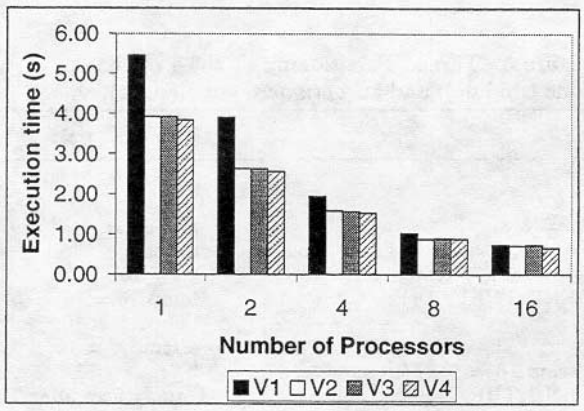

Figure 9: Execution Times for Perimeter on MANNA

was based upon an existing multiprocessor called MANNA (Massively parallel Architecture for Numerical and Non-numerical Applications), developed at GMD-FIRST in Berlin [4]. Each MANNA node consists of two Intel i860XP CPUs clocked at 50 MHz, 32 MB of dynamic RAM and a bidirectional network interface capable of transferring 50 MB/s in each direction. The two processors on each node are mapped to the EARTH EU and SU.

The second system used in our experiments is a more recent mapping of the EARTH model to an 8 processor Beowulf named Earthquake. It is a cluster of 4 Pentium II-333's with 128 MB of memory and 4 Pentium Pro-200's with 64 MB of memory connected with 100Base-T Ethernet. The cluster uses Linux as its operating system.

4.2 Benchmark Programs

As mentioned previously, our target applications contain dynamic and recursive control patterns and/or dynamic pointer-based data structures. Our overall goal is to enable these applications to be written in a high-level parallel language and compiled for efficient execution on distributed memory machines and clusters of workstations. We have used 6 applications written in EARTH-C for our study. Though the current compiler produces Threaded-C output from each of these applications, not all of them could be run on the EARTH mappings because of a number of limitations. Therefore, we have reported execution times on four applications: `N-Queens`, `Treeadd`, `Perimeter`, and `MST`. We also have reported static reductions in thread counts on all 6 benchmarks. The benchmarks are summarized in Figure 6.

4.3 Results from Experiments

We have presented three analyses for improving thread granularity in this paper. Each analysis was applied on the six benchmarks and four versions (`V1`, `V2`, `V3` and `V4`) were created. Version `V1` is what the compiler produces without any of our analyses. We created version `V2` by applying the thread partitioning technique in the presence of structured control flow to version `V1`. Version `V3` augments version `V2` with the addition of the technique for disambiguating heap allocated structure references. The final version, `V4`, additionally uses must alias information to coarsen threads.

As previously described, we are able to present execution times only on four of the benchmarks, `N-Queens`, `Treeadd`, `Perimeter`, and `MST`. `N-Queens` was executed on both MANNA and Earthquake. `Treeadd`, which is a very fine grained tree traversal code, does not produce any speedups on Earthquake, because of expensive communication on top of the Ethernet interconnect. `Perimeter`, which is also a very fine-grained code, was executed on MANNA and `MST` was executed on Earthquake.

Figure 7 presents execution times for `N-Queens` on MANNA. `N-Queens` exhibited near perfect speedup on MANNA. This is due to MANNA's low context switching time and its high bandwidth interconnect. Version `V2` produced nearly a 20% reduction in execution times over version `V1`. The second and third techniques did not produce further gains, as versions `V3` and `V4` were the same as version `V2` for this code.

Figure 8 presents execution times for `Treeadd` on MANNA. `Treeadd` displayed good speedups even though the computation performed is extremely simple (summing

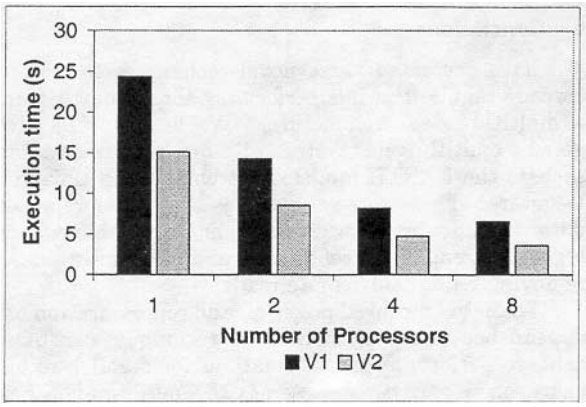

Figure 10: Execution Times for N-Queens on Earthquake

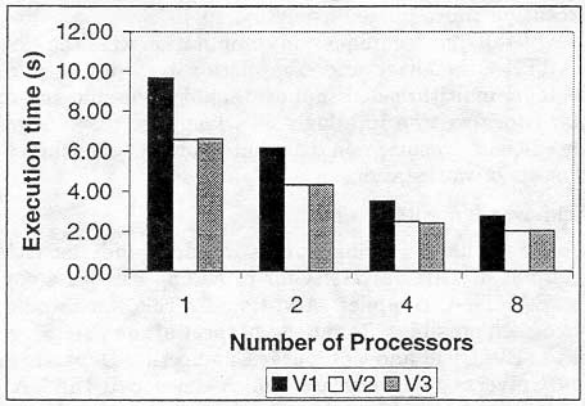

Figure 11: Execution Times for MST on Earthquake

the integers stored at each node in a binary tree, with one integer per node). Reductions in execution times from version V2 ranged from 16% on 8 processors to 21% on 1 processor.

Figure 9 presents execution times for Perimeter on MANNA. Speedups for Perimeter ranged from 1.40 on 2 nodes to 7.44 on 16 nodes. Reductions in execution times from version V2 ranged from 33% on 2 processors to 2% on 16 processors. Statically, version V2 has 22 fewer threads than version V1. As the same problem is executed on more processors, more threads are required to keep all processors busy. Therefore, the gains from reducing the context switching overhead are offset by the reduction in available parallelism. Version V3 did not produce further reductions. Though V3 has 14 fewer threads, all the threads eliminated are in an allocation routine which is executed only once and is not a time-consuming component of the program. Reductions from version V4 ranged from 2% on 1 processor to 10% on 16 processors. Statically, V4 has 3 fewer threads than V3 and 2 of the threads eliminated are in the core routines, which results in modest performance gains.

The execution times for N-Queens on Earthquake are presented in Figure 10. The speedups for N-Queens on Earthquake were not as impressive as those for MANNA, as Earthquake uses a commodity network (Ethernet) for communication. However, because of the Beowulf's increased overhead, the effects of the optimization were much more pronounced. Reductions in execution times ranged from 38% on 1 processor to 46% on 8 processors.

The execution times for MST on Earthquake are presented in Figure 11. Again, the speedups for MST on Earthquake were not that impressive, ranging from 1.52 on 2 nodes to 3.48 on 8 nodes. However, reductions in execution time were strong. Version V2, which statically has 10 fewer threads, produced reductions ranging from 27% on 8 nodes to 32% on 1 node. Version V3 did not produce further reductions. V3 has 2 fewer threads, but these threads were in the edge allocation routine which is executed only once in the initialization phase.

We summarize the static results for all six benchmarks. The four different versions of these programs were examined manually (and by using Perl scripts). The number of threads and average thread granularity are reported for each version of each benchmark. Average thread granularity was obtained by counting the total number of statements in all threaded functions and dividing by the number of threads. Figure 12 shows the static thread counts and Figure 13 presents the average thread granularities.

All three techniques improved thread granularity in three of the six programs in our benchmark set. Two of the three techniques improved thread granularity on the fourth code, MST, whereas all the benchmarks benefited from at least one of the three techniques. The technique for improving thread partitioning in the presence of structured control flow improved thread granularity on all 6 benchmarks, the technique of disambiguating heap allocated structure references benefited 4 of the 6 codes, and the technique of using must aliases reduced the number of threads in 3 of the 6 benchmarks.

The reduction in the number of threads was substantial, ranging from 23% for Treeadd to 48% for Power. Even a small reduction in thread count can cause a large performance benefit, as the execution times for N-Queens and Treeadd demonstrate. Two obvious gains from decreasing the thread count are the decrease in synchronization cost and the decrease in thread switching overhead.

The overall percentage decrease (across all 6 benchmarks) in the number of threads generated was 19.2% by using the technique for thread partitioning in the presence of structured control flow, an additional 15.6% by using the technique for disambiguating heap allocated structure references, and an added 3.6% from the use of must alias information.

Because each version contained fewer threads and the total number of statements either decreased very marginally or remained constant, thread granularity increased by applying our techniques. Overall, these increases result in better processor utilization, as context switching overhead is amortized over larger units of work.

5 Related Work

In this section, we discuss the relationship of our work with existing related research efforts.

A number of imperative languages have been designed in recent years for multithreaded systems. Nikhil developed a "shared-memory" extension of C called Cid [16]. This system is mainly based upon programmer specified directives and the runtime system; very limited compiler analysis is performed. Cilk is a language developed at MIT for pro-

Benchmark	V1	V2	V3	V4
N-Queens	10	7	7	7
Treeadd	13	10	10	10
Power	77	68	44	40
Perimeter	83	61	47	44
Health	143	120	103	97
MST	39	29	27	27

Figure 12: Static Thread Count

Benchmark	V1	V2	V3	V4
N-Queens	4.30	5.71	5.71	5.71
Treeadd	3.08	3.70	3.70	3.70
Power	4.32	4.76	7.36	7.68
Perimeter	2.93	3.62	4.70	5.02
Health	2.21	2.44	2.88	3.06
MST	3.92	4.93	5.30	5.30

Figure 13: Average Thread Granularity

gramming multithreaded machines, mainly shared memory machines [3]. Similarly, EM-C is a extension to C developed in the context of the EM-4 multithreaded machine [20] which also expects the programmer to specify remote accesses and synchronization explicitly. These languages are similar to the Threaded-C language (which is the target of the EARTH-C compiler) and requires that programmers explicitly specify threads and synchronization between the threads.

Tera corporation's MTA machine [2] is now commercially available along with its production level compilers. In this machine, thread execution is preemptive. Such a design eases the compilation task, but such machines cannot be built from off-the-shelf uniprocessors.

The recent simultaneous multithreading project from the University of Washington [15] focuses on using several threads within a single-chip superscalar machine. The different threads can share the superscalar capabilities of the system within a single cycle. Similarly, the Superthreaded project from the University of Minnesota [25] allows multiple threads to be executed concurrently with thread-level control speculation and runtime data dependence checks to speed up single program execution. In our work, we rely on the advanced alias analysis technique to eliminate nonexistent data dependences.

The thread partitioning problem has long been studied within the functional language community. For compiling for lenient languages, the dependence set method merges threads with the same inputs [13], while the demand set algorithm combines threads with the same output [21]. The combination of both, together with global analysis, is reported in [6]. In compiling for strict languages, top-down and bottom-up approaches are proposed [18]. Our current work is an improvement over the thread partitioning work built on the EARTH-C compiler [10, 11, 24, 23, 22].

The Concert compiler uses the notion of dynamic pointer alignment for tiling and communication optimizations for pointer-based codes [28]. Another system that allows pointer-based codes to execute on distributed memory machines is Olden, developed by Rogers *et al.* [17]. We believe that our approach will be able to achieve better performance on a larger number of codes because of the use of multithreading.

6 Conclusions

We have presented three novel techniques for improving thread granularity while performing thread partitioning for a multithreaded architecture. We have specifically targeted a multithreaded system with non-preemptive threads, such as the EARTH modeled systems at the University of Delaware. These three techniques are better representing data dependences between different control blocks, disambiguating heap-based references, and using must aliases for removing redundant remote reads.

While the improved precision and representation of data dependences resulting from these techniques can be applicable to performing transformations for parallelism and locality on a variety of systems, the performance benefits are very pronounced in performing thread partitioning on a multithreaded architecture with non-preemptive threads. On a set of 6 benchmark programs with pointer-based data structures and dynamic control patterns, these three techniques result in an overall 38% reduction in the static number of threads generated. For the four programs that we have reported runtime performance, the reductions in the execution times ranged from 16% to 45%.

Overall, our techniques, in combination with the existing EARTH-C language and compilation techniques, and the EARTH multithreaded support, enable dynamic and irregular codes written in a high-level language to be compiled for efficient execution on distributed memory machines and clusters of workstations.

Acknowledgments

We would like to thank Laurie Hendren and the ACAPS group at McGill University for providing us with a copy of the EARTH-C compiler. Additionally, this work would not have been possible without the support of the CAPSL group in the Electrical and Computer Engineering Department at the University of Delaware, which developed the EARTH model and frequently helped us during our experiments.

References

[1] Anant Agrawal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of ISCA-17*, 1990.

[2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6. ACM Press, June 1990.

[3] R. D. Blumofe and C. F. Joerg *et al.* Cilk: An efficient multithreaded runtime system. In *Proceedings of PPoPP-5*, 1995.

[4] U. Bruening, W. K. Giloi, and W. Schoroeder-Preikschat. Latency hiding in message passing architectures. In *Proceedings of IPPS-8*, 1994.

[5] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, South Carolina, January 1993.

[6] David E. Culler, Seth C. Goldstein, Klaus E. Schauser, and Thorsten von Eicken. TAM – a compiler controlled threaded abstract machine. *Parallel and Computing*, 18:347–370, July 1993.

[7] 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 1994 Conference on Programming Languages Design and Implementation*, June 1994.

[8] 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 *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–15. ACM Press, January 1996.

[9] Rakesh Ghiya, Laurie J. Hendren, and Yingchun Zhu. Detecting parallelism in C programs with recursive data structures. In *Proceedings of the 1998 International Conference on Compiler Construction*, March 1998.

[10] Laurie Hendren, Xinan Tang, Yingchun Zhu, G. R. Gao, Xun Xue, H. Cai, and Pierre Ouellet. Compiling C for the EARTH multithreaded architecture. In *Proceedings of PACT*, 1996.

[11] Laurie J. Hendren, Xinan Tang, Yingchun Zhu, and Guang R. Gao. Compiling C for the EARTH multithreaded architecture. *International Journal of Parallel Programming*, 1997.

[12] H. H. J. Hum and Oliver Maquelin *et al.* A design study of the EARTH multiprocessor. In *Proceedings of PACT*, 1995.

[13] Robert A. Iannucci. A dataflow/von Neumann hybrid architecture. MIT/LCS/TR-418, MIT for ACM Computing Surveys, Cambridge, July 1988. PhD thesis, May 1988.

[14] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27, pages 235–248, San Francisco, CA, June 1992.

[15] Jack Lo, Susan Eggers, H. M. Levy, Sujay Parikh, and Dean Tullsen. Tuning compiler optimizations for simultaneous multithreading. In *Proceedings of Micro-30*, 1997.

[16] Rishiyur Nikhil. Cid: A parallel C for distributed memory machines. Technical report, Cambridge Research Laboratory, Digital Equipment Corporation, 1994.

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

[18] 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 on Parallel Architectures and Compilation Techniques, PACT '94*, pages 37–46, Montreal, Canada, August 24–26, 1994. North-Holland.

[19] Radu Rugina and Martin Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 77–90, May 1999.

[20] M. Sato, Y. Kodama, S. Sakai, and Y. Yamaguchi. EM-C: Programming with explicit parallelism and locality for EM-4 multiprocessor. In *Proceedings of PACT*, 1994.

[21] Klaus Eric Schauser, David E. Culler, and Thorsten von Eiken. Compiler-controlled multithreading for lenient parallel languages. No. UCB/CSD 91/640, ACM Computing Surveys , at Berkeley, 1991.

[22] Xinan Tang. *Compiling for Multithreaded Architectures.* PhD thesis, Electrical and Computer Engineering Department, University of Delaware, Newark, DE, 1999.

[23] Xinan Tang and Guang R. Gao. Automatically partitioning threads for multithreaded architectures. *Journal of Parallel and Distributed Computing*, 58(2):159–189, August 1999.

[24] Xinan Tang, Rakesh Ghiya, Laurie J. Hendren, and Guang R. Gao. Heap analysis and optimizations for threaded programs. In *Proceedings of the 1997 Conference on Parallel Architectures and Compilation Techniques*, November 1997.

[25] Jenn-Yuan Tsai, Zhenzhen Jiang, Eric Ness, and Pen-Chung Yew. Performance study of a concurrent multithreaded processor. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 24–35, Las Vegas, Nevada, January 31–February 4, 1998.

[26] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–13. ACM Press, June 1995. ACM SIGPLAN Notices, Vol. 30, No. 6.

[27] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer analysis for programs with structures and casting. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 91–103, September 1999.

[28] Xingbin Zhang and Andrew A. Chien. Dynamic pointer alignment: Tiling and communication optimizations for parallel pointer-based computations. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 37–47. ACM Press, June 1997.

[29] Yingchun Zhu and Laurie Hendren. Communication optimizations for parallel c programs. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 199–211. ACM Press, 1999.

[30] Yingchun Zhu and Laurie J. Hendren. Locality analysis for parallel C programs. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, November 1997.