PROFILING IRREGULAR APPLICATIONS ON THE EARTH MULTITHREADED ARCHITECTURE

by

William M. Lowe

PROFILING IRREGULAR APPLICATIONS ON THE EARTH MULTITHREADED ARCHITECTURE

by

William M. Lowe

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer and Information Sciences with Distinction

May 1999

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION AND BACKGROUND

Parallel processing is an important design strategy in modern high-performance computing, but applications with unpredictable and dynamic data access patterns (i.e., emphirregular applications) are difficult to parallelize and to optimize for parallelism. Multithreading has shown promise as a method of improving the performance of irregular applications. In order to improve our understanding of how to best parallelize and optimize irregular applications, we must gather profiling information about these applications at runtime. This thesis investigates the problem of providing profiling information for multithreaded programs on the EARTH multithreaded architecture.

1.1 Parallel Processing

Many problems in modern computer science contain several different tasks which have varying levels of dependencies on each other. The algorithm in Figure 1.1 is comprised of four interrelated steps.

Step 1 has no dependencies on the other steps. We must, however, complete Step 1 before proceeding to Steps 2 and 3, because these steps rely on the products of Step 1. We find that Step 4 also must wait for all three of the previous steps to complete.

There is, however, no reason that Step 3 must wait for Step 2 to finish. If we have two knives and a means for holding two slices of bread simultaneously, we can do Steps 2 and 3 at the same time (in *parallel*). If all steps take the same amount

```
To make a peanut butter sandwich:

 1. Get two slices of bread and open the peanut butter jar.
 2. Spread peanut butter on slice 1.
 3. Spread peanut butter on slice 2.
 4. Put the slices together and eat.
```

Figure 1.1: An algorithm for making a peanut butter sandwich.

of time, we have now finished making our sandwich in 3/4 of the time it would have taken without parallelism.

Computer programs also often contain steps which can be executed in parallel, but many computers are capable of executing only one step at a time. *Multithreaded Architectures* [1, 3, 5, 6, 9, 19, 20, 21] are one class of computing machines which are designed to execute parallel programs. Individual sequences of sequential steps are called *threads* or *strands*, and the process of dividing a program into threads by identifying the data-implied boundaries between threads is known as *thread partitioning*. Threads which rely on these data produced by other threads are said to have *data dependencies* on those other threads. Some multithreaded systems create *non-preemptive* threads; in the model that I focus on in this thesis, a thread runs from beginning to end without interruption by any other threads.

1.2 Irregular Applications

All computer programs use data stored in some sort of computer memory, such as random access memory (RAM), magnetic disk, and CD-ROM. Many programs will access this data in a predictable pattern that will not change unpredictably during execution of the program. An example of such an application is matrix multiplication, shown in Figure 1.2 (adapted from [22], p. 532).

```
#define N 1000
void matrix_multiply(int a[N][N], int b[N][N], int c[N][N])
{
        int i, j, k;

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

Figure 1.2: A regular application: matrix multiplication.

Because the data access patterns of such applications are mostly static, compilers can identify these patterns without running the program, and perform optimization on these programs to improve their performance. These kinds of programs are called *regular applications*.

Irregular applications are applications with data access patterns that cannot be reliably predicted before the program is run. Because these patterns cannot be deduced by the compiler, the compiler is often unable to apply optimizations which might improve the program. Other methods must be used to obtain optimization clues for this class of applications.

1.3 Compilers and Optimization

Compilers are tasked not only with the translation of a program from a language usable by humans to one understood by computer hardware, but also with the job of analyzing a program for inefficiencies and improving the program by removing them.

Modern computer programs are often complicated and consume significant computing resources (CPU time or memory) when executing. The goal of *compiler optimization* is to enable compilers to minimize the time or space required to

```
/* this function will never execute */
void foo()
{
        printf("Hi!  I'm function foo.\n");
}

/* just print a message and exit */
int main()
{
        printf("Hello,  world!");
        exit(0);

        /* this line also never executes */
        foo();
}
```

Figure 1.3: Sample program containing code which is never executed.

execute programs. Optimizing compilers generally do their work by applying *transformations* to (an intermediate form of) the programmer's source code which, for example, make it more amenable to the underlying hardware, reduce redundancy, or eliminate code that is never executed.

It is essential that the code produced by an optimizing compiler function identically to the code provided by the programmer. This implies that the compiler must be able to deduce information from the program's code about how the program will function while it is running. Consider the example in Figure 1.3.

If the compiler is able to correctly determine that function `foo()` will never execute, it can remove the call, and if this is the only call to `foo()`, the compiler can completely remove the function from the program before the final translation to machine code.

Some programming languages allow the programmer to give "hints" to the compiler about optimizable features. The C code

```
const int x;
```

allows the programmer to inform the compiler that the value of `x` will not change during the execution of the program, which then allows the compiler to perform certain optimizations based on this information. It is not possible, however, for a programmer to completely specify such information. Furthermore, the programmer may be unaware of information which would be of use to the compiler, possibly because the programmer is not familiar with the underlying hardware architecture or because the program is large and the programmer cannot mentally trace all of the uses of the variable `x` in the code. In addition, many optimizable features of programs are due to the inefficiencies introduced by the first few phases of the compiler, are not due to bad programming, and are not even visible at the user's level.

1.4 Profiling

Often, programmers want information about how parts of the code were excercised during a run of the program with a particular data set. Information about the execution of a program can be gathered if the programmer inserts explicit commands to the code, for example: "When function `foo()` is called, print a message on the console." While this will allow the user to determine that `foo()` has begun to execute, this method is cumbersome, as the programmer must remember to insert these calls, and then must usually remove them before the final version of the program is compiled.

Software tools called *profilers* can be used to automatically add such commands into a program; this technique is called *code instrumentation.* Several tools which can be used to implement profilers are freely available, such as `gprof` [12, 13], ATOM [23], and EEL [18]. Aside from gathering function call information, many of these tools can also monitor the values of memory locations and collect other useful data. Programs can be instrumented at the source code level, intermediate (internal to the compiler) level, assembly language level, or binary level.

Information gathered using a profiler can also be used by a compiler to apply program-specific optimizations in a process called *profile-guided optimization*. Such techniques have been used, for example, to eliminate some redundancy in programs [15] and remove code which never executes [14]. Profiling can also be used to identify program variables whose values do not change frequently, allowing the compiler to apply some optimizations as if these variables were constants [7]. With profiling information, the optimization phase of compilation can concentrate expensive optimization techniques on "hot spots" in the code (those areas of the program which are executed most frequently).

Chapter 2

PROBLEM AND MOTIVATION

Parallel processing has shown promise as a method of solving many computationally complex problems, and several systems capable of parallel execution have been constructed in the last decade [1, 3, 5, 6, 9, 19, 20, 21]. One design issue in the construction of such systems is the increased burden to the user; if parallel systems are more difficult to use than standard sequential computer systems, fewer people will be able to take advantage of the opportunities afforded by parallel execution. Thread partitioning, in particular, is difficult for programmers accustomed to the sequential execution model. This may be because the data dependencies between potential threads are complex, or simply because parallelizable components of a task are not always easily identifiable. It is thus useful to remove as much of the burden of thread partitioning as possible from the user. The EARTH-C compiler system, described below, attempts to achieve this goal by allowing the user to provide some information about possible parallelism, but actually performing the parallelization tasks in the compiler.

2.1 The EARTH Multithreaded Architecture

The EARTH (Efficient Architecture for Running THreads) parallel architecture was developed at McGill University and continues to be developed at the University of Delaware to study parallel architecture design and compilation issues. The EARTH architecture, described in detail in [17], provides for building a multiprocessor multithreaded machine from off-the-shelf components such as individual

PCs by linking the machines (called *nodes*) via a network such as Ethernet (see Figure 2.1). Because each node has its own memory (and thus its own copy of global variables), it can be difficult to make sure that the value of a variable on node 1 is the same as the value of the same variable on, say, node 5. To help resolve these difficulties, each node in the system has two processors; one is dedicated to handling synchronization and data requests (the *synchronization unit* or SU) and one is dedicated to actually running parts of the program (the *execution unit* or EU). One node is designated *Node 0*; the first thread of the program's `MAIN()` function runs on Node 0, and Node 0 maintains the master copies of global variables.

EARTH Threads are *non-preemptive*; that is, once a thread begins running, that thread will continue running uninterrupted until it has completed.

2.1.1 Languages for Programming EARTH

Two dialects of the C programming language have been designed for use on EARTH. One of these, called Threaded-C, requires the programmer to specifically set up threads and ensure that all data dependencies are met before a given thread can begin execution [16, 24]. In Figure 2.2, the `RSYNC` call is used to alert waiting threads that this instance of the `print_hello` function has completed. The other language, EARTH-C, allows the programmer to give hints to the compiler about which parts of the program may be parallelized, but places the burden of thread partitioning, handling data synchronization, and other low-level tasks on the compiler. Figure 2.3 shows a simple EARTH-C program in which the {^ and }^ operators are used to inform the compiler that the operations in this block of code do not have any dependencies on each other.

Memory addresses in EARTH are grouped into two types: *local* and *global*. The local pointer types are identical to standard ANSI C pointers and are used to access memory on the local node. Global pointers (declared with the `GLOBAL`

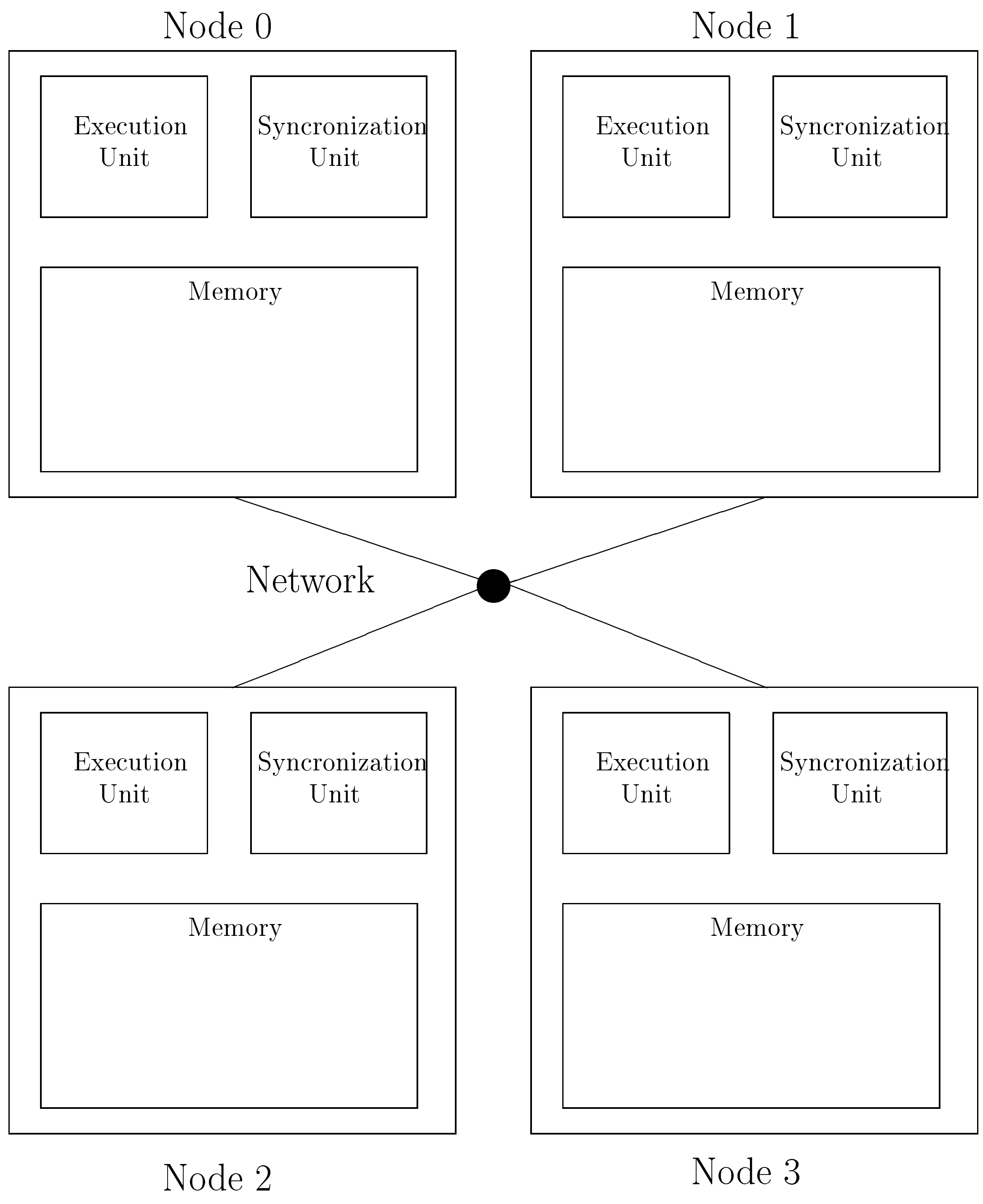

Figure 2.1: An EARTH system consisting of four nodes.

```
#include        <stdio.h>

THREADED print_hello(SPTR done)
{
        printf("node %d: Hello World!\n", NODE_ID);

        RSYNC(done);
        END_FUNCTION();
}
```

Figure 2.2: Sample code written in Threaded-C.

```
replicated int i1, i2;

int foo()
{
    return 0;
}

void main (void)

    {^
        i1 = foo();
        i2 = foo();
    ^}
}
```

Figure 2.3: Sample code written in EARTH-C.

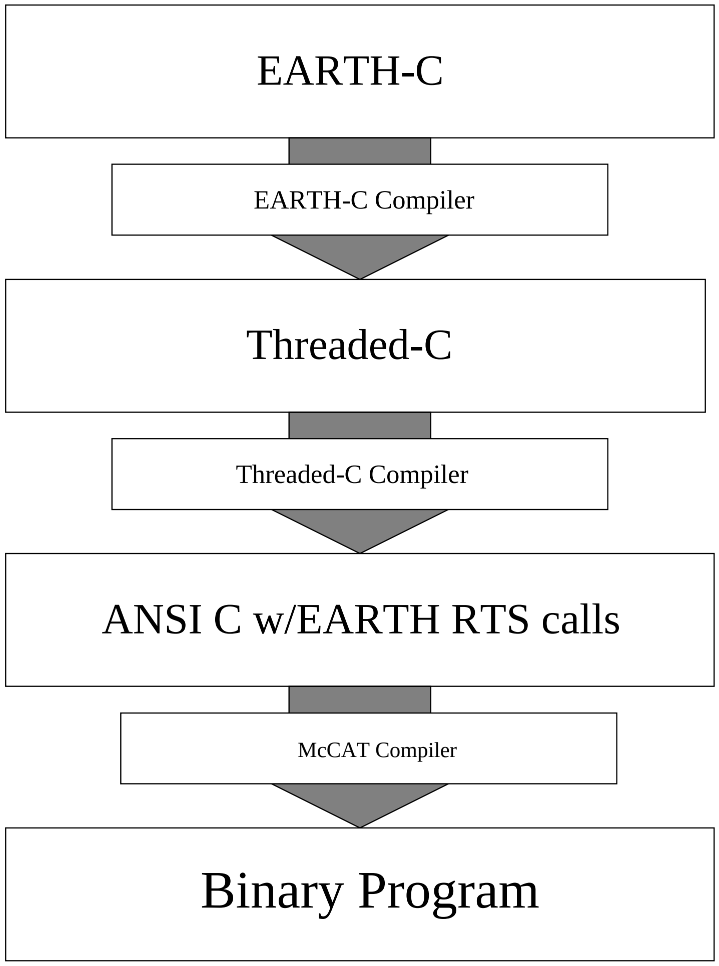

Figure 2.4: The EARTH compilation system.

primitive) contain information about both the memory location to be accessed, and which node contains the data being addressed.

2.1.2 Compilation and Execution on EARTH

The EARTH-C compiler is used to produce Threaded-C code from EARTH-C source code. The resulting Threaded-C code is run through the Threaded-C preprocessor, which translates Threaded-C into ANSI C with calls to the EARTH runtime system (RTS), which handles thread creation and management as well as communication (data requests and synchronization) while the program is executing. The system C compiler (e.g. `gcc`) is then used to compile this ANSI C code into an executable binary file (see Figure 2.4).

When the program is invoked, the first thread of the `MAIN()` function begins executing on Node 0. More threads, local (running on the same node) or remote (running on a different node), may be spawned from this initial thread via calls to the EARTH RTS. Console output from threaded programs (e.g. output resulting from calls to `printf()`) is directed to a file in the current directory named etc.PROCESS.NODE_ID where PROCESS is the process number of the executing program and NODE_ID is the node number of the current node.

EARTH hardware is not yet available, but an emulation system has been constructed to study the implications of the EARTH model. This emulation system is implemented on a MANNA machine (EARTH-MANNA), a network of Sun Ultra-Sparcs running Solaris (EARTH-SMP), and a network of PCs running Linux/Beowulf (EARTHQuake). In these emulations, each machine on the network functions as one EARTH node.

2.2 The Need for Profiling

In order to evaluate the usefulness of the EARTH model of multithreading, it is necessary to examine the behavior of programs at runtime. The thread partitioning algorithms have a direct effect on the performance of programs in this environment. By partitioning the work to be done, the compiler effectively dictates the degree of parallelism (and therefore the degree of utilization of the EARTH system) occurring in a program.

Figure 2.5 shows a possible situation in which several threads (dark blocks) experience varying amounts of latency due to, for example, varying load on the EARTH system. Overlapping blocks illustrate situations in which there are threads running concurrently on different EARTH nodes, and gaps between the issuance of

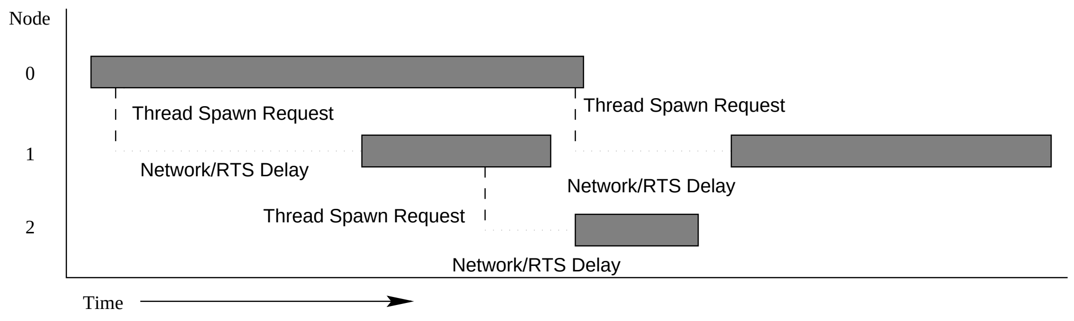

Figure 2.5: An example of thread distribution and network delay on the EARTH architecture.

a thread spawn request and the actual beginning of a block indicate network or RTS delay.

It is not efficient to simply partition the program into as many threads as absolutely possible, because the network used to connect EARTH nodes introduces considerable latency when a program accesses remote data. We must also consider the number of remote accesses to data and the size of the data transferred if we are to get a clear picture of the effectiveness of the multithreading system.

On a traditional sequential architecture, runtime information can often be obtained via a profiler; software packages are also available which allow the construction of customized profiling tools [23, 18]. None of these tools is compatible with the EARTH architecture and its extensions to the C language, however. Also, because the EARTH emulation system is currently available on several hardware platforms, it is desirable that the EARTH profiling tool be portable from one hardware (and operating system) platform to another.

2.3 Thesis Research Overview

Agrawal, Gao, and Pollock [2] have been examining optimizations for the EARTH-C compiler which require that analysis tools for EARTH programs be constructed. In this thesis, I present the design and implementation of a portable Threaded-C profiler designed to be used as part of their project. I also discribe the construction of a useful benchmarking application for EARTH and present insights into directions for future work in the area of profiling to evaluate the EARTH system and compiler optimizations.

Chapter 3

A BENCHMARKING APPLICATION FOR EARTH

3.1 Motivation

In order to identify areas needing improvement in the EARTH compilers, it is neccessary to have a selection of EARTH-C or Threaded-C programs to compile and use to gather information about the code output by the compiler.

A small collection of such programs already exists [17]; however, I also implemented a parallel benchmark based on the CTSP originally described by Amaral and Gosh, which has the potential to be particularly useful for benchmarking various loads on the multithreaded system [4].

3.2 The Traveling Salesman Problem

The Traveling Salesman Problem is a problem often considered by computer scientists in which one salesman must visit some number N of cities on a business trip. The general idea is to find the shortest route which the salesman can take that will visit all of the cities and return him to his city of origin. In general, this problem is considered to be NP-complete [8, 11].

3.2.1 The Contemporaneous TSP

The Contemporaneous TSP (CTSP) [4] is a modification of the classic TSP problem such that the salesman's schedule now includes cities in several discrete geographic regions (states or countries); the salesman is restricted to entering and leaving each state only once.

The state-to-state path is specified in advance, and cities are generated randomly according to a Gaussian distribution with a user-specified mean and standard deviation. It is assumed that the transition between states is made at the midpoint of the border between the two states.

3.3 The EARTH Contemporaneous TSP

The EARTH Contemporaneous TSP (ECTSP) is an adaptation of the CTSP to the EARTH multithreaded environment, and is coded in Threaded-C. The ECTSP represents the CTSP problem as a graph in 2-dimensional space where each city c_i is a point (x_i, y_i), and the distance $d_{i,i+1}$ between cities c_i and c_{i+1} is given by the standard distance formula:

$$d_{i,i+1} = \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2} \tag{3.1}$$

It is possible to consider that the solution to the problem is an ordering of the cities $c_0 \ldots c_{N-1}$ such that the total distance D given by Eqn. 3.2 is minimized. The path sought is thus one of the $N!$ possible orderings of the cities.

$$D = \sum_{i=0}^{N} d_{i,i+1} \tag{3.2}$$

Because the entrance and exit points for each state are known in advance to be the midpoints of the states' borders, and because the salesman must enter each state only once, the path in each state can be computed independently of the paths in all other states.

3.3.1 The Greedy Algorithm

The Greedy algoritm is the simpler of the two algorithms used in the ECTSP; it is not guaranteed to identify a shortest path. The algorithm is outlined in Figure 3.1 and operates in $O(n * s)$ time, where n is the number of cities in a state and s is the number of states in the tour.

1. Move to the city closest to the midpoint of the state's incoming border.
2. Identify the city closest to the current city and move to it.
3. Repeat Step 2 until all cities in the state have been visited.
4. Move to the midpoint of the state's outgoing border.

Figure 3.1: The Steps in the ECTSP Greedy Algorithm.

3.3.2 The Branch-and-Bound Algorithm

The Branch-and-Bound algorithm is a more sophisticated algorithm which is guaranteed to find a shortest path through a state. It attempts to reduce the number of permutations which must be searched by eliminating groups of paths which are longer that some previously-determined path. The implementation of the ECTSP Branch-and-Bound algorithm is adapted from Way [25] and illustrated in Figure 3.2. Here `P` is the permutation under consideration, `P[c]` is the `c`th city in `P`, `i` is a state variable used to track the path under consideration, and `S` is the shortest path found so far.

3.4 Implementation

The ECTSP first executes the Greedy algorithm to determine a possible shortest path in each state, and then runs the Branch-and-Bound algorithm, using the path identified in the Greedy algorithm as the initial shortest path. In both algorithms, each state is searched in a separate thread. The node on which a thread executes is determined by the EARTH RTS built-in load-balancing system. Each state thread executes several `BLKMOVE_RSYNC`s to obtain local copies of needed data structures (states, cities, and borders), runs the appropriate algorithm, and executes another `BLKMOVE_RSYNC` [1] to return the identified path to Node 0.

[1] The `BLKMOV_RSYNC` EARTH primitive is used to access data on a (possibly) remote node.

3.5 Summary

The ECTSP was useful as a study of the Threaded-C language and a beginning point from which to consider the profiling issues introduced by the EARTH model of multithreading. The ECTSP is also useful as a flexible benchmark for the EARTH system, because the load it produces on a system varies greatly according to the input TSP parameters.

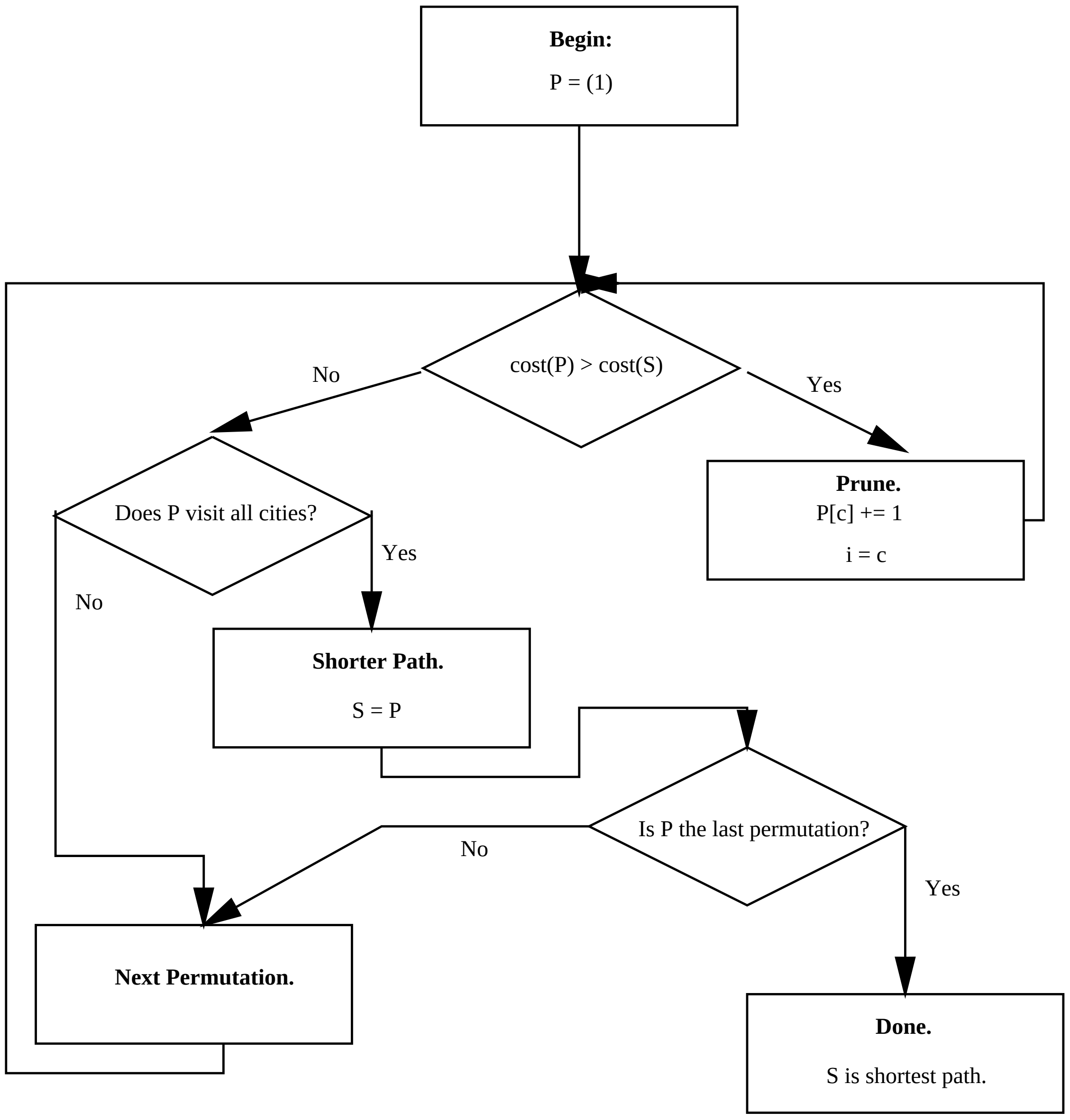

Figure 3.2: The ECTSP Branch-and-Bound Algorithm

Chapter 4

DESIGN OF A PROFILING SYSTEM

4.1 Overall Design

The EARTH Threaded-C profiling system is designed to gather information about the EARTH events which occur when a parallel program is run on an EARTH system. While not all possible events are currently profiled, it is desirable to design the profiling system to be easily *extensible* to gather new information, and to be *flexible* so that gathered information can be used for several different types of analysis and displayed in several different formats.

The EARTHQuake (Linux/Beowulf) implementation of EARTH was chosen as the host platform for profiler development because it was the most stable at the beginning of the project. It had originally been hoped that the profiler could be implemented with an already-available instrumentation or profiling package such as ATOM [23] or EEL [18], but neither of these were available for the Intel i386 architecture of EARTHQuake; porting either to i386 was also judged to be extremely nontrivial. One reason for this is the necessary adaptation to the variable-length instruction word of a CISC i386 machine, which would require significant redesign. Because a profiler designed on EARTHQuake should also eventually be portable to the EARTH-SMP or another architecture, it was decided that any profiling strategy involving binary code manipulation was inappropriate.

This decision requires the code to be instrumented *before* compilation is complete. It is also desirable that the burden of implementation not be placed on the user of the profiling tool. One solution which meets all these criteria is to modify the EARTH compilers to instrument code, gather data at runtime, and construct an analysis tool to analyze this data after runtime. Because the lexical and syntactical elements of the EARTH compilers are generally portable, any profiling modifications made in those elements is likely to be portable as well.

It was decided to implement profiling at the Threaded-C level rather than the EARTH-C level because EARTH-C is translated into Threaded-C before compilation, so a Threaded-C implementation allows us to capture information about EARTH-C as well, without implementing two separate systems. This system also gives the programmer a modicum of control over which functions will be profiled by profiling only those source files which contain functions to be profiled.

The system consists of two tools: an instrumentation program which inserts profiling output calls into a Threaded-C application before compilation, and an analysis tool which analyzes the resulting output to create a profile of the program. The instrumentation tool, `tcinst` (Threaded-C INSTrumenter), is written in PERL, while the analyzer, `tcprof` (Threaded-C PROFiler), is written in standard C++. The interaction between these tools and the rest of the EARTH system is illustrated in Figure 4.1.

Because both applications are written in standard languages found on most computer platforms today, it seems likely that they will be portable to other platforms in the future. Also, because they are implemented as standalone applications in common computer languages, it is likely that the profiling system will be more easily extended to meet future design goals than a similar system implemented, for example, inside the EARTH-C or Threaded-C compilers, which are much larger projects with much steeper learning curves.

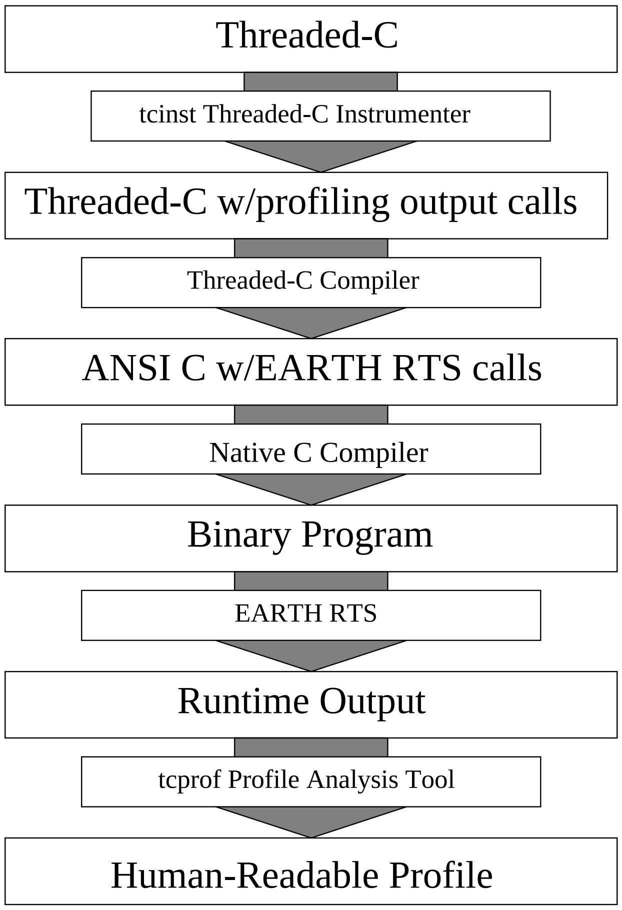

Figure 4.1: The profiling system components.

4.2 Events to profile

Because this profiling system was built for use by compiler writers, EARTH-C programmers, and Threaded-C programmers, the events to be examined by the profiler were specified for these targeted users. These events include:

- The number of threads executed on each node and dependencies between these threads
- The number of block moves (i.e., the number of times a consecutive block of memory is accessed from a possibly remote node)

EARTH Primitive	**Type**
`THREADED`	Thread Boundary Marker
`THREAD_`X	Thread Boundary Marker
`END_THREAD()`	Thread Boundary Marker
`RETURN()`	Thread Boundary Marker
`INIT_SYNC()`	Syncronization
`SYNC()`	Syncronization
`RSYNC()`	Syncronization
`BLKMOV_SYNC()`	Remote Data Access
`BLKMOV_RSYNC()`	Remote Data Access
`GET_SYNC_`X	Remote Data Access
`GET_RSYNC_`X	Remote Data Access
`DATA_SYNC_`X	Remote Data Access
`DATA_RSYNC_`X	Remote Data Access
`TO_LOCAL`	Pointer Cast
`TO_GLOBAL`	Pointer Cast

Table 4.1: The EARTH primitives examined by the profiling system.

- The number of remote and local loads and stores
- Value profiling of pointer variable values (to be used in determining which "may alias" pairs computed by the compiler are actually aliased at runtime)

Information about threads and dependencies will be useful for improving the thread partitioning algorithms used in the compilers. Loads, stores, and blockmoves are significant because of the latency introduced by the EARTH RTS and the underlying physical network; by optimizing for fewer remote operations, we can improve the performance of an EARTH-C or Threaded-C application. Information about aliases will allow us to perform standard alias optimizations and to study the effects of aliasing on the EARTH model.

The specific EARTH primitives examined by the system are listed in Table 4.1, and can generally be grouped into four classes: Thread Boundary Markers, Syncronization calls, Remote Data Access calls, and Pointer Casts. Thread Boundary

Markers identify the beginnings and ends of threads, and are profiled to provide information on how many threads are executing and which profiled events occur in which threads. Synchronization calls are those used to manipulate the EARTH synchronization unit and enable threads for execution. Remote Data Access calls allow a thread to access (load or store) data on a remote node, and also function as Synchronization calls by sending synchronization signals when they have finished the remote operation. Pointer Casts are used to convert local addresses into addresses on a remote node (or vice versa), and are often used to specify data locations accessed via the Remote Data Access primitives. For more information on each of these primitives, the reader is referred to [24].

4.3 The Instrumentation Tool

The `tcinst` instrumentation tool is written in PERL and makes extensive use of the PERL regular expression facilities to identify Threaded-C primitives. `tcinst` reads a line of source code at a time from a specified file (or `stdin`) and scans for strings which match those of the EARTH primitives specified in Figure 4.1.

When a primitive is found, `tcinst` inserts a `printf()` call which outputs the type of primitive found and the arguments given to the primitive in the original call for use in analysis. `tcinst` then adjusts the line numbers (via the standard C `#line` preprocessor directive) in the profile-enabled source code to match those in the unprofiled code.

With the execption of thread beginning primitives (`THREAD_`X and the `THREADED` keyword), these profiling output calls are inserted *just before* the call in the original Threaded-C code to insure that the values of arguments do not change between the printing of the profiling information and the execution of the EARTH primitive. The calls for thread beginning primitives are located after the calls and any related variable declarations, because ANSI C requires that all variable declarations occur before function calls.

Each profile output statement begins with a *profile header* containing a tag marking the line as a profile statement, the current filename, line number, function name and thread number containing the statement being profiled. These output statements are then analyzed with the analysis tool described below.

4.4 The Analysis Tool

As discussed in Section 2.1.2, the EARTH RTS redirects standard output from each node to a file in the current directory. Profiling information is viewed and analyzed with the help of the profiling analysis tool `tcprof` (Threaded-C PROFiler).

4.4.1 Design

`tcprof` is written in C++ and uses a different C++ class to represent each type of profilable statement. The `thread` class, for example, contains a list of profilable events which happened during a run of that EARTH thread. A list of available classes is given in Table 4.2.

Figure 4.2 gives a general overview of the functioning of the `tcprof` analysis tool. This tool uses `flex` to scan through the output of the program to be analyzed and search for profiling output statements (statements beginning with the profile header). Tokens identified by `flex` are then passed to a syntactic analysis unit built using `bison` [10] which identifies the EARTH primitive which this output statement profiles and instantiates a member of the appropriate `event` class to hold the information gathered.

4.4.2 Analysis

Analysis in `tcprof` is implemented via the `analysis` class. All classes which inherit from `event` have an `analyze()` function which accepts a reference to an

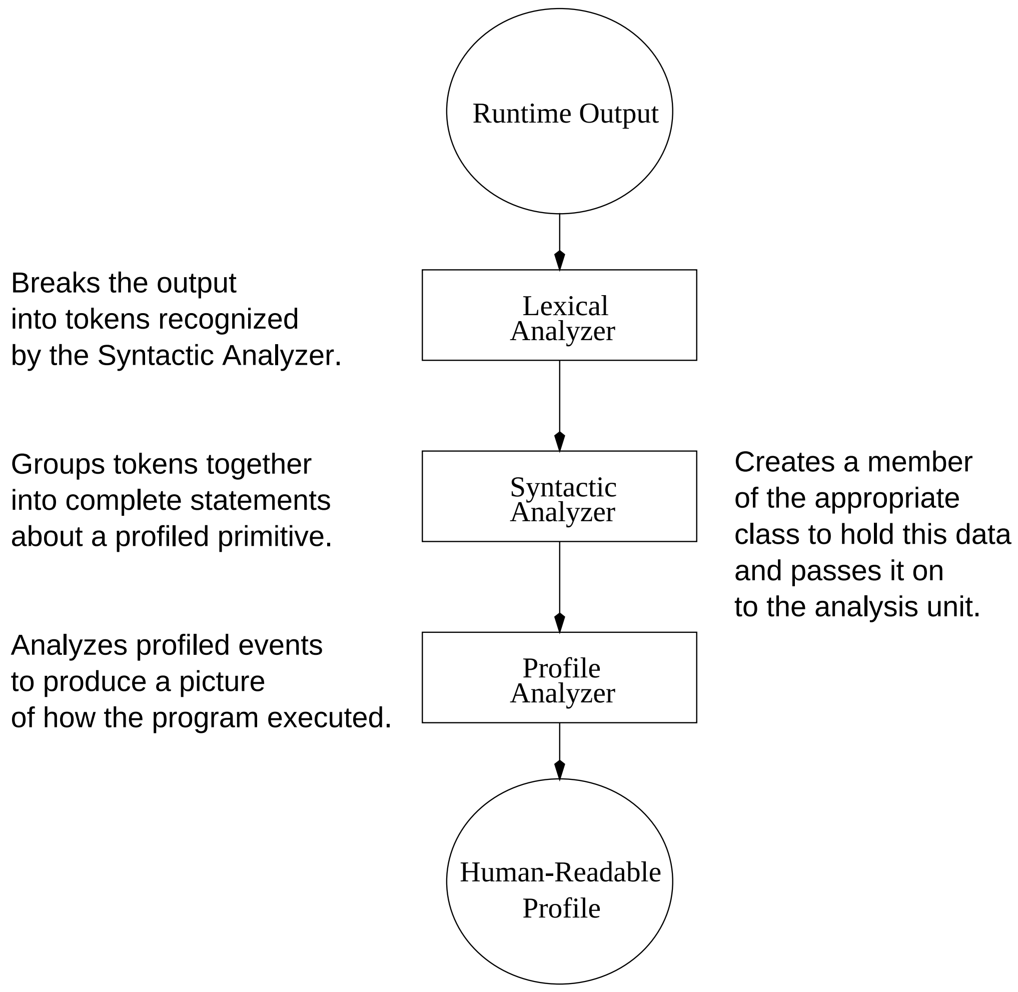
Runtime Output
Breaks the output into tokens recognized by the Syntactic Analyzer.
Lexical Analyzer
Groups tokens together into complete statements about a profiled primitive.
Syntactic Analyzer
Creates a member of the appropriate class to hold this data and passes it on to the analysis unit.
Analyzes profiled events to produce a picture of how the program executed.
Profile Analyzer
Human-Readable Profile

Figure 4.2: Internal functioning of the `tcprof` analysis tool.

Class Name	Description
`event`	Root of the event heirarchy; all classes below inherit from this class.
`function`	Container class to group threads from the same threaded function. (Currently not used).
`thread`	An instance (execution) of a thread. Contains a list of `events` which occurred during the run of this instance of this thread.
`blockmove_local_sync`	A single `BLKMOV_SYNC` call.
`blockmove_remote_sync`	A single `BLKMOV_RSYNC` call.
`local_sync`	A single `SYNC` call.
`remote_sync`	A a single `RSYNC` call.
`get_local_sync`	A single `GET_SYNC_`X call.
`get_remote_sync`	A single `GET_RSYNC_`X call.
`init_sync`	An `INIT_SYNC` call.
`to_local`	A cast from the `GLOBAL` pointer type to a local pointer type.
`to_global`	A cast from a local pointer type to the `GLOBAL` pointer type.
`data_local_sync`	A single `DATA_SYNC_`X call.
`data_remote_sync`	A single `DATA_RSYNC_`X call.

Table 4.2: The C++ classes used to represent EARTH primitives in `tcprof`.

`analysis` object and modifies that object to reflect the information gathered by profiling.

Currently, analysis consists of simply counting the number of times each event occurs when the program is run. The events can then be printed in a simple indented call graph format.

Chapter 5

USE OF THE PROFILING TOOL AND FUTURE DIRECTIONS

5.1 A Profiling Study

This section describes a short study conducted with the Threaded-C profiling system on sample Threaded-C programs. These applications (with the exception of the ECTSP, described in 3) are example programs distributed with the EARTH system or are from the EARTH Benchmark Suite described in [17]. Application sizes are given in lines of source code before instrumentation.

Program	**Size**	**# Threads**	**Remote Data Accesses**	**Pointer Casts**
`fib1`	57	267	177	355
`fib2`	54	63	41	42
`hello2`	49	0	3	0
`matrix2`	252	9298	0	16

Table 5.1: Sample information gathered via `tcinst` and `tcprof`.

5.2 Summary

This thesis has described profiling issues introduced by the EARTH model of multithreading, and described the implementation of a flexible source-level profiling system for use with Threaded-C on EARTH.

Major contributions include:

- Implemented a benchmarking program in Threaded-C.
- Designed and implemented a source-level Threaded-C instrumentation utility in PERL.
- Designed and implemented a flexible, extensible profile analysis tool in C++ which analyzes the profile information collected by a program instrumented with the above instrumentation tool.
- Demonstrated the usefulness of the profiling system.

5.3 Future Directions

Much of the information gathered by the profiling system is not yet used in the analysis phase (see Section 4.1). This information should be considered and used to build more comprehensive profiles of applications, so that these profiles can be used to guide improvements to the EARTH compilation systems.

The system attempts to gather profiling information about pointer variable values which are used as arguments in EARTH primitives. This information could eventually be used to examine aliasing in order to improve the compiler's handling of "may-alias" pairs.

I have also tried to collect information which can be used to determine the data dependencies between running threads by gathering data about `INIT_SYNC` and the various synchronization calls (including global pointer values for those syncronization events which use them). This information could be used to build a runtime thread dependency graph, for example.

Because the `tcinst` instrumentation tool is currently unable to recognize valid C statements which span more than one line, and because it is unable to recognize valid C expressions other than variable names or constants used as arguments to EARTH Threaded-C primitives, restrictions are placed on the textual format of the programs to be profiled. These restrictions are onerous, as they require the user of the profiling system to visually inspect each line of source code in the application to be profiled. More development work on `tcinst` should be done to address these inadequacies. One possible means of addressing these shortcomings could be the reimplementation of `tcinst` in C, using `flex` and `bison` to recognize Threaded-C syntax more effectively than can be accomplished with regular expressions in PERL. Much of the needed code (e.g. `bison` grammar descriptions) could be borrowed from the existant Threaded-C preproccessor.

While the current source-code instrumentation scheme may be useful for gathering counts, simple pointer analysis and thread dependency information, several interesting items of information are simply not available at this level. Low-level hardware instruction scheduling and instruction counting is obviously not available, for example. One option for such a scheme might be to adapt or construct a binary-editing tool such as ATOM or EEL for EARTH, but this is likely to be difficult to port across the several platforms used for EARTH development.

Another alternative may be available: adapt the GNU `gprof` profiler to understand Threaded-C and EARTH-C syntax. Because EARTH-C and Threaded-C are eventually compiled as ANSI C (see section 2.1.2 on page 18), and because the source code for `gprof` is readily available, it seems likely that the underlying C compiler and `gprof` could be adapted to profile EARTH applications at the binary level.

Appendix A

USING THE THREADED-C PROFILING SYSTEM

A.1 Instrumenting a Threaded-C Program

To instrument a Threaded-C source file, run the `tcinst` instrumentation utility on the Threaded-C file with the command:

`tcinst.pl` *filename.c*

This will produce an instrumented Threaded-C file named *filename_p.c* in the current working directory.

A.2 Compiling an Instrumented Source File

Instrumented source files can be compiled with the Threaded-C compiler `etcc` via:

`etcc` *filename_p.c*

A.3 Executing an Implemented Application

The resulting executable can then be run via `etc_run` as with all other Threaded-C applications:

`etc_run -n` *num_nodes executable args*

The compiled application will now run as usual, and will write profiling information to `stdout`.

A.4 Using Profile Information

Profiling output from the application is found mixed with normal (non-profiling) output in the files created for `stdout` by the EARTH RTS. All profiling statements are marked with `%PROFILER`, and thus profiling information can be removed from the file via

```
grep -v "%PROFILER" filename
```

This output is then parsed and analyzed by the Threaded-C profile analyzer `tcprof`. A simple script, `gen_report`, has been provided which performs the profile-output separation and analysis steps as one command line:

```
gen_report filename
```

Appendix B

A SAMPLE PROFILING SESSION

This appendix demonstrates a sample session instrumenting and profiling a Threaded-C application. The application used here is a variant of the classic "Hello, World!" program from computer science texts. This variation is stored in the file `hello2.c`.

B.1 Instrumentation

We instrument the code, which produces `hello_p.c`.

```
earthquake[501] [~]> cd examples/hello2
earthquake[502] [~/examples/hello2]> ls
Makefile     hello2.c     session.txt
earthquake[503] [~/examples/hello2]> tcinst.pl hello2.c
earthquake[504] [~/examples/hello2]> ls
Makefile     hello2.c     hello2_p.c session.txt
```

B.2 Compilation

Now we invoke the `etcc` Threaded-C compiler to create the `hello_p` executable file.

```
earthquake[505] [~/examples/hello2]> etcc hello2_p.c
earthquake[506] [~/examples/hello2]> ls
Makefile     hello2.c     hello2_p     hello2_p.c     session.txt
```

B.3 Running the Application

The `hello_p` binary is run via the EARTH `etc_run` RTS system. Its output can then be found it the `etc.21064.0` file.

```
earthquake[507] [~/examples/hello2]> etc_run -n1 hello2_p
Starting processes on remote nodes...done.
Initializing the network...done: 0 s
Executing user code...done.
earthquake[508] [~/examples/hello2]> ls
Makefile     etc.21064.0     hello2.c     hello2_p     hello2_p.c
session.txt
```

B.4 Profiling

Now we invoke the `tcprof` profile analysis tool via the `gen_report.sh` script to generate our report.

```
earthquake[509] [~/examples/hello2]> gen_report.sh etc.21064.0

Analysis:
0 BLKMOVE_RSYNC()s
0 BLKMOVE_SYNC()s
1 RSYNC()s
0 SYNC()s
0 TO_GLOBAL()s
0 TO_LOCAL()s
0 INIT_SYNC()s
0 GET_SYNC()s
0 GET_RSYNC()s
0 DATA_SYNC()s
0 DATA_RSYNC()s

A total of 3 threads ran on this node.
Call graph:
Thread: File: hello2.c Line: 35 Function: MAIN Thread: 0
Thread: File: hello2.c Line: 17 Function: print_hello Thread: 0
   RSYNC File: hello2.c Line: 19 Function: print_hello Thread: 0
Thread: File: hello2.c Line: 42 Function: MAIN Thread: 1

Done.
```

Appendix C

GLOSSARY

Aliases Two pointer variables which point to the same location in memory are said to be *aliased*. During compilation, a compiler may label a pair of pointer values as a "may alias" pair, indicating that the pair may be aliased at runtime, but it is impossible to guarantee that they will be aliased without further information.

`bison` A freely available `yacc` implementation.

Irregular Applications Computer applications in which data access patterns cannot be predicted before the application is executed.

`lex` A LEXical analysis program often used in compilers to break source code into lexical units (called *tokens*) like keywords, numbers, and punctuation.

`flex` A freely available `lex` implementation.

Regular Expressions Many programming packages which are designed for text processing (for example, `lex` and PERL) allow the programmer to specify how to identify specific strings via a set of consistent patterns called regular expressions.

PERL The Practical Extraction and Report Language. Originally designed to produce summary reports from large text files, PERL is a scripting language with built-in facilities for regular expression matching and string processing.

`printf()` The standard function in the C programming language for printing text on the system console.

Profiling A method of gathering information about how a computer program actually executes at runtime. This information often includes (for example) a list of which subprograms are called at runtime and in what order they are called.

`stdin,stdout` Programs written in the C language by default read input from a source called `stdin` (which is usually attached to the system keyboard) and write output to a sink called `stdout` (which is usually attached to the system console).

Value Profiling A profiling method which examines the runtime values taken on by variables in a program.

`yacc` Yet Another Compiler Compiler. A tool often used in compilers to perform syntactical analysis of source code files.

BIBLIOGRAPHY

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

[2] Gagan Agrawal, Guang Gao, and Lori Pollock. Compiling irregular applications on a multithreaded architecture. NSF (funded) Proposal, December 1997.

[3] 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.* ACM Press, June 1990.

[4] Josè Nelson Amaral and Joydeep Gosh. Automatic generation of versatile benchmarks for parallel production system architectures. Technical Report TR-PDS-1996-011, Department of Electrical and Computer Engineering, University of Texas, July 1996.

[5] Boon Seong Ang, Arvind, and Derik Chiou. Start: the next generation: Integrating global cache and dataflow architecture. Technical Report CSG Memo 354, MIT Laboratory for Computer Science, 1994.

[6] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. URSA: A unified resource allocator for registers and functional units in VLIW architectures. In *IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Orlando, Florida, January 1993.

[7] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *Micro-30*, December 1997.

[8] T. H. Corman, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[9] David Culler, Anurag Sah, Klaus Schauser, Thorsten von Eiken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler controlled threaded abstract machine. 1991.

[10] Charles Donnelly and Richard Stallman. Bison: The YACC-compatible parser generator. Free Software Foundation User's Manual, December 1992.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[12] S. Graham, P. Kessler, and M. McKusick. gprof: A call graph execution profiler. *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, 17(6):120–126, June 1982.

[13] S. Graham, P. Kessler, and M. McKusick. An execution profiler for modular programs. In *Software - Practice and Experience*, volume 13, pages 671–685, 1983.

[14] Rajiv Gupta, David A. Berson, and Jesse Z. Fang. Path profile guided partial dead code elimination using predication. *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 102–115, November 1997.

[15] Rajiv Gupta, David A. Berson, and Jesse Z. Fang. Path profile guided partial redudancy elimination using speculation. *International Conference on Computer Languages (ICCL)*, pages 230–239, May 1998.

[16] Laurie J. Hendren, Guang R. Gao, Xinan Tang, Yincun Zhu, Xun Xue, Haiyaing Cai, and Pierre Oullet. Compiling C for the EARTH multithreaded architecture. ACAPS Technical Memo 101, McGill University School of Computer Science ACAPS Laboratory, March 1996.

[17] Herbet H.J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Xinan Tang, Guang R. Gao, Phil Cupryk, Nasser Elmasriand Laurie J. Hendren, Alberto Jimenez, Shoba Krishnan, Andres Marquez, Shamir Merali, Shashank S. Nemawarkar, Prakash Panangaden, Xun Xue, and Yingchun Zhu. A design study of the EARTH multiprocessor. In *Proceedings of the 1995 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 59–68, Limassol, Cyprus, June 1995. International Federation for Information Processing.

[18] James R. Larus and Eric Schnarr. EEL: Machine independent executable editing. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 1995.

[19] Rishiyur Nikhil and Arvind. Can dataflow subsume Von Neumann computing. In *Proceedings of ISCA-16*, 1989.

[20] Rishiyur Nikhil, G. M. Papadopoulus, and Arvind. *t: A multi-threaded massively parallel architecture. In *Proceedings of ISCA-19*, 1992.

[21] S. Sakai, K. Okamoto, H. Matsuoka, H. Hirono, Y. Kodama, and M. Sato. Superthreading: Architectural and software mechanisms for optimizing parallel computation. In *Proceedings of the International Conference on Supercomputing*, 1993.

[22] Robert Sedgewick. *Algorithms in C++.* Addison-Wesley, 1992.

[23] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. Western Research Laboratory Research Report 94/2, Digital Equipment Corporation, March 1994.

[24] Kevin B. Theobald, José Nelson Amaral, Gerd Herber, Olivier Maquelin, Xinan Tang, and Guang R. Gao. Overview of the Threaded-C language. CAPSL Technical Memo 19, University of Delaware Department of Electrical and Computer Engineering Computer Architecture and Parallel Systems Laboratory, March 1998.

[25] Tom Way. Load balancing of a parallel branch-and-bound traveling salesperson problem solver. Unpublished Technical Report from Graduate Research Course, December 1996.