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

Design of the Runtime System for the Portable Threaded-C Language

Prasad Kakulavarapu†
Olivier Maquelin†
Guang R. Gao

CAPSL Technical Memo 24
July 21, 1998

†McGill University, Montréal, Québec, Canada

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

Abstract

Multithreaded programming helps in overcoming two major pitfalls of conventional parallel programming - *communication latency* and *synchronization overhead*. A **runtime system (RTS)** for multithreaded systems assumes the responsibility to provide an ideal interface between the multithreaded code and the hardware platform, and implements an environment for efficient execution of threads. The portability of the RTS may be enhanced by minimizing interaction between the RTS and the hardware/network platforms. Our study shows that an optimized RTS for the implementation of the multithreaded programming model is achievable by pursuing sound policies implemented with advanced programming methodologies.

EARTH is a multithreaded programming model developed at the ACAPS lab at McGill University. Further development of EARTH is currently underway both at the University of Delaware and McGill University. EARTH is implemented on many platforms, such as Sun workstations, Manna, IBM SP-2, IBM SP-3, and the Beowulf. Significant work has been directed into the evolution of a parallel multithreaded language for EARTH- Threaded-C. Recently the **Portable Threaded-C** was implemented over a portable RTS. The implementation of the portable RTS is the same for all platforms, except for the CPU and the network interface.

This paper describes the design and implementation of the EARTH portable runtime system. Design issues concerning the implementation of a distributed multithreaded environment such as the execution of non-pree mptive threads, context-switch, global memory, data communication and synchronization, and dynamic load balancers are addressed. The portable RTS on the IBM-SP platform with a tb-3 card as its network interface, is studied in this document.

Keywords

Multithreaded architectures, distributed memory, non-preemptive threads, call-graph, synchronization, scheduling.

Contents

List of Figures

1 Introduction

Fine-grain parallelism provides a reasonable solution for irregular applications with high demands on communication and synchronization. In addition to representing irregular applications (ex. paraffins), fine-grain parallelism performs well for diverse classes of applications like the SPMD (ex. matrix multiplication, Tomcatv), recursive (N-Queens, TSP), etc. by facilitating easy context-switching and an efficient runtime system. On architectures that support fast remote access and minimum context switching overhead, fine grain threads are an ideal choice for dynamic applications. Multithreaded languages efficiently manage the low computation to communication ratio (R/C) in fine-grain parallelism by switching to another *ready* (or enabled) thread, while servicing the communication requirements of another thread. Multithreaded systems supporting fine-grain threads can deal effectively with the two common bottlenecks for performance - small grain size, and barrier synchronization, by including good load balancing in their runtime systems.

Threaded-C is a multithreaded variant of C, and provides for definition, inter-thread communication, and synchronization of threads. Initially, Threaded-C was available only on the Manna platform. Implementation of the `EARTH` multithreaded model on Networks of Workstations, irrespective of their hardware/network platforms required minor design changes in Threaded-C resulting in the portable Threaded-C language. A detailed explanation of the portable Threaded-C programming model is provided in [5]. Currently, Portable Threaded-C is available on the Manna, IBM SP-2, IBM SP-3, Sun workstations, and the Beowulf system (a collection of LINUX Pentiums, connected with high speed ethernet).

At a lower level, the RTS acts as an interface between the language and the parallel machine. The runtime system (**RTS**) provides the environment for the execution of threads. Thread scheduling, context-switching, inter-node communication, synchronization, global memory management and dynamic load balancing are the core responsibilities of the RTS.

Execution patterns of threads forming a multithreaded computation are governed by the scheduling and load balancing policies of the RTS. The difference between scheduling and load balancing, and the resource pool for each of them are defined in section 4.1. Dynamic load balancing is implemented with seven policies and is presented in section 4.6. A detailed explanation of the dynamic load balancer policies, their implementation, and performance results are subject of another document [9].

The RTS was originally written for the Manna platform. Consequent to the design of the portable Threaded-C language, the RTS underwent comprehensive modifications. The portable RTS now supports multiple platforms like the IBM SP-2, IBM SP-3, Sun workstations and the Beowulf system. The portable RTS stresses minimum interactions with the hardware, as might be observed in minimum amount of assembly code and references to machine specifications. Unlike the platforms mentioned above, the RTS for Manna is not portable. However, portable Threaded-C programs can execute on the Manna version of the RTS.

This document aims to explain the strategies involved in the design of RTS for the portable

Threaded-C language. We study the the implementation of the RTS from the Threaded-C language - RTS boundary to the final execution of code. Initially, we look at the preprocessed code from Threaded-C to see the realization of Threaded-C constructs in the RTS. This is followed by a detailed explanation of the strategies adopted to implement the RTS functionality. This is supplemented by a running example of the Fibonacci code in Threaded-C. The hardware/network interface modules are touched upon with reference to the IBM SP-2 and tb-3 card combination.

The Threaded-C programming model is reviewed from the RTS point of view in section 2. Preprocessed code for a Threaded-C program is studied in section 3. The RTS is studied in section 4. Scheduling of threads is explained in section 4.1. RTS behavior while implementing two EARTH operations is traced in section 5. The RTS directory structure is shown in section 5.3. Compilation/execution information is provided in section 5.4. Performance results are documented in section 6. Section 7 ends the document with conclusion. The profiling support for the RTS is based on a set of profiling parameters that can be used to monitor the Threaded-C program execution. A description of these profiling parameters is provided in Appendix A. Timing information for various EARTH operations on the SP-2 platform is provided in Appendix B.

1.1 A Sample Compilation Sequence

A Threaded-C program is translated into sequential code with the `etcpre` preprocessor, and then linked with the RTS code to form the final executable. The various stages in the compilation sequence are invoked by the `etcc` driver (similar to the cc driver). The `etcc` driver first invokes the `xlc` compiler (IBM's C Compiler) with the preprocessing option to resolve header files and comments. The resulting file (with extension *cpp.c*) is given as input to the `etcpre` preprocessor. At this stage, the Threaded-C code is transformed into sequential C code. This C code is compiled with the C compiler `xlc` for object code. The object code corresponding to the application thus obtained, and is then linked with the RTS object code to arrive at the final executable.

A detailed listing of the options for the `etcc` driver is provided in the section 5.4. For the present, a sample compiling format for the Fibonacci program is shown below. Note that Threaded-C programs have `.c` extensions.

`etcc` -target rs6000-tb3 -prof -lb dual -O2 *fib.c* -o *fib.dual.prof*

This results in *fib.cpp.c* being created first. Next a temporary file with transformed C code is compiled to object code, and linked with the RTS object code to get the final executable as *fib.dual.prof*.

The `etcc` driver has various options for different target platforms in addition to traditional C compiler options, and invokes relevant components in correct order throughout the translation sequence. Threads are preprocessed into C functions with their parameters and local context stored in pointer accessible data structures. Threaded-C macros are replaced with code sequences that make calls to the RTS. Considerable optimization is performed at this preprocessing stage, which is reflected in minimum overhead for context switching and portability of the preprocessed code. The preprocessed code is compiled to object code, linked with the RTS object code as per command-line options options and executed with job submitting scripts.

The usage of `etcc` is presented with the `EARTH` web pages at `http://www-acaps.cs.mcgill.ca/EARTH/earth-manna/etcc.html`. The usage of the job submitting script on EARTH SP-3 is detailed in section 5.4.

2 Threaded-C

Threaded-C was originally conceived as an intermediate language for a higher level parallel language - the **EARTH-C** [3] as part of the EARTH project [2]. As the name suggests, Threaded-C extends the C language with threaded constructs. With gradual improvements, Threaded-C emerged as a programming language in its own right. It is *complete* in the sense that, it provides for constructs that fully capture the multithreading model, giving the programmer complete control of thread construction and thread launching.

Some salient features of the Threaded-C language are as follows:

- *Fine grain, non-preemptive* threads perform the multithreaded computation.
- Multithreaded computations may be composed from arbitrary call graphs.
- Rich semantics associated with the Threaded-C constructs support thread definition, parallel invocation, synchronization and inter-thread communication.
- Multiple threads can be launched simultaneously, either on the receipt of relevant data for which the threads are waiting (in a dataflow-like fashion), or by synchronization signals sent to the waiting thread by the programmer. Alternatively, threads may also be explicitly spawned.
- Applications execute in a global memory space comprising the local memories on all nodes in the system. Local and global pointers on Network of Workstations can be accessed in the programs.

Applications with dynamic behavior, involving high levels of communication pose significant challenge to multithreaded languages. Performance studies so far have shown significant speedup with Threaded-C. This better performance is attributed to the ability of Threaded-C to handle fine-grain parallelism and the close interaction between the Threaded-C compiler and the runtime system. An efficient thread scheduling policy, and a wide choice of dynamic load balancing algorithms have helped overcome the communication/synchronization latencies.

The multithreaded constructs of Threaded-C are explained from the RTS viewpoint in section 2.1. However a prior understanding of the Threaded-C programming model is required before proceeding further. A complete reference of the programming language is provided with [7] and the Threaded-C constructs are listed in [1].

2.1 Programming Model

Multithreaded computation on a NOW provides for temporal and spatial parallelism. The partial ordering of instructions, unlike in sequential programming, allows for considerable dynamic behavior. This dynamic schedule is governed by the RTS at run-time, while the language helps specify a call graph sequence at compile-time.

A **thread** is a set of instructions that is executed sequentially in an atomic fashion. Fine-grain, non-preemptive threads are the atomic units of multithreaded computation in EARTH. Interacting threads sharing context are grouped into bigger units - **threaded functions**. Every thread in a threaded function is numbered, with numbers starting from `0`. Execution of a threaded function always starts from thread `0`. Information regarding threaded functions are stored in activation frames. **Activation frames** are the data structures that hold the context of a threaded function, and are explained in section 3.4. With various threads of control (corresponding to various threaded functions) active at the same time, the activation frames are stored on heap rather than stack.

The **thread boundary** is a point in a threaded function where one thread finishes, and another starts. In Threaded-C, the producer and consumer parts of code are split into two threads, both of which are linked by synchronization slots. At the thread boundary, while the consumer thread is waiting for a sync signal, another *ready* thread is executed. A thread is ready when it has received a synchronization signal. This implicitly means that the thread has all the data required for its execution, and need not wait any further. Synchronization signals may or may not be preceded by any data that the thread may be waiting for. A thread is in the *wait* state when it is yet to receive some data required for its execution. A detailed explanation of the portable Threaded-C language model is given in [5, 7].

Parallelism is realized through threaded function calls. Invoking a threaded function has the result of the caller and callee functions executing concurrently. Execution of a multithreaded program with concurrent function invocations leads to dynamic unfolding of the computation and the call graph sequence is referred to as **activation tree**. This is similar to a directed acyclic graph (DAG) with the threaded functions as its nodes. The edges represent threaded function invocations (creation of child nodes) and data synchronization latencies. As in any concurrent execution, all the function frames are active, and the caller continues execution after invoking the callee. This is in contrast to the normal sequential function call mechanism, where the caller suspends until the callee returns.

Threaded-C provides for both *concurrent* (`INVOKE, TOKEN`) and *sequential* call mechanisms(`CALL`) for threaded functions. The syntax is described in [1]. Fig. 1 shows the concurrent function invocation in the Fibonacci program, and the corresponding activation tree

```
THREADED fib (SPTR done, int n, int *GLOBAL result)
{
    SLOT SYNC_SLOTS [1];
    int r1, r2;

    INIT_SYNC(0, 2, 2, 1);
    if (n < 2) {

        r1 = 1; r2 = 0;
        SPAWN(1);

    } else {
           INVOKE(0, fib, SLOT_ADR (0), n - 1, TO_GLOBAL (&r1));
           TOKEN(fib, SLOT_ADR (0), n - 2, TO_GLOBAL (&r2));
 } END_THREAD();

  THREAD_1:
    DATA_RSYNC_L (r1 + r2, result, done);
    END_FUNCTION ();
}
```

Figure 1: Parallel Function Invocation in Fibonacci Program

is shown in Fig. 2[1]. Observe that Threaded-C allows for normal sequential C function calls. These functions are allocated on stack. However, these C functions are semantically different from the threaded functions that are initiated with the sequential call mechanism (by using the `CALL` construct). The C functions are not allowed to have any Threaded-C constructs, except for the `POLL` primitive for network polling. The only similarity between the C functions and the threaded functions invoked with the `CALL` construct, is the manner in which they return. In both cases, the caller function is suspended until the callee returns. Also, when the callee returns, execution resumes from a point after the function call.

In Fig. 2, the rectangular blocks represent the threaded functions, and the inner circles represent the threads. The `TOKEN/INVOKE/CALL` edges (the `TIC` edge) are shown as thicker arrows when a parallel function is invoked from a thread in the caller function.

The dotted arrows in Fig. 2 are the *sync edges*, which represent the end of synchronization latencies of the waiting threads. On the receipt of a synchronization signal over the sync edge, the receiver thread is moved from the *wait* state to *ready* state by the RTS. Usually, the sending of the sync signal by the *producer* thread to the *consumer* thread, is preceded by placing some data in the address space of the consumer thread by the producer thread. It is also possible

[1]Normally, the first recursive call in Fig. 1 would also be issued with the `TOKEN` construct to ensure maximum dispersability of the tokens. In this example we are using the `INVOKE` construct to illustrate its use. In order to show all parallel constructs, an optimized version is not presented here.

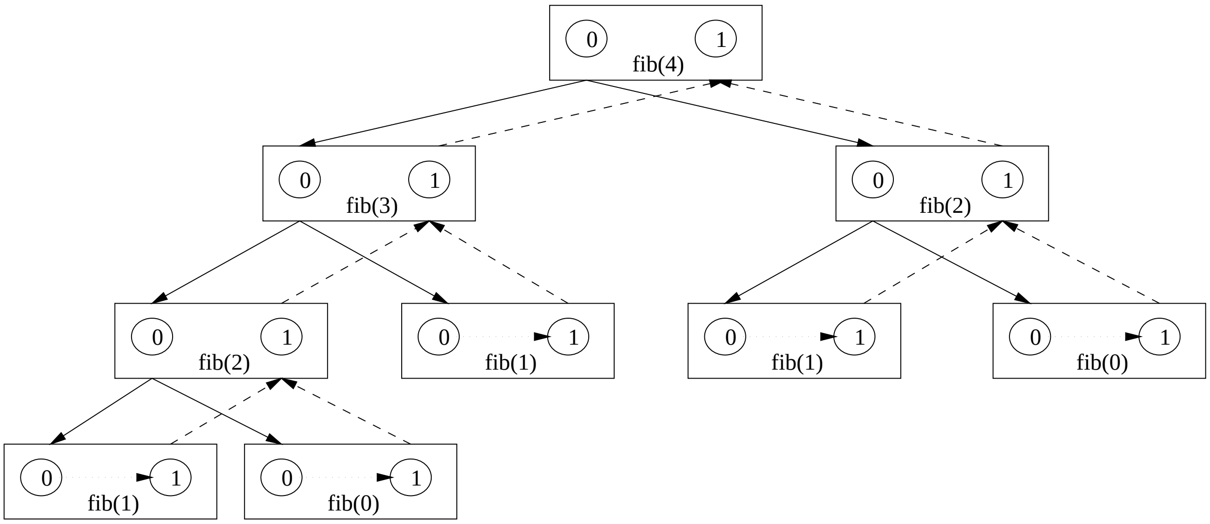

Figure 2: Activation Graph for Fib(4)

```
THREADED vadd (SPTR done, int N, float *GLOBAL a,
    float *GLOBAL b, float *GLOBAL res)
{
      SLOT SYNC_SLOTS [2];
      int i;
      float la, lb;

      INIT_SYNC (0, 2, 2, 1);
      INIT_SYNC (1, 1, 1, 2);
      for (i = 0; i < N; i++) {
           GET_SYNC_F (a++, TO_GLOBAL (&la), 0);
           GET_SYNC_F (b++, TO_GLOBAL (&lb), 0);
           END_THREAD ();
THREAD_1:
           DATA_SYNC_F (la + lb, res++, 1);
           END_THREAD ();
THREAD_2:;
      }
      RSYNC (done);
      END_FUNCTION ();
}
```

Figure 3: Threaded-C version of Vector Addition

that there may be no data preceding the sync signal.

Fig. 4 shows the graph representation for a threaded function that uses the `SPAWN` construct.

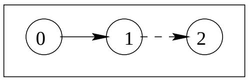

Figure 4: Activation Graph with a Spawn Construct

The thin arrow denotes the spawn edge, while the dotted arrow denotes a synchronization edge. The spawned thread is now *ready* for execution. No synchronization slots are involved in this case of explicit firing of a thread. In this figure, thread `0` spawns thread `1`, and thread `1` enables thread `2` by sending it a sync signal.

In some cases, threads in a function frame may be linked by data synchronization conditions. Here, the data-synchronization is not between the *parent* and *child* threaded functions, but between the threads of the same threaded function. For example, Fig. 3 shows a threaded code for vector addition. Thread `1` is waiting for data from thread `0`. In a way, these threads represent the producer-consumer relationship.

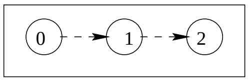

Figure 5: A node in Activation tree for Vector Addition

Fig. 5 shows a typical function frame for the vector addition (`vadd`) in one iteration. The dotted arrows represent the data synchronization edges. Another ready thread may be scheduled for execution at the thread boundary while the communication latency is being serviced. Observe that there are no TIC edges, but only sync edges.

The TIC edge may be seen as the passing of arguments from the parent node to the sub-computation represented by the child node. A TIC edge in the activation tree implies that the child node will not be invoked before the computation of its arguments. Thus we can state that Threaded-C is a **strict** language. When the sync edges from the child nodes end up only in their parent nodes, then the activation tree is **fully strict**. Threaded-C, in addition, supports back-sync arcs, and any arbitrary call sequence. Efficient execution schedules with bounds on time and space are possible for strict computations [11]. The computation in Fig. 2 is fully strict.

There is a constraint on the TIC edge when the child threaded function is instantiated with the `CALL` construct. In this case, the sync edge from the child node can be destined only to the thread in the parent node, that is numbered next to the thread which started the child computation with the `CALL` construct. For instance, in Fig. 2, if the child nodes were started with the `CALL` construct, then the sync edges have to return only to thread `1` in the parent node, whereas there is no such restriction with the `TOKEN` or `INVOKE`. They could have been sent to another thread, say thread `2`, if desired. This is because, after the `CALL` construct was used to instantiate a child threaded function, the parent threaded function stalls. It resumes execution from the point after the function call, only after the child threaded function returns. However, threaded functions started with `TOKEN` or `INVOKE` do not return any values, and therefore, can execute concurrently with the parent threaded functions. These child computations can direct their sync edges to any thread in any threaded function, though generally they follow the *fully strict* property by sending a sync edge to some thread in their parent threaded function.

Threaded functions instantiated with the `TOKEN` construct represent a threaded function whose host node is decided by the dynamic load balancer at runtime. The `INVOKE` construct specifies the node on which the function is to be executed. The thread `0` of the child threaded function is placed among other ready threads, for execution on the node specified. The `CALL` construct guarantees that the child threaded function will be scheduled on the same node as the parent node, and also will be immediately executed. Also, the caller is suspended until the callee returns. The `SPAWN` construct spawns a local thread. If a sequential C function is started, the C function is executed on the same node as the caller function, and also returns as per normal C convention. The frames for the callee and any further nesting are allotted on the stack.

Threads may stall when expecting results from a subcomputation or during communication latency while waiting for data. Any arbitrary call sequence may be launched by manipulating the *sync slots* (or the synchronization slots). Any synchronization, whether related to computation or communication, is performed through the sync slots. All threads that are to be enabled with the sync edges, except for thread `0` which is by default the *starting thread* in a threaded function, are associated with a sync slot by the statement `INIT_SYNC(slot_num, cnt, rst, th_no)`. Another exception to this rule are the threads that may be explicitly spawned, using the `SPAWN` construct. The value `cnt` indicates the number of sync signals, the thread `th_no` is awaiting, before getting fired. As each data unit is received, the `cnt` value is decremented, and the thread is *ready* for execution when it reaches zero. The `cnt` value is reset to `rst`, and the thread is placed in the queue for scheduling.

The declaration for the sync slots is expected to be the first statement in a threaded function with more than one thread. The RTS keeps track of threads to be scheduled through sync slots. The sync slots along with local variables comprise the activation frame for every function, which is made accessible to all threads in the function.

The RTS view of the Fibonacci threaded function is shown in Fig. 6. Thread `0` launches the subcomputations, while thread `1` is waiting for two sync signals. The **activation frame** comprises of the arguments for this threaded function, sync slots and local variables as declared in the beginning of the threaded function. The **context** for a threaded function is contained in the activation frame. The sync slots associated with their corresponding threads, help the RTS keep track of the scheduling status of those threads. The context within each threaded function is passed on among the ready threads of the same threaded function. Thread `0` initiates the activation frame and passes it along to the next *ready* thread, thereby preserving the context during context-switching within the threaded function at thread boundary. All the activation frames corresponding to threaded functions are allotted on the heap, while those of sequential functions are allotted on stack. From Fig. 6, it may be seen that arrival of results from child

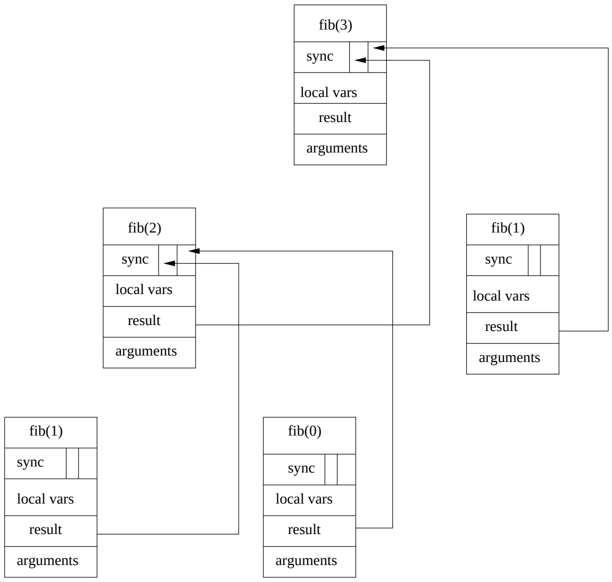

Figure 6: Runtime System's view with Activation Frames for fib(3)

computations signal the corresponding sync slot in the parent computation, which enables the associated thread for execution.

This section summarized the model of multithreaded computations in EARTH, and prepared ground for a detailed study of the strategies required to implement a multithreaded environment in the RTS.

3 Preprocessing Threaded-C

This section studies the preprocessed code for a Threaded-C version of the Fibonacci. The code generated contains sequential C equivalents of corresponding Threaded-C constructs. The preprocessed code for each Threaded-C construct can be divided into two parts. The first part sets the arguments for a later call to a RTS function. The second part actually calls a RTS function. The preprocessed code for individual Threaded-C constructs in the Fibonacci program are studied. Later, at the end of this section, all these pieces are added to obtain the total translation for a threaded function. The Threaded-C code for the Fibonacci function is shown in Fig. 1. The final translation for the threaded function is listed in Fig. 14.

It is important here that a distinction be made between the preprocessor (`etcpre`) and the preprocessed code. This section looks at the preprocessed code. The preprocessor itself is subject of another document, currently in preparation [6]. The abstract syntax tree for the `etcpre` is studied in detail in that document.

Threads from the Threaded-C code are transformed into functions in the `C` language along with necessary linkage code. The translation sequence for Threaded-C code is shown in Fig. 7.

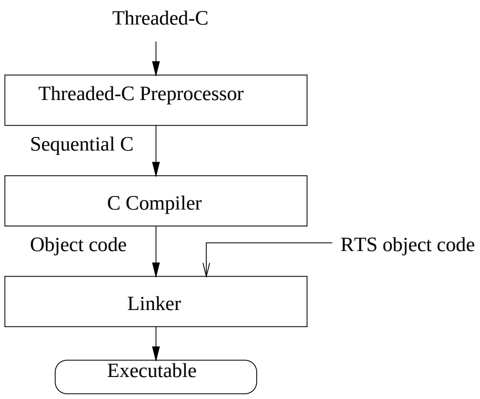

Figure 7: Translation Sequence of Threaded-C code

Further on, the term 'function' may be read as threaded function unless specified otherwise. The `etcc` driver initially invokes the Earth Threaded-C preprocessor (`etcpre`) on the input Threaded-C code. The preprocessed output is `C` code, with threads replaced by functions and relevant calls to the RTS. The Threaded-C constructs are transformed into equivalent C code according to their semantics. The next step of the `etcc` driver is to compile the preprocesssed output to object files through a C compiler. The resulting object code is linked with the RTS object code to obtain the final executable. The `etcc` driver checks the command line options, and accordingly invokes different modules. The options for the `etcc` driver are a superset of those of any commercial C compiler. The result of a successful use of `etcc` is the final executable. The syntax of `etcc` may be observed with the '`-h`' option. The use of `etcc` is explained in section 5.4. The preprocessed code for the Fibonacci threaded function is studied in this section.

A point worth mentioning before proceeding further is the allocation and disposal of heap memory. Before starting application execution, the RTS reserves some heap memory for a singly linked availability list, called the *free element list*, or the free list. All dynamic memory requirements of the RTS during application execution, are met from the free list. In case the free list becomes empty, the dynamic memory is obtained by using the `malloc` statement. After use, the memory is returned back to the free list. There is a free list in the address space of the RTS on every node. The contents of the free list on different nodes are not coherant.

3.1 Sync Slot

In Fig. 8, the current '`cnt`' value associated with the thread pointer indicates the number of synchronization signals required to enable the thread for a ready status. The '`rst`' field holds the reset count. The thread associated with this sync slot is represented by a combination of it's instruction pointer and frame pointer.

A thread with a non-zero sync count may be viewed as *consumer* thread (execution starts only after receipt of all relevant data). Usually, the *producer* (which may be another thread

```
typedef struct etc_slot {
          int cnt;
          int rst;
          etc_handler ip;
          long fp;
} etc_slot;
```

Figure 8: Type definition for Sync Slot

sending data, or results of a subcomputation), after depositing the data in some global location, for instance namely *result*, decrements the sync count of the consumer thread. Alternatively, the sync signal may not be preceded by any data. If the sync count reaches zero, the *ready* thread is ready for execution.

The RTS keeps track of threads from the information contained in the sync slots. It updates the status of the thread (either *wait* or *ready*), whenever a sync signal is received at the associated sync slot. For those threads which have become *ready*, the RTS fetches the *thread pointer* (a pointer to the C function representing the thread, and its frame pointer) from the structure for the sync slot, and places the thread among other ready threads for execution.

3.2 Global Addresses

Unlike the RTS for Manna implementation, portable Threaded-C has to accommodate 64 bit wide global pointers, thereby making the distinction between local and global pointers visible to the compiler [7]. Because the portable RTS has no access to the MMU, global pointers are accommodated at the software level. This is done by representing a global address as a combination of a node number and a local address. With the same executable running on all nodes, the global address is recognized by the particular node hosting the memory location. The data structure adopted is shown in Fig. 9.

```
typedef struct etc_gptr {
             void *ptr;
             int node;
} etc_gptr;
```

Figure 9: Type definition to represent Global Pointers

The preprocessed code for the Threaded-C construct `TO_GLOBAL` may be seen in Fig. 10. Local pointers are converted into global pointers by enclosing the node number and the local address as the two fields of a variable of type `etc_gptr` defined in Fig. 9.

```
_gptr1.node = etc_rts.node_id;
_gptr1.ptr = &_fp->r1;
```

Figure 10: Preprocessed code for TO_GLOBAL

3.3 SLOT_ADR

The structure for the sync slot is allocated from dynamic memory of the corresponding node. When the sync slots are to be globally accessed, the node number and the slot number form the global address of the sync slot. This is precisely the result of using the Threaded-C construct SLOT_ADR. Preprocessed code for this may be seen in the first two lines of part (b) in Fig. 13. For instance, while constructing the parameters for a subcomputation, the global address of sync slot is passed as the return sync pointer 'done', as may be seen in line 6 of part (b) in Fig. 13.

3.4 Frame based Data Structures

The context of a threaded function is stored in its *activation frame.* The activation frame contains local variables, arguments and any sync slots that may fire threads in the threaded function, as shown in part (a) of Fig. 11. The activation frame is passed among the C functions corresponding to individual threads in a threaded function to maintain the context within a threaded function. As may be seen in Fig. 11, the declarations for local variables r1, r2, and the sync slot (corresponding to Thread 1 in the threaded function), along with the argument variable declarations for n, done, and result, form the declaration for activation frame for that threaded function. The consumer thread information is stored in the sync slot. When the required number of sync signals have been received, the relevant thread is launched by accessing the *thread pointer* from the sync slot.

Another frame - the *parameter frame* holds the parameters passed to the threaded function. It is used mainly to pass parameters to the C function corresponding to thread 0 of a threaded function. A pointer to a structure similar to a parameter frame is passed as argument to the C function representing thread 0 (the first thread in a threaded function). Parameter frame is shown in part (b) of Fig. 11.

3.5 INIT_SYNC

The sync slots are initialized with the count and reset values, and are associated with a thread through the INIT_SYNC statement. Consider the preprocessed code for threaded macro INIT_SYNC in Fig. 13. Once the control enters a threaded function, and the activation frame is allocated on heap, the sync slot components are filled in with data pertaining to the relevant thread. This data includes current sync count, reset count, and the thread pointer.

(a) Activation Frame	(b) Parameter Frame

<pre>typedef struct {
        etc_slot _slots[1];
        SPTR done;
        int n;
        etc_gptr result;
        int r1, r2;
} _token_fib_F;</pre>
	<pre>typedef struct {
	        long _next, _prev, _ip, _fp;
	        SPTR done;
	        int n;
	        etc_gptr result;
	} _token_fib_P;</pre>

Figure 11: Structures generated for Frame-passing

3.6 SPAWN

The `SPAWN(1)` construct spawns locally thread `1` of the threaded function, that has the SPAWN construct. The preprocessed code is shown in Fig. 12. The frame pointer is converted into a global pointer. The *thread pointer* (instruction pointer and frame pointer) is passed as an argument to the RTS function `etc_spawn`. This spawns the thread `1` in the Fibonacci threaded function.

```
_gptr1.node = etc_rts.node_id;
_gptr1.ptr = _fp;
etc_spawn(_gptr1, _fib_1);
```

Figure 12: Preprocessed code for SPAWN(1)

3.7 INVOKE

The preprocessed code for the parallel construct `INVOKE` is shown in part (b) of Fig. 13. The arguments for this construct, the sync slot address and the destination location where the result is to be placed, are to be converted into global pointers. Next, as the number of user arguments in the INVOKE construct is not known at compile-time (the number varies depending on application), *variable parameter passing* is employed, by storing the parameters in an intermediate data structure - the *free element list* or the free list. Once the arguments for the INVOKE construct are set up, the RTS function `etc_invoke` is invoked. The preprocessed code is almost the same for `TOKEN` except for two changes. There is no argument for node number, and the RTS function called is `etc_token`.

Fig. 13, part (b) shows the preprocessed code for the `INVOKE(0, _fib, SLOT_ADR(0), n - 1, TO_GLOBAL(&r1))` construct. A variable of type pointer to parameter structure (`token_fib_pp`) of the threaded function that is going to be invoked is declared at the beginning of the `C` function representing the first thread of the caller threaded function. Memory for this variable is allocated on top of the free list. The arguments for the child threaded function are assigned to the fields of the parameter structure variable `token_fib_pp`. Those arguments

```
_fp->_slots[0].cnt = 2;
_fp->_slots[0].rst = 2;
_fp->_slots[0].ip = _token_fib_1;
_fp->_slots[0].fp = (long) _fp;
```

(a) INIT_SYNC(0,2,2,1)

```
_gptr1.node = etc_rts.node_id;
_gptr1.ptr = _fp->_slots + 0;

_gptr2.node = etc_rts.node_id;
_gptr2.ptr = &_fp->r1;

_fib_pp = (_fib_P *) etc_rts.next_free;
_fib_pp->done = _gptr1;
_fib_pp->n = _fp->n - 1;
_fib_pp->result = _gptr2;

etc_invoke (0, _fib_0, 24);
```

(b) INVOKE _fib function

Figure 13: Preprocessed code for INIT_SYNC and INVOKE

that represent addresses are converted into global pointers, and stored in two variables - `gptr1` and `gptr2`, which are then assigned to the fields of `_fib_pp`. Thus, with the arguments for the threaded function on top of the free list, the RTS function `etc_invoke` is called with the name of the first thread of the child threaded function (pointer to a `C` function), and the size of its arguments.

3.8 Frame Passing

The local variables of a threaded function are preserved along with their declarations in the activation frame. The context in a threaded function (arguments, sync slots and the local variables of a threaded function) is preserved in the activation frame. The activation frame shown in part (a) of Fig. 11, is created in the `C` function for thread `0` of a threaded function. At the beginning of this C function, a request for a 64 byte buffer element from the free list is made. If no element of that size exists in the free list, then the memory is obtained by using the malloc statement. The activation frame on the heap is deallocated and local variables are deleted from memory, when the `END_FUNCTION` primitive is executed.

The parameter frame is allocated outside a threaded function. The parameter frame structure is filled in with arguments for this threaded function, and a pointer to this structure is passed as an argument to the thread `0` of the threaded function. As explained in section 3.7, and later in section 3.9, the structure for the parameter frame is allocated on top of the free list.

Each one of the threads in a threaded function are transformed into a C function. The C function representing thread `0` of a threaded function receives the arguments for the threaded function, C functions representing other threads in the threaded function receive the activation frame. The key for identifying a thread is the combination called the *thread pointer - activation frame pointer* and the *instruction pointer*.

One way of invoking a thread at runtime, without knowing its name, is through the function pointer. The *instruction pointer* is used to call the thread by invoking the function pointer to the C function representing the thread. The arguments and the relevant context are obtained from the activation frame, which is pointed to by the frame pointer.

A *token* is the RTS representation of a threaded function. A token consists of two parts : the name of the C function corresponding to the first thread in a threaded function (instruction pointer), and the total size (number of bytes) of arguments for the threaded function. Remember that the execution of a threaded function starts from the thread `0` (first thread) in a threaded function. In a way, token is the handle meant for executing a threaded function. Heap memory for the activation frame has not yet been allocated and hence the tokens are free to migrate to remote nodes. If the token was created with the `TOKEN` construct, then the token may be used for load balancing and can be scheduled for execution on any node at runtime by the load balancer. On the other hand, if the token was created with the `INVOKE` construct, the token can be scheduled only on the node designated by the programmer. Once the threaded function is scheduled for execution on a node, i.e. guaranteed to execute on a node, memory is allocated for the frame pointer on that node, and the arguments for the threaded function are down-loaded into the parameter frame structure. The frame now contains the context of the threaded function, and may be passed onto remaining threads of the threaded function. The start of execution of a threaded function implies that heap memory is allocated for holding the activation frame. Thus the activation frame for a threaded function is *expanded*. Once the activation frame has been expanded, the threaded function cannot be executed on another node. Also, all the threads of the threaded function have to execute on the same node. Tokens, as units of load balancing, are discussed in section 4.1.

All threads forming a threaded function should be scheduled on the same node, because the structures holding the context are allocated from the heap memory of that node. Once the token representing an unexpanded Threaded-C function is expanded, the component threads form separately invocable C functions representing the non-preemptive threads. Context-switching between threads is made simple, by terminating one C function, and starting another C function. This manner of context-switching reduces overheads as it does not require any context (register file, program counter, stack pointer, status register) to be saved. Absence of such context savings improves portability, as the RTS code does not need to access any machine specific areas.

Now we describe the mechanism for sharing context (arguments and local variables of a threaded function) among the threads of a threaded function by referring to an example. The technique employed here is namely, the *frame-passing mechanism*. Fig. 14 shows the preprocessed code for the two threads of the Fibonacci threaded function shown in Fig. 1.

The structures holding the activation frame and the parameters (parameter frame) of a threaded function are named with the function name followed by a letter, either 'F' or 'P'. A pointer to the parameter frame is passed as argument to thread `0`. The remaining threads receive the frame pointer as argument. The context (activation frame) is passed between the threads of a threaded function. Memory is allocated for the frame pointer in the C function for the first thread (thread `0`), and its fields are filled with arguments for the threaded function, local variables, and the sync slots. The arguments are for the child threaded function, as specified in the `TOKEN`, `INVOKE`, or the `CALL` constructs. The local variables and the sync slots are obtained from the Threaded-C code for the child threaded function. The current frame pointer value is stored in the frame field (*fp*) of the sync slot that is associated with the second thread (thread `1`). When this sync slot receives a sync signal that changes status of thread `1` from *wait* state to *ready* state, the RTS uses the instruction pointer (C function pointer in the *ip* field of the sync slot) to invoke the C function `_fib_1`, and passes to it as argument, the frame pointer value from the *fp* field of the sync slot. Thus, context of a threaded function is passed between the C functions representing its threads. Remember that the threads of a threaded function are executed at different points on the time axis, though they all will execute on the same node.

3.9 Variable Parameter Passing

The number of arguments for a subcomputation may vary with each threaded function invocation, thereby requiring a parameter-passing mechanism for variable number of parameters. The parameters are set up before initiating the C representation of the `INVOKE`, `TOKEN` or the `CALL` construct. Initially, memory is allocated on the top of the free list for a variable that holds the parameter frame structure. The parameters are now loaded into this top element of the free list. The *thread pointer* containing both the frame pointer (pointer to activation frame) and the instruction pointer is used in initiating the first thread of a threaded function, in case the threaded function is scheduled for execution on the current node. In contrast, if the threaded function is scheduled on another node, a message composed of the instruction pointer and the parameters (from top of the free list) are sent to the destination node (seen in the function `etc_invoke` in the file `calls.c`).

3.10 Preprocessed Code for Fibonacci - Detailed Study

The preprocessed code for the Fibonacci threaded function from Fig. 1 is listed in Fig. 14. A typical activation tree for this code is shown in Fig. 2. The RTS view of this tree is depicted in Fig. 6.

A pointer to the parameter frame containing the arguments for the threaded function is passed as a parameter to the C function representing the first thread - `_fib_0`. The frame pointer is allocated and the relevant slots are filled in to store the local context. Next, the parameter frame structure, which holds arguments for subcomputations, is placed on the top of the free list. The parameters for the `INVOKE` statement are stored in the parameter structure on the top of the free list. Once the parameters are set up, the RTS function `etc_invoke` is called with the destination node for execution, instruction pointer, and the size of the parameter structure stored on top of the free list. The same procedure is repeated for the `TOKEN` statement as well. The C function for the second thread in the threaded function, `_fib_1` is passed the frame pointer as parameter, thus ensuring that the second thread has access to the context of the threaded function. A line by line study of the preprocessed code for the Fibonacci follows.

```
1: void _fib_0 (_fib_P *_pp)
2: {
3:    etc_gptr _gptr1, _gptr2;
4:    _fib_F *_fp;
5:    void _fib_1 ();
6:    _fib_P *_fib_pp;
7:    void _fib_0 ();
/* Obtain memory for frame pointer */
8:    _fp = (_fib_F *) etc_rts.free_64;
9:    if (_fp)
10: etc_rts.free_64 = _fp->_slots[0].cnt;
11:   else
12: _fp = (_fib_F *) malloc (64);
13:     _fp->done = _pp->done;
14:     _fp->n = _pp->n;
15:     _fp->result = _pp->result;
/* Insert code for INIT_SYNC(0, 2, 2, 1) */
16:     _fp->_slots[0].cnt = 2;
17:     _fp->_slots[0].rst = 2;
18:     _fp->_slots[0].ip = _fib_1;
19:     _fp->_slots[0].fp = (long) _fp;
20:     if (_fp->n < 2) {
21: _fp->r1 = 1;
22: _fp->r2 = 0;
/* Insert code for SPAWN(1) */
23: _gptr1.node = etc_rts.node_id;
24: _gptr1.ptr = _fp;
25: etc_spawn (_gptr1, _fib_1);
26:    } else {
/* Globalize the local variables
   syncs slot and r1              */
27: _gptr1.node = etc_rts.node_id;
28: _gptr1.ptr = _fp->_slots + 0;
29: _gptr2.node = etc_rts.node_id;
30: _gptr2.ptr = &_fp->r1;
/* Obtain memory for Parameter structure */
31: _fib_pp = (_fib_P *) etc_rts.next_free;
/* Load parameters for INVOKE onto
   the top of free element list   */
32: _fib_pp->done = _gptr1;
33: _fib_pp->n = _fp->n - 1;
34: _fib_pp->result = _gptr2;
/* Once the arguments are set up,
   call the RTS function          */
35: etc_invoke(0, _fib_0, 24);
/* Repeat same process as above for
   next TOKEN call                  */
36: _gptr1.node = etc_rts.node_id;
37: _gptr1.ptr = _fp->_slots + 0;
38: _gptr2.node = etc_rts.node_id;
39: _gptr2.ptr = &_fp->r2;
40: _fib_pp = (_fib_P *) etc_rts.next_free;
41: _fib_pp->done = _gptr1;
42: _fib_pp->n = _fp->n - 2;
43: _fib_pp->result = _gptr2;
44: etc_token (_fib_0, 24);
45:    }
/* End of code for Thread 0 */
46: }
```

(a) Thread 0

```
/* The local context of threaded function passed
   between threads through the frame pointer   */

1: void _fib_1 (_fib_F *_fp)
2: {
3:   etc_gptr _gptr1, _gptr2;


4:   etc_data_sync_l (_fp->r1 + _fp->r2,
                      _fp->result, _fp->done);

/* End of code for Thread 1 */
5: _end_fun:
6:   _fp->_slots[0].cnt = etc_rts.free_64;

/* Return space for frame pointer
   to free element list          */
7:   etc_rts.free_64 = (long) _fp;
8: }
```

(b) Thread 1

Figure 14: Frame passing among 2 threads of fibonacci function

line no : Comments for preprocessed code for Thread `0`*, shown in part (a) of Fig. 14*

line 1: The C function corresponding to the first thread in the threaded function is `_fib_0`. The arguments to the threaded function are placed in a parameter frame structure, and a pointer to this structure is passed as argument to `_fib_0`.

line 3: A maximum of two global addresses are used in any statement in the threaded function `fib`. So, two dummy declarations for global pointers are made to accomodate remote references.

line 4: A variable is declared to hold the frame pointer. Please refer to Fig. 11 to view the type definition of activation and parameter frames.

line 6: A variable is declared to hold the parameter frame pointer. This holds the parameters for subcomputations (used in *variable parameter passing*).

line 8,12: Memory for the frame pointer allocated from the free list. If memory is not available on the free list, dynamic memory is obtained by explicit use of the `malloc` statement.

line 13,15: The arguments to the threaded function are copied into the frame pointer structure from the parameter frame structure. Maintaining the values for arguments in the activation frame is required for context sharing.

line 16-19: This code corresponds to the `INIT_SYNC` construct from the threaded function `fib` in Fig. 1. The preprocessed code may be seen in part (a) of Fig. 13. From the parameters of the `INIT_SYNC` construct in `fib`, the values for sync count, reset count are filled into the sync slot structure for the thread in the actiation frame. The thread pointer is obtained from the thread, with which this sync slot is associated.

line 20: If `n<2`, then send data to the global pointer in `result`, and decrement the sync slot `done` (both obtained from the activation frame).

line 23-25: The local thread `1` (`_fib_1`) is to be spawned here. The frame pointer is made a global pointer, and passed along with the function pointer for `_fib_1` to the RTS function.

line 27-35: This C code corresponds to the `TOKEN` statement. Initially, the location for storing the result and the relevant sync slot are to be converted into global addresses. From the declaration of the frame pointer, we can see that sync slot `0` and the local variable `r1` are not global pointers. Note the `SLOT_ADR` and the `TO_GLOBAL` prefixes to the sync slot `0` and the result location `r1` in the `TOKEN` statement in the threaded function. Lines 27-30 perform this function.

line 31: The arguments for `TOKEN` may be any in number. An intermediate buffer area is required to hold the parameters, as its not possible to assign any specific data structure to hold the parameters, without knowing their number and type. So, the top element of the free list is type casted to the parameter frame type.

Lines 32-34 correspond to filling in the values for the newly declared parameter frame structure on top of the free list. In this case, the arguments for the `TOKEN` construct - the sync slot `0`, the value `n-1`, and the result location `r1`, are loaded onto the top of the free list.

Now, the setup for the TOKEN construct is complete. The RTS function, etc_token is called with the instruction pointer, and the size of the parameters on top of the free list. The destination node for execution is determined by the RTS at runtime, through dynamic load balancing. The only difference between the INVOKE and TOKEN is that, the programmer decides the node for the execution of a threaded function in the case of the INVOKE.

line 36-44: The same procedure as above, is repeated for the next TOKEN construct. Note that, the destination node number is not specified in the final RTS call etc_token.

line 46: The code for _fib_0 is complete now, and the function returns.

line no: *Comments for preprocessed code for Thread* 1, *shown in part (b) of Fig. 14*

From the lines 16-19 in the C function _fib_0, the activation frame has the sync slot information corresponding to thread 1. When the required number of sync signals have been received at the sync slot, the RTS obtains thread pointer for Thread 1 from the sync slot, and invokes the C function corresponding to Thread 1, by using the instruction pointer (a function pointer in C), and passing it the frame pointer as argument.

line 1: A pointer to the frame pointer, which has the local context for this threaded function, is passed as parameter to the C function representing thread 1 (_fib_1). This frame pointer is allocated from the dynamic memory of the node on which the threaded function is scheuled, and hence the restriction that all threads belonging to a threaded function should be scheduled on the same node.

line 4: The sum is placed in the location result and the relevant sync slot (done) decremented, within the data_sync function call.

line 5: This statement marks the end of code for thread 1.

line 7: The frame pointer is returned (deallocated) to the free list.

The threads of a threaded function may execute at different times with many threads belonging to other threaded functions executing in the intervening time. To understand the manner in which the context (stored in an activation frame for a threaded function) is shared among all the threads of a threaded function, the role of sync slots is to be remembered. As seen in part (a) of Fig. 14, the initial section of _fib_0 is spent in initializing the frame pointer, and the thread pointer (instruction pointer and frame pointer) is assigned to the relevant fields of the sync slot. When the thread associated with this sync slot is ready for execution, the RTS function (inline_hdl_sync in the file data_inc.c) schedules the enabled thread for execution. The ready thread is executed by using the instruction pointer and the frame pointer obtained from the structure for sync slot associated with this thread. The instruction pointer identifies the executable code for the corresponding C function, while the frame pointer points to the context of the threaded function in which this thread should execute.

3.11 Sequential-Call mechanism with CALL

The discussion on frame-passing mechanism for threaded functions so far, has dealt with concurrent-call mechanism for threaded functions. Its implementation for the sequential-call mechanism is slightly different.

The function that calls another function is the *caller*, while the function that is called is the *callee*. In a sequential function call mechanism (invoking child threaded functions with the `CALL` statement), the caller is suspended until the callee returns. In this case, the C function corresponding to the first thread (thread `0`) of the callee has two additional parameters : the caller frame pointer; and a pointer to the instructin after the `CALL` statement in the caller. Similarly, the last statement in the C function, that represents the last executed thread of the callee threaded function, is a call to the RTS function `etc_return`. This statement is generated by the preprocessor from the Threaded-C primitive `RETURN`.

The preprocessed code for threaded functions with the `CALL` construct, has at least as many C functions as the number of CALL instructions, such that each CALL instruction is serviced in one C function. Each of these C functions terminates after calling the first thread of the callee with arguments. These C functions are formed in two phases. In the first phase, the arguments for the callee function are stored in a parameter frame structure. A pointer to the parameter frame is declared and is allocated heap memory. Some arguments, like the *result*, may have to be converted into global addresses. This parameter frame is indirectly referenced by the parameter pointer. In the second phase, the callee threaded function is invoked by calling the C function corresponding to its first thread with three arguments: the parameter pointer (to access the arguments) the instruction pointer of the next thread (the name of the C function which corresponds to the code in the caller threaded function) and the frame pointer for the caller threaded function. The last two arguments are combinedly called the *return thread pointer*. The C function in the caller threaded function terminates immediately after initiating the callee threaded function in this manner.

The parameters for the callee threaded function and the return thread pointer (instruction pointer and frame pointer) are stored in the activation frame of the callee threaded function. There is no difference between the preprocessed codes for threaded functions instantiated with `CALL`, with `INVOKE`, or with `TOKEN`, except for the parameters to the callee threaded function, and its exit semantics.

Henceforth, we shall use the term *callee* to refer to the phrase - "child threaded function invoked with the `CALL` construct". The preprocessed code for a callee is shown in Fig. 15. Preprocessed code for a threaded MAIN function using two CALL instructions is shown in part (a), while the preprocessed code for the CALLed function is shown in part (b). The code shown here is not complete, only the parts relevant to the discussion are included.

line no : Comments for part (a) of Fig. 15

line 3: The `CALL` construct is used to invoke the child threaded function `call_token_fib` with two arguments.

```
THREADED MAIN (void)
{
        ---
CALL (call_token_fib, val, TO_GLOBAL (&res));
CALL (call_mixed_fib, val, TO_GLOBAL (&res));
    RETURN ();
}

void _MAIN_0 (_MAIN_P *_pp)
{
    etc_gptr _gptr1;
    _MAIN_F *_fp;
    _call_token_fib_P *_call_token_fib_pp;
    void _MAIN_C1 ();

    _fp = (_MAIN_F *) etc_rts.free_64;
    if (_fp)
 etc_rts.free_64 = _fp->_slots[0].cnt;
    else
 _fp = (_MAIN_F *) malloc (64);
           -------
   _gptr1.node = etc_rts.node_id;
   _gptr1.ptr = &_fp->res;
   _call_token_fib_pp = (_call_token_fib_P *)
                           etc_rts.next_free;
   _call_token_fib_pp->n = _fp->val;
   _call_token_fib_pp->result = _gptr1;
   _call_token_fib_0 (_call_token_fib_pp,
                        _MAIN_C1, _fp);
}

void _MAIN_C1 (_MAIN_F *_fp)
{
    etc_gptr _gptr1;
    _call_mixed_fib_P *_call_mixed_fib_pp;
    void _MAIN_C2 ();

   _gptr1.node = etc_rts.node_id;
   _gptr1.ptr = &_fp->res;
   _call_mixed_fib_pp = (_call_mixed_fib_P *)
                           etc_rts.next_free;
   _call_mixed_fib_pp->n = _fp->val;
   _call_mixed_fib_pp->result = _gptr1;
   _call_mixed_fib_0 (_call_mixed_fib_pp,
                         _MAIN_C2, _fp);
}
void _MAIN_C2 (_MAIN_F *_fp)
{
    etc_gptr _gptr1;

        ----
        ----
}
```

(a) MAIN threaded function

```
typedef struct {
    long _next, _prev, _ip, _fp;

    int n;
    etc_gptr result;
} _call_token_fib_P;
typedef struct {
    etc_slot _slots[1];
    void (*_ret_ip) ();
    void *_ret_fp;
    int n;
    etc_gptr result;
    int r;
} _call_token_fib_F;

void _call_token_fib_0 (_call_token_fib_P *_pp,
                 void _ret_ip (), void *_ret_fp)
{
    etc_gptr _gptr1, _gptr2;
    _call_token_fib_F *_fp;
    void _call_token_fib_1 ();
    _token_fib_P *_token_fib_pp;

    _fp = (_call_token_fib_F *) etc_rts.free_64;
    if (_fp)
 etc_rts.free_64 = _fp->_slots[0].cnt;
    else
 _fp = (_call_token_fib_F *) malloc (64);
    _fp->_ret_ip = _ret_ip;
    _fp->_ret_fp = _ret_fp;
    _fp->n = _pp->n;
    _fp->result = _pp->result;
    _fp->_slots[0].cnt = 1;
    _fp->_slots[0].rst = 1;
    _fp->_slots[0].ip = _call_token_fib_1;
    _fp->_slots[0].fp = (long) _fp;
    _gptr1.node = etc_rts.node_id;
    _gptr1.ptr = _fp->_slots + 0;
    _gptr2.node = etc_rts.node_id;
    _gptr2.ptr = &_fp->r;
    _token_fib_pp = (_token_fib_P *)
                      etc_rts.next_free;
    _token_fib_pp->done = _gptr1;
    _token_fib_pp->n = _fp->n;
    _token_fib_pp->result = _gptr2;
    etc_invoke (0, _token_fib_0, 24);
}
void _call_token_fib_1 (_call_token_fib_F *_fp)
{
    etc_gptr _gptr1, _gptr2;

    *(int *) _fp->result.ptr = _fp->r;
_end_fun:
    _fp->_slots[0].cnt = etc_rts.free_64;
    etc_rts.free_64 = (long) _fp;
    etc_return (_fp->_ret_ip, _fp->_ret_fp);
}
```

(b) Callee threaded function `call_token_fib`

Figure 15: Pre-processing of CALL instruction

line 4: Another threaded function `call_mixed_fib` is the callee with two arguments.

line 7: Preprocessed code for the first thread of the `MAIN` threaded function is shown here. Though there is only one thread in the MAIN threaded function, three C functions are present in the preprocessed code, because of its two CALL statements.

line 11: A pointer is declared to parameter frame of function `call_token_fib`.

line 12: Function declaration of next C function, to which control has to be returned by the callee threaded function.

line 18-19: The `result` location is converted into a global address.

line 20: Obtain heap memory for parameter pointer.

line 22-23: Upload arguments into parameter frame of the callee threaded function.

line 24: A C function call to the first thread of the callee threaded function, with parameter pointer (for arguments), return instruction pointer (function pointer for next C function of MAIN), and return frame pointer (frame pointer of the caller threaded function) is made.

line 26: The first C function for the MAIN threaded function ends. The remaining part of the MAIN code is executed in the next two C functions. This is done to enable specification of the program counter, now pointing to the statement following the `CALL` instruction in MAIN threaded function, as the return instruction pointer.

line 27: Control reaches here, when the callee threaded function returns, as may be seen in `line 52` of part (b) of Fig. 15.

line 28-40: The same steps as above, repeated for the second CALLed function `call_mixed_fib`.

line 41-44: This is the last C function representing the MAIN threaded function. Control reaches here after the callee `call_mixed_fib` has returned.

line no : Comments for part (b) of Fig. 15

We shall discuss the preprocessed code for one of the callees, the `call_token_fib`.

line 8: Declaration for return function pointer.

line 9: Declaration for return frame pointer.

line 14: This is the first thread of the callee threaded function. The C function receives as parameters, the return *thread pointer* (instruction pointer, and frame pointer) and a pointer to the parameter frame holding arguments for this child threaded function. The thread pointer points to the C function of the caller threaded function, to which control has to be returned when this threaded function exits.

line 26-27: The thread pointer (as obtained from the parameter pointer) is stored in the activation frame of the callee.

line 45: Start of second thread of the callee. The return thread pointer is preserved in the activation frame pointed to, by frame pointer.

line 51: Returning memory held by activation frame to heap.

line 52: Returning control to C function of the caller threaded function.

The `END_FUNCTION` and `RETURN` constructs are two different ways of returning from threaded functions. END_FUNCTION is used when a threaded function is invoked with the concurrent-call mechanism (using TOKEN or INVOKE). The invoking function and the invoked function can execute simultaneously. The RETURN is used in the case of sequential-call mechanism (using CALL), when the caller function is blocked until the called function returns. For this purpose, the return thread pointer in the caller threaded function is passed as one of the arguments to the callee.

3.12 Loops spread over Threads

Threaded-C allows for loops to be spread over more than one thread. In the preprocessed code, the loop will be represented in all the C representations of the threads. However, control does not toggle between these C functions, as it does among the threads. A threaded function, with a long *while* loop spread among 2 threads, is shown in part (a) of Fig. 16, and its preprocessed code is shown in part (b) of Fig. 16.

Before proceeding further, we define the term *ready queue*. Threads guaranteed to be scheduled on a node are placed in the ready queue (RQ) of that node. Threads from the ready queue are executed in the FIFO fashion. The RQ is discussed in detail in section 4.1.

For instance, the `while` loop is spread over threads `0` and `1`, in part (a) of Fig. 16. The loop ends in the second thread (*line 13*). At the threaded function level, the understanding of execution of this loop is that, when thread `0` finishes, control enters thread `1` (as this thread is spawned from thread `0`), and after executing the only statement before the ending brace for while loop, control jumps back to the beginning of the loop body in thread `0`. This continues until the termination of the loop. However in the preprocessed code for thread `0` (`_get_loops_p1_0`) seen in part (b) of Fig. 16, the `while` statement is replaced by an `if` statement (*line 14*). Part of the loop body, that is in thread `0`, is mapped into the C function representing that thread, i.e. part of the first iteration of the loop is retained in this C function.

In the C function representing the second thread, the complete `while` loop is retained, including a C `return` statement (preprocessed code for `END_THREAD`), at the thread boundary (*line 42*). However, the thread `0` part of the first iteration in this `while` loop is skipped, by using a C `goto` statement (*line 34*), as that part must already have been executed in code for thread `0`.

When the thread `0` completes execution, it has spawned thread `1` (*line 17*). Control jumps to end of loop in C function corresponding to thread `1`, increments the `loops` variable (*line 44*), and jumps back to start of the `while` loop (*line 35*). The loop body is executed for next iteration, but the function terminates and exits at the `return` statement (*line 42*). Nevertheless, there is another thread `1` spawned in the loop body that shares the same activation frame. Execution of the new C function for thread `1` starts, and ends in same fashion as earlier. This continues until

```
THREADED get_loops_p1 (SPTR done,
               int *GLOBAL result)
{
   SLOT SYNC_SLOTS [1];
   int del, loops;

   loops = 0;
   while (!stop) {
   SPAWN (1);
  wait (10);
   END_THREAD ();

THREAD_1:

   loops += 1;
    }
    DATA_RSYNC_L (loops, result, done);
    stop = 0;
    END_FUNCTION ();
}
```

(a) Threaded function with `While` loop spread over threads

```
void _get_loops_p1_0 (_get_loops_p1_P *_pp)
{
   etc_gptr _gptr1;
   _get_loops_p1_F *_fp;
   void _get_loops_p1_1 ();

   _fp = (_get_loops_p1_F *) etc_rts.free_64;
   if (_fp)
etc_rts.free_64 = _fp->_slots[0].cnt;
   else
_fp = (_get_loops_p1_F *) malloc (64);
    _fp->done = _pp->done;
    _fp->result = _pp->result;
    _fp->loops = 0;
    if (!stop) {
_gptr1.node = etc_rts.node_id;
_gptr1.ptr = _fp;
etc_spawn (_gptr1, _get_loops_p1_1);
for (_fp->del = 0; _fp->del < 10 * 4;
                                   _fp->del++)
     nop ();

return;
    }
    etc_data_sync_l (_fp->loops, _fp->result,
                                    _fp->done);
    stop = 0;
_end_fun:
    _fp->_slots[0].cnt = etc_rts.free_64;
    etc_rts.free_64 = (long) _fp;
}

void _get_loops_p1_1 (_get_loops_p1_F *_fp)
{
    etc_gptr _gptr1;
    void _get_loops_p1_1 ();

    goto THREAD_1;
    while (!stop) {
_gptr1.node = etc_rts.node_id;
_gptr1.ptr = _fp;
etc_spawn (_gptr1, _get_loops_p1_1);
for (_fp->del = 0; _fp->del < 10 * 4;
                                  _fp->del++)
     nop ();
return;

THREAD_1:
_fp->loops += 1;
    }
    etc_data_sync_l (_fp->loops, _fp->result,
                                    _fp->done);
    stop = 0;
_end_fun:
    _fp->_slots[0].cnt = etc_rts.free_64;
    etc_rts.free_64 = (long) _fp;
}
```

(b) Pre-processed code for `While` loop

Figure 16: Pre-processing of *While* Loop spread over Threads

the loop counter reaches its limit. In the last iteration, when the loop condition fails, control jumps to the statements after the loop body (*line 46*), thus skipping the `return` statement. After the execution of the statements at end of thread `1` after the loop body, the C function terminates properly, and exits.

One fact to be noted is that the threads spawned in each iteration will be placed at end of the ready queue, and therefore will be executed after the threads already in the queue. The early threads may correspond to another threaded function. Thus, splitting loops over threads provides a chance for threads of other threaded functions to be executed alongside/among the threads of this threaded function.

In summary, threaded code might be preprocessed into individual C functions for the following reasons :

- To separate long latency operations into separate threads, i.e., place the request for an operation in one thread, and code that uses the results of the operation in another thread. The decision to split threaded functions into individual threads is made by the Threaded-C programmer. Here each thread is represented by a C function.

- To facilitate restart of execution of a threaded function, from the instruction next to the one in which a callee is instantiated with the `CALL` construct. By translating the remaining part of the threaded function into another C function, the C function pointer can be easily specified as instruction pointer in the *return thread pointer*. Here the preprocessor makes the decision as to the number and structure of the C functions to represent the threaded code.

- To enable a scheduling fashion in which, a thread can be made to execute among other threads (belonging to another threaded function), that are ahead in the *ready queue*. Here, the loop is designed by the Threaded-C programmer, while the preprocessor generates a C function for each thread. The structure of these individual C functions allows for the representation of a loop spread over multiple threads.

To elaborate the third point further, some applications require that a thread be scheduled on a remote node, while there is another thread (say *Th. A*) that is still executing there, i.e. scheduling a thread (say *Th. B*) on a remote node, while another thread is, for example, executing a long loop there. However, threads in EARTH are fine-grain, and non-preemptive. In such a case, the remote thread (*Th. A*) will be executing forever, and the later threads will never get CPU time. If instead, the long loop in the remote thread (*Th. A*) is spread over two threads, and the first thread spawns the second thread, as shown in Fig. 16, then the second thread will be placed behind *Th.B* in the RQ, thus enabling *Th. B* to execute among the threads of another threaded function scheduled earlier on the RQ.

4 The Runtime System

The **Runtime System (RTS)** provides a multithreaded environment for running application threads efficiently. It's core responsibilities are as follows.

- Thread-Scheduling
- Context Switching
- Data Communication
- Synchronization
- Global Memory Management
- Dynamic Load Balancing

Thread scheduling is explained in section 4.1. Regarding context-switching, threads in Threaded-C are non-preemptive, therefore the term *context* here does not mean the usual combination of the register file, program counter, stack pointer, and the status register. Instead, the context of a threaded function includes the arguments, sync slots and local variables of a threaded function. Therefore, context-switching should be seen as *context-sharing* in Threaded-C. Context-sharing between the threads of a threaded function is explained with the frame-passing mechanism in section 3.8. Data communication is presented in section 4.3. Synchronization and global memory management are explained in section 4.4. Section 4.6 details dynamic load balancing.

Though this document details the SP-2/tb-3 version of RTS, the operating principles are the same among all versions, barring minor implementation details that are localized to a few routines.

4.1 Scheduling of Threads

Thread scheduling is different from load balancing in the EARTH model. Scheduling is the last activity by the RTS on threads before their execution, similar to *dispatching* in Operating Systems. The node for execution of a thread has already been identified prior to scheduling. Load balancing is at a higher level in the RTS functional hierarchy than scheduling, as may be noticed in Fig. 19. Load balancing makes threads available for scheduling. While scheduling is confined to one node, load balancing happens between all nodes.

From the definition of `token` in section 3.8, it may be recalled that tokens are still unexpanded function frames, and therefore can migrate between nodes. Tokens are the units of load balancing. Threaded function frames are expanded when their execution is guaranteed on a node. To aid in segregation of resource pools for scheduling and load balancing, two queues -

the `ready queue (RQ)` and `token queue (TQ)` are maintained by the RTS on each node. Load balancing operates on the TQ, whereas scheduling is based on the RQ.

The Threaded-C construct `INVOKE` enables the programmer to directly place threads comprising a threaded function in the RQ. On the other hand, threaded functions launched with the `TOKEN` construct, whose destination node is decided by the dynamic load balancer in the RTS at runtime, are placed in the TQ of the node that executed the `TOKEN` primitive.

Threads, rather than tokens, are the units of work in the RQ. These threads are the components of the threaded functions that are guaranteed to execute on a node. The threads in the RQ are in the *ready* state since all of their synchronization conditions have been met. While threads of a threaded function may execute in consecutive order, this order is not guaranteed as the firing of a thread depends on its synchronization conditions. Note that all the enabled threads of a threaded function have to execute on the same node. The FIFO scheduling policy is used to execute threads in the RQ. Threads are added at the tail, and removed from the head of the queue. Whenever the RQ is empty, a token is fetched from the TQ and the instruction pointer from the token is used to launch the first thread of the threaded function.

The TQ in contrast, is a `DEQUE` - a data structure similar to a queue, but operable on both ends. Tokens are the units of work in the TQ. The TQ behaves locally like a stack. When a node generates a token, the token is appended to the tail of the TQ (`PUSH` operation). For local consumption, a token is extracted from the tail (`POP` operation). However, the TQ acts like a FIFO queue when a token is to be sent to remote nodes as part of a load balancing operation. A token is removed from the head of the TQ, and sent to a remote node. When remotely generated tokens are received at a node, they are added at the head of the TQ. The basic principle is that, tokens are removed from the tail of the TQ for local consumption, and from the head of the TQ for remote consumption. The flow of tokens amongst the application, RQ, TQ and remote nodes is shown in Fig. 17.

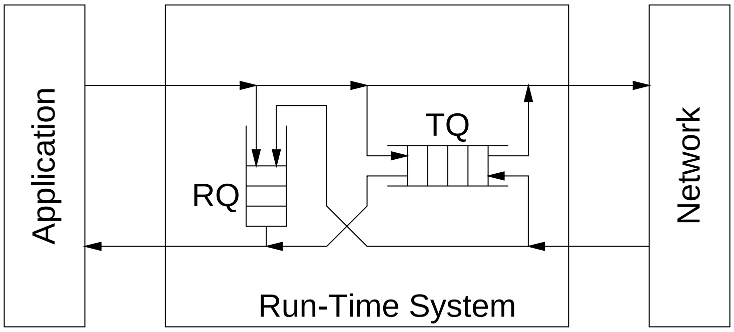

Figure 17: Internal Queues in the EARTH RTS

The TQ design is intended to reduce token migrations and thereby reduce space explosion (memory space for holding the activation frames). When expanding tokens or exchanging them with other nodes, the choice of a token is important as it determines the order in which the program is executed. The execution order in turn determines the amount of parallelism which can be exploited and the amount of memory needed to execute the program [8].

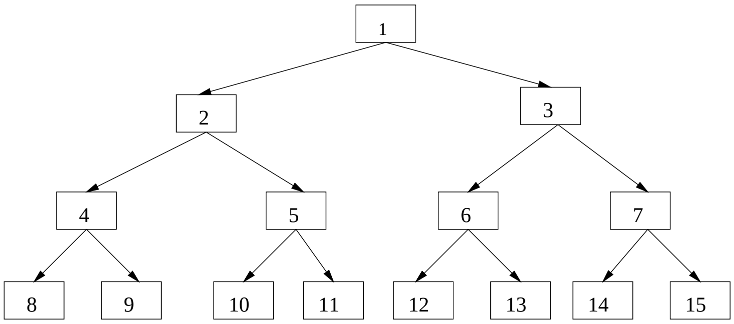

Figure 18: A Sample Activation Tree

Fig. 18 shows the activation tree of a simple, doubly recursive function. The rectangular blocks represent threaded functions, while the edges represent the `TIC` edges. If both calls can be executed in parallel, several execution sequences are possible. For instance, the sequence 1, (2,3), (4, 5, 6, 7), (8, 9, 10, 11, 12, 13, 14, 15) corresponds to a *breadth-first* execution. This execution order makes it possible to execute threaded functions 8 to 15 in parallel, i.e. maximum parallelism of 8. On the other hand, the sequence 1, 2, 4, 8, 9, 10, 11, 3, 6, 12, 13, 7, 14, 15 corresponds to a *depth-first* execution. In that case all threaded functions are executed sequentially.

The main differences between these two execution orders are the amount of parallelism that can be exploited and the amount of memory needed to execute the program. The depth-first strategy, which is the normal execution strategy for sequential processors, does not exploit parallelism, but at most 4 instances of the threaded function are active at the same time. On the other hand, the breadth-first approach takes advantage of all available parallelism but there must be enough memory to keep all 15 activation frames in memory at the same time. The choice of the execution order depends on the number of nodes in the multiprocessor system, , the time needed to start a threaded function on another node, the amount of work available to the other nodes, etc. An optimum strategy would be to exploit just enough parallelism to keep all processors busy while at the same time minimizing memory usage.

The TQ design approximates the ideal behavior, as explained above, quite well. For instance, when all processors are busy, no tokens are exchanged over the network. The TQ acts like a stack, and the token that was generated last is the first to be expanded. This corresponds to the normal depth-first execution strategy of sequential processors. On the other hand, when a node sends the tokens it has produced to one of its neighbors, it is the oldest token which is sent first. The resulting execution order is breadth-first, as the threaded functions nearest to the root of the activation tree are executed first.

The execution order therefore becomes dependent on the load of the machine. As long as the machine is busy, the code will be executed mostly in a depth-first order, using minimum of resources. As soon as one of the processors needs more work, however, tokens are sent to it in breadth-first order and some more parallelism is exploited.

Another advantage of this strategy is that it is able to take some advantage of locality. Executing code depth-first means that threaded functions are executed on the same node as their parent. Because parameters and results can be created and accessed locally, network traffic is reduced. This works especially well in the case of recursive functions.

To summarize, depth-first expansion of the activation tree is desired locally, whereas breadth-first expansion is preferred over a set of nodes. Function frames at higher levels in the activation tree represent more work than those in lower levels. Therefore, frames with more work in the activation tree should migrate to remote nodes, to offset the work done on remote nodes with the migration costs. Depth-first expansion on a node not only reduces token migrations, but also adds to the locality of the tokens migrated. When this idea is mapped on to queue structure, the tokens near the head of the TQ correspond to the functions on the top level of the activation tree. Hence the TQ is accessed in FIFO fashion for remote consumption, and in stack fashion for local consumption. The effect of a DEQUE structure for token queue is described in [8] and [10].

4.2 Thread Execution by the Runtime System

The multithreading support provided by the RTS includes tracking functions, scheduling ready threads, polling the network at thread boundaries for messages, dynamic load balancing and data communication. The RTS keeps track of tokens produced in the token queue, ready threads in ready queue, and initiate the threads by accessing their C function pointers.

During initialization, the executable of a Threaded-C program is loaded into memory of all nodes participating in the execution. Remember that the executable includes object code for both the RTS and the application. Thread launching starts on node `0` from the '`_MAIN_0`' function, which corresponds to the first thread of the `MAIN` function of the Threaded-C program. Any child threaded functions are initiated by adding a token to the TQ on the relevant node. Enabled threads are placed in the RQ for execution. In the case of idle nodes, the dynamic load balancer on that node is invoked and it fetches tokens from other nodes according to the load balancing policy. Eventually threads are running on all nodes. The execution ends when no more threads remain to be executed.

Remote operations involving spawning of threads, sync operations, handling global memory, and inter-node data communication are based on the technique of active messages. An *active message* contains data content and the name of a function handler that needs to be invoked at remote destination, on the polling of a message [12].

The RTS activity [10] at a thread boundary is depicted in Fig. 19. When the execution of a thread completes, before executing the next thread, the RTS polls the network for any incoming messages. Depending on the type of the incoming message, a handler routine is invoked with arguments. Both the name of the handler routine, and its arguments are obtained from the message received. After servicing any incoming messages, the RTS starts executing the thread at the head of the RQ. If there are no threads to execute in the ready queue, a token from the token queue is executed. If the token queue is also empty, then a load balancing request is composed and sent to a remote node. The choice of the remote node is based on the load balancer policy. All load balancer related activities like : token request; token response; token forwarding; and any updation of load state information, happen at the thread boundary.

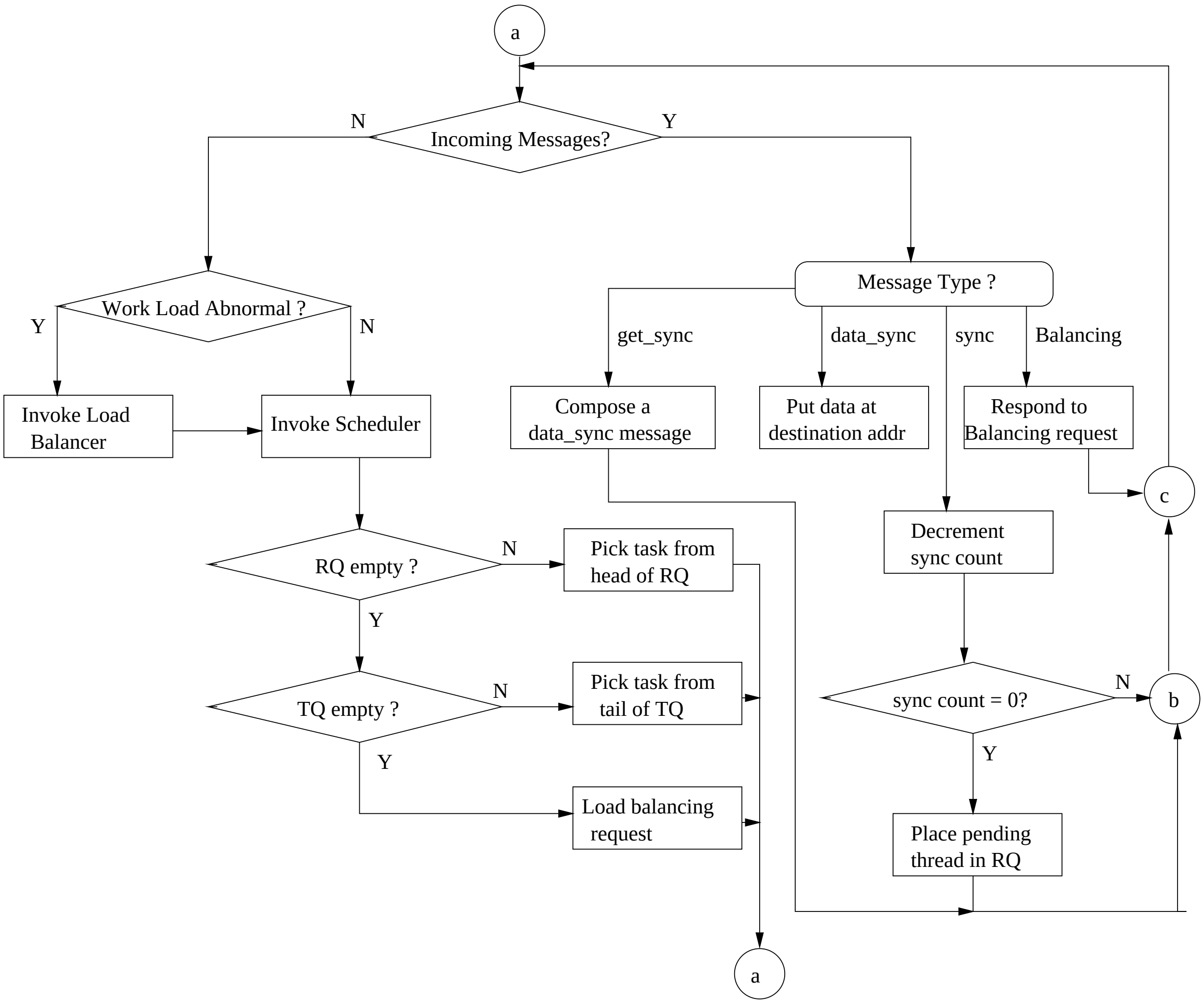

Figure 19: RTS activity at Polling

An incoming message is in one of the four groups :

Sync Request: For a sync operation, first the sync slot is obtained from the message. The sync count is decremented, and if it becomes zero, the sync count is reset and the relevant thread is placed in the ready queue for execution. The handler routine is `etc_sync`.

Remote Data Request: In the case of a request from a remote node for local data, the local data is composed into a message and sent to the requesting node. The handler routine is `etc_get_sync_X`.

Response to Remote Data Request: When remote data is received following a request sent earlier (`etc_get_sync_X`), the data is placed in the local destination address, and the sync count decremented. The sync count is accessed from the sync slot that is specified in the message. The handler routine is `etc_data_sync_X`.

Load Balancing Request: A load-balancing request is serviced depending on the load-balancing policy. If there are surplus tokens in the token queue, a token is sent to the remote node that requested work. Otherwise, the request is forwarded to the next logical neighbor. The handler routine is `hdl_token_req`.

Response to load balancing request: The incoming message contains a token from a remote node. This token is consumed if the node is still idle. Otherwise, depending on the load balancer policy, this extra token is either appended to the tail of the TQ, or forwarded to next neighbor. The handler routine that performs this operation is `hdl_token`.

4.3 Network Layer

The arrival of an active message causes a function handler to be invoked at the destination which acts on the data transmitted. The message transmitted contains two parts : the name of the handler routine to be invoked at the remote node, and the parameters for this handler routine. The C function pointer is used to represent the starting address of the handler routine on the remote node.

The data packets are identified depending on the number and size of the parameters transmitted as part of the active message. The RTS provides four primitives for sending active messages. All data packets have the handler routine name, but differ in the number of parameters they support. The handler routine at the remote node is invoked with these parameters.

- `etc_send2`: For data packets with two parameters.
- `etc_send4`: For data packets with four parameters.
- `etc_send6`: For data packets with six parameters.
- `etc_send2n`: For data packets with two parameters and a data block.

These four prototypes are sufficient to achieve inter-node communication for all types of data. They are part of the network interface of the RTS, and access the network card data structures (the tb-3 card data structures on the IBM SP-2 platform). To maintain speedy communications, the network card data structures are retained in the RTS communication interface, but this interaction is very limited to a few routines. By maintaining the above prototypes for the `etc_send` routines, the rest of the RTS code is independent of the network platform on the underlying parallel machine.

The number of tasks and the identity of the current node is obtained from the network environment. The network parameters like pointers for the send, receive, and overflow buffers are initialized in the network card data structures. Acknowledgment messages are sent so that remote nodes can send some more packets. When the send queue is full, outgoing messages are shifted to the overflow buffer, from where they are later sent.

Active messages are sent through the `etc_sendX` routines. The data packet is loaded into the tb-3 network data structure along with the number of parameters required for executing the handler routines at the remote end - either 2, 4, or 6. The composed data packet is placed in the network send queue. Fig. 20 shows the prototypes for the send routines.

```
void etc_send2(int dest,etc_handler hdl,int p1,int p2)

void etc_send4 (int dest etc_handler hdl,int p1,
                 int p2,int p3,int p4)

void etc_send6 (int dest,etc_handler hdl,int p1,
                 int p2,int p3,int p4,int p5,int p6)

void etc_send2n (int dest,etc_handler hdl,int p1,
                  int p2,void *p,long bytes)
```

Figure 20: Send routines for Active Messages

Polling the network is performed at the thread boundary, or at the explicit use of the 'POLL' statement [4]. The incoming message is picked from the receive queue, and depending on the message type (number of parameters), the handler routine is invoked with the parameters (both obtained from the message).

The RTS makes use of the `etc_send` primitives to implement remote data communication, remote synchronization, global memory management, and dynamic load balancing. The handler routines are present on all nodes, as the same executable is running on all the nodes. For instance to implement remote synchronization, the RTS on the local node composes an active message with the handler routine name (`etc_sync`) and the global address of the sync slot. The handler routine on invocation at the remote node, checks for node identity, and then performs the synchronization operation. While dealing with global addresses such as that of the sync slot in this example, the handler routine first checks the node number in the global address. If the node identity number matches the node number in the global address, then the requested operation is performed, otherwise the message is forwarded through an `etc_sendX` routine to the destination node. Thus global addresses are supported over the network layer.

4.4 Common RTS core

This section looks at the organization of the RTS source code while implementing some of the RTS core functionality - synchronization and scheduling, data communication and global memory management.

The starting point for the executable is the 'main' function in the file `rts.c`. The RTS variables, timer, network, load balancer, and profiler (if load balancer and profiler are chosen as options on `etcc` command-line) are initialized. The `etc_run()` function starts the execution of the Threaded-C code from it's `_MAIN_0` function and from here on, the RTS keeps track of application execution. The `etc_next_thread` function specified in the load balancer modules, determines the next thread to be executed - either from the RQ, TQ, or if both are empty, by requesting from the logical neighbors as per the load balancing policy. The threads are placed in the RQ by placing the function pointer in the 'ip' field, and the parameters in the 'fp' field.

The file `data.c` contains the macro names for the handlers. The inline expanded macro definitions are given in the file `data_inc.c`. The handlers are organized in two levels. The first level corresponds to the handler name obtained from the incoming message at polling. These invoked handlers have their name starting with the keyword `etc_` (for ex. `etc_sync`). The function of `etc_XXX` is to determine if the message is meant for the current node. If the message is for the local node, the corresponding function whose name starts with the keyword `hdl_` is invoked. The routine `hdl_XXX` proceeds to act on the parameters and perform relevant function. On the other hand, if `etc_XXX` determines that the incoming message is not meant for the local node, it forwards the handler name and parameters to the destination node. Fig. 21 gives an example of invoking handlers with the code for **etc_sync**. The code for `hdl_sync` is shown in Fig. 22.

```
/*
void inline_etc_sync (s_node, s_sp)
*/
#define inline_etc_sync(s_node, s_sp)  \
{\
 buf_elem *bp; \
 \
 if (s_node == etc_rts.node_id) { \
     inline_hdl_sync (((etc_slot *) s_sp)); \
 } else { \
 etc_send2 (s_node, hdl_sync, (int) s_sp, 0); \
 INC_REMOTE \
        } \
}
```

Figure 21: Invoking handler for Sync Operation

Global memory management makes use of the handlers as specified above. For a global address, the current node sends a request to the node on which the global address is present. The destination node, on identifying the node number on the global address as its own, immediately composes an active message with the data and the name of handler routine and sends it to the source node. If the node number in the global address does not match the node identity, it forwards the message to the node, whose number is derived from the global address. This strategy of checking for the node number before performing an operation, not only provides fault-tolerance from missing messages, but also enables the RTS code to run on any interconnection network in the parallel machine. This is the reason why it is required to convert local pointers into global addresses before issuing any remote operation.

To access a global address, the requesting node composes an active message with the global address, and the handler routine `etc_get_syncX`. This message is sent to the node which hosts the global address. When the host node receives a `get_sync`, it constructs a `etc_data_syncX` message which after returning the data content, also decrements the sync counter for the relevant thread. If the sync count is zero, the thread is placed in the RQ.

Regarding thread synchronization, an explicit sync message causes the handler routine `etc-sync` to be invoked on the destination node. The handler routine decrements the sync counter, and if the sync count reaches zero, places the corresponding thread in the RQ. Fig. 22 shows a typical handler for sync operation.

```
#define inline_hdl_sync(sp) \
{\
 buf_elem *bp; \
 \
 if (sp->cnt == 1) {\
     sp->cnt = sp->rst;\
     bp = etc_rts.next_free;\
     etc_rts.rdy_t->next = bp;\
     etc_rts.rdy_t = bp;\
     etc_rts.next_free = bp->next;\
     bp->ip = sp->ip;\
     bp->fp = sp->fp;\
     if (!etc_rts.next_free)
          etc_alloc_buf_elem ();\
 } else\
 sp->cnt -= 1;\
}
```

Figure 22: Handler for Sync operation

4.5 Architecture Specific Code

Instruction timing and cache parameters directly influence the execution time. Timing statistics for the instructions are obtained by accessing the on-board programmable timer. Assembly language routines for the IBM SP platforms return the accurate real-time clock, after accessing the on-board 64 bit *time base* register. These routines are called from the C function, `etc_time()` whenever system time is required. The `etc_time` routine returns time in *seconds*, as a double-precision value.

In the early versions of the portable RTS, the `ct_read()` routine is used to obtain timing information. This routine was used to maintain compatibility with the Threaded-C code developed for the EARTH-MANNA. The `ct_read()` routine in turn called `etc_time_raw()`, which is in assembly, and returned time in nanoseconds as a double precision value. But since the current version of the portable RTS, `etc_time()` is the routine used to access on-board real time clock, and should be used in future Threaded-C applications and the RTS profiling code.

To obtain exact execution times, cache misses are reduced by touching both the code and data segments initially. Corresponding assembly language routines flush cache lines specified as a single address or in an address range. In addition, memory-to-memory copy operation optimized for small blocks using both integer and floating-point registers is provided. Time delay is provided by an assembly language routine `etc_delay()`.

The above stated functions contribute to the minimal machine specific code of the portable RTS. These routines are available for RS6000 CPU, and SUN workstations respectively.

4.6 Dynamic Load Balancing

Multithreaded systems supporting fine-grain threads can deal effectively with the two common bottlenecks for performance - small grain size, and barrier synchronization, by including good load balancing in their runtime systems. The purpose of the dynamic load balancer in the portable EARTH system is to keep all the nodes busy, i.e. to avoid idle state for any node. A node is said to be *hungry* or to be in the *idle* state, when there are no threads to execute. A node with surplus workload is called a *rich* node. Depending on the load balancers policy used, the tokens migrate between the nodes that are running the application. Threshold values for workload are defined, and are used in classifying nodes either as rich or idle nodes. For instance, a node is hungry or idle, when the number of tokens in the TQ is less than two. Similarly a node is rich if its TQ has greater than or equal to 2 tokens. During the execution at the thread-boundary, the balancer on a node is invoked whenever a threshold is reached. Presently, there are seven dynamic load balancing policies implemented - *dual, spn, shis, snd, his, range, catapult.*

The dual, spn, and shis balancers are *receiver-initiated* balancers. With these balancers, the load balancer operation is initiated by the idle node. The idle node sends requests to other nodes, and receives work in the form of tokens from the rich nodes. This strategy is also referred to as *work-stealing.*

The snd balancer is a *sender-initiated* balancer. Here, the load balancing operation is started whenever a node has surplus workload. The rich node disperses its surplus tokens among other nodes in the system.

The his, range and catapult belong to the *hybrid* class of balancers. These balancers can act as both receiver-initiated and sender-initiated balancers depending on the load situation on the node. They also make use of load history information while deciding the destination node for sending a request or a token.

The seven load balancer policies [10, 9] provide a platform to study the various parameters that affect application performance, like the application model, grain size, logical topology, polling interval, and scalability. Our experimental results so far, show that there is no perfect load balancer for all situations. However, balancers that are based on history information performed well in most situations.

4.7 Portability

Portability is achieved by allowing minimum interactions with the hardware and network platforms. The solution adopted here is that the RTS be written in a high-level language (in this case C) which forms another interface layer between the application and the platform specific system software. The possible undermining of the execution speed is minimized by employing highly efficient algorithms and source code level optimization like code-inlining.

The functionality of the multithreaded constructs in Threaded-C is provided by the presence of RTS function calls in the preprocessed code for Threaded-C programs. This follows ample sharing of data structures between the preprocessed code and the RTS code resulting in a tight cohesive code. Therefore, a joint study of the preprocessed code and the RTS is necessary.

Two features that affect the portability of any parallel system emulator are, the *hardware*, and *interconnection network* platforms. The `EARTH` RTS to be *portable*, was designed to minimize any specific references to the underlying hardware instructions and network protocols. Precluding the RTS at any level lower than high level language satisfies the first condition, while localizing RTS code that interacts with the network in a way satisfies the second condition. The portable version of RTS code has minimum fraction of it in assembly.

The context which normally refers to the combination of the program counter, status register, stack pointer, and the participating general purpose registers, poses another problem to portability in multithreaded systems. However, as the `EARTH` system is a fine-grained multithreaded architecture, there is a simple solution to this problem. Since the threads are non-preemptive, there is no need to save the machine context for every thread during context-switching. Instead, the threads of a threaded function have to share the context. The *context* here, means the combination of the parameters, local variables, and the sync slots for a threaded function. The portable RTS achieves *context-sharing* with the *frame-passing* mechanism, as explained in section 3.8. Briefly, a threaded function with '`n`' threads is preprocessed into '`n`' separate C functions, each of them being scheduled separately. The context of a threaded function is stored in an activation frame, which is pointed to by the frame pointer. The frame pointer is passed between the C functions (representing threads) to enable context sharing. In this technique, non-preemptive threads execute as C functions which exit after completion, thus not requiring to save any register contexts between them. The reduced hardware interactions, in turn improves portability.

The current version of RTS is ported onto the SUN workstations, `IBM SP-2, IBM SP-3`, and the `Beowulf`. Its network interfaces maybe `myrinet, tb-2, tb-3`, and `TCP/IP`. The RTS may be ported onto new platforms very easily and quickly.

5 Putting it all Together

This section examines the RTS behavior in two specific cases - invoking a local function, and execution of a remote `GET_SYNC_L` operation. The Run-time system directory structure (as is presently on the EARTH SP-2 system) is presented in section 5.3. Finally, using the portable Threaded-C system is mentioned in section 5.4.

5.1 Invoking a Local Function

Consider the Threaded-C code for invoking a local function in Fig. 1. The local function invocation statement, `INVOKE(NODE_ID, fib, SLOT_ADR(0), n-1, TO_GLOBAL(&r1))` is preprocessed into C code shown in Fig. 13. The sync slot and the result location address are converted into global addresses so as to be accessible by remote nodes (`SLOT_ADR` makes sync slot global, while `TO_GLOBAL` makes an address globally accessible). The parameter pointer is assigned the top of free element buffer on current node, and parameters assigned to the fields of this structure.

The last statement in the preprocessed code is `etc_invoke`, a function call to the RTS. The function `etc_invoke` is defined in the file `calls.c`. This function has the current node id (0 in this case), the instruction pointer for first thread in this threaded function, and the size of the parameter structure stored on top of free element list in bytes, as arguments.

```
static void hdl_invoke (etc_handler fun,
                        long bytes)
{
  buf_elem *bp, *bn;

  bp = etc_rts.next_free;
  etc_rts.rdy_t->next = bp;
  etc_rts.rdy_t = bp;
  etc_rts.next_free = (bn= bp->next);
  bp->ip = fun;
  bp->fp = (long) bp;
  if (!bn) etc_alloc_buf_elem ();
  etc_get2n (bp->parms, bytes);
}
```

(a) Handler routine

```
void etc_invoke (int node, etc_handler fun,
                 long bytes)
{
  buf_elem *bp;

  if (node == etc_rts.node_id) {
      bp = etc_rts.next_free;
      etc_rts.rdy_t->next = bp;
      etc_rts.rdy_t = bp;
      etc_rts.next_free = bp->next;
      bp->ip = fun;
      bp->fp = (long) bp;
      if (!etc_rts.next_free) etc_alloc_buf_elem ();
  } else
    etc_send2n (node, hdl_invoke, (int) fun, bytes,
                etc_rts.next_free->parms, bytes);
}
```

(b) RTS routine for INVOKE statement

Figure 23: RTS performing Local Function Invocation

The function definition for `etc_invoke` is shown in part (b) of Fig. 23. The node number (a parameter) is compared with the current node number. If they are the same, the top element from free element list is grabbed, it's instruction pointer filled and placed on the RQ. The free element list is a singly linked list of free nodes, each node with fields for instruction pointer, frame pointer, an array for parameters, and typical link fields (prev, next). Dynamic memory is usually accepted and returned from this list. Memory is allocated manually (by using the malloc statement), only when the free element list is empty. It may be noted that the top of the free element list already has parameters stored on it. On the contrary, if node number is not the same as current node number, then the arguments as well as the contents of top of free element list are composed into an active message with a handler routine to perform the invocation on remote node (though this case won't arise in case of local function invocation) and sent to neighboring node, the message destined to reach the node with id as the node number.

5.2 Execution of a Remote GET_SYNC_L

The implementation strategy of the EARTH communication primitives (`GET_SYNC_X`, and `DATA_SYNC_X`) had been mentioned in section 4.3. The behavior of a remote `GET_SYNC_L` is explained with an example in this section.

```
THREAD_1:
   time_1 = ct_read ();
   for (i = count; i > 0; i--) {
        GET_RSYNC_L (valp, resp, ssp0);
END_THREAD ();
    }
```

Figure 24: Typical use of the `GET_RSYNC_L` primitive

Consider the use of the primitive `GET_RSYNC_L` as shown in Fig. 24. The preprocessed code for this segment is included in Fig. 25.

```
_fp->time_1 = ct_read ();
if ((_fp->i = 100 * 1000) , (_fp->i > 0))
   {
    etc_get_sync_l (_fp->valp, _fp->resp,
                    _fp->ssp0);
    return;
   }
```

Figure 25: Preprocessed code for `GET_RSYNC_L`

The `GET_RSYNC_L` primitive is used to request for `long` data from a remote location. As may be noticed in the generated code, a function call to RTS (`etc_get_sync_l`) is made. This function in the file `data.c` has a macro call in it's body (`inline_etc_get_sync_l`) as shown in the Fig. 26. The arguments for the macro are formed from the components of global pointer structures for source and destination locations. The macro definition for `inline_etc_get_sync_l` is included in the file `data_inc.c`. This macro checks if the source node number is the same as current node number, and if so, calls another handler routine (`inline_hdl_get_sync_l`), as shown in in Fig. 28, that composes a **data_sync** message with the value taken from source location. The **data_sync** message places the value at the destination location and decrements the relevant sync count.

```
void etc_get_sync_l (etc_gptr src, etc_gptr dest,
                     etc_gptr s)
{
  inline_etc_get_sync_l (src.node, ((long *) src.ptr),
                         dest.node, ((long *) dest.ptr),
                  s.node, ((etc_slot *) s.ptr));
}
```

Figure 26: RTS function etc_get_sync_l

```
static void hdl_get_sync_l (long *src_ptr, int dest_node,
               long *dest_ptr,int s_node, etc_slot *s_sp)
{
  inline_hdl_get_sync_l (src_ptr, dest_node, dest_ptr,
                         s_node, s_sp);
}
```

Figure 27: Handler hdl_get_sync_l

On the other hand, if source node number is different from current node number, the **inline_etc_get_sync_l** routine sends an active message to the source node with a handler name **hdl_get_sync_l** in file **data.c**. This handler definition is shown in Fig. 27. The handler routine **hdl_get_sync_l** makes a macro call to **inline_hdl_get_sync_call**, whose definition is in file **data_inc.c**. This macro, as mentioned above, composes a **data_sync** message to the destination location.

```
/*
void inline_hdl_get_sync_l (long *src_ptr,
     int dest_node, long *dest_ptr,int s_node,
     etc_slot_ptr s_sp)
*/
#define inline_hdl_get_sync_l(src_ptr, dest_node,
                        dest_ptr, s_node, s_sp) \
{ \
   inline_etc_data_sync_l ((*(long *) src_ptr),
             est_node, dest_ptr, s_node, s_sp); \
}
```

(a) Macro inline_hdl_get_sync_l

```
#define inline_etc_get_sync_l(src_node,src_ptr,
            dest_node, dest_ptr,s_node,s_sp) \
{\
 if (src_node == etc_rts.node_id) { \
     inline_hdl_get_sync_l (src_ptr, dest_node,
                   dest_ptr, s_node, s_sp); \
 }else{ \
       etc_send6 (src_node, hdl_get_sync_l,
        (int) src_ptr, dest_node, (int) dest_ptr,
        s_node, (int) s_sp, 0); \
       INC_REMOTE \
      } \
}
```

(b) Macro inline_etc_get_sync_l

Figure 28: Macro Definitions in file **data_inc.c**

5.3 Run-Time System Directory

The portable EARTH programming environment consists of several tools, in particular the `etcc` compiler driver, the `etcpre` preprocessor and the run-time system libraries. The EARTH home directory, pointed to by the EARTH_HOME environment variable, contains all these tools, as well as the necessary include files and libraries. The latest stable version of the source code is also available on some machines. The resulting directory structure is summarized in Fig. 29, as far as the portable run-time system is concerned.

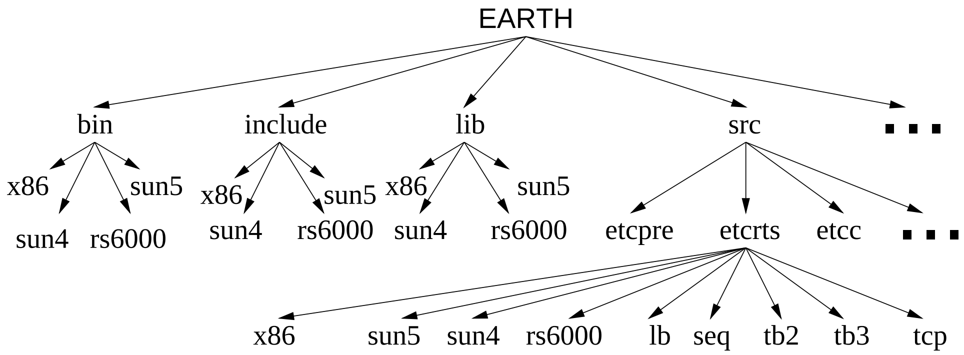

Figure 29: Partial EARTH Directory Structure

As can be seen from Fig. 29, three machine architectures are supported: sun4/sun5 (Sun workstations), rs6000 (RS6000 workstations and SP-2/SP-3 machines) and the Beowulf (LINUX PCs). In fact, MANNA would be a fourth architecture, but we do not include it in this diagram as the runtime system for the MANNA is written mostly in i860 assembly language, and therefore not portable across platforms. However, it may be noted that portable Threaded-C programs execute on the Manna architecture.

A closer look at the `etcrts` directory, which contains the source code for the run-time system libraries, shows separate subdirectories for the different machine architectures (`sun4`, `sun5`, `X86` and `rs6000`), the load balancers (`lb`), and each network implementation (`seq`, `tb2`, `tb3` and `tcp`). Different combinations of the CPU and network interfaces may be specified on the command-line to enable linking of the application object code with different versions of the run-time system object code. The machine and network independent parts of the run-time system are linked with one of the machine-dependent modules, a network module to generate the final executable. In addition, one of the load balancers, and the profiling option may be included on the command-line at compile-time. Off-the-shelf NOWs may operate in sequential mode, or may be connected with the `tcp/ip` interface. NOW products like the IBM SP-2, SP-3 are interconnected with the `tb-2` and `tb-3` network interface cards. The command-line arguments are fed to the `etcc` compiler driver which, on finding no errors, generates the final executable.

5.4 Running Threaded-C Programs

Presently, we support a 137 node IBM SP-2 at the Cornell Theory Center, an 80 node IBM SP-3 at the Argonne National Labs, and an 8 node Beowulf at the CAPSL Lab at the University of Delaware. This section details the compilation and execution sequences for Threaded-C programs on the IBM SP platforms.

In order to run Threaded-C programs, it is necessary to first compile them. In order for the Threaded-C compiler to work properly, the user's environment has to be set properly. The following paragraphs give a brief overview of what has to be done. More information can be found on the Web at `http://www-acaps.cs.mcgill.ca/info/EARTH`, though part of the information that is found there is specific to the EARTH-MANNA implementation.

In order to use the portable run-time system, the EARTH_HOME and the PATH environment variables have to be setup.

On the IBM SP-3 at the Argonne National Labs:

`setenv EARTH_HOME  kakulava/EARTH`

On the IBM SP-2 at the Cornell Theory Center:

`setenv EARTH_HOME  kprasad/EARTH`

The PATH variable should be set as follows:

`set path=($EARTH_HOME/bin/rs6000 $path)`

After the environment has been set up, programs can be compiled with the `etcc` command. This command is similar to the `cc` compiler driver and supports similar switches. For example, the following command can be used to compile the file `hello.c` and to generate a sequential version that runs on the Sun: `etcc -O hello.c -target sun4-seq`. The resulting Sun Sparc executable can be run as any sequential program. When accessing a parallel machine, on the other hand, some tool has to be used to get access to the machine and start the program on a specific number of nodes. The tools used for that purpose are : `tb3run` on the CACR SP-2, and `submit` on the Argonne SP-2 and Cornell SP-2. Note that this is not an exhaustive list, as more versions of the portable EARTH run-time system are being implemented.

A detailed explanation of the options for `etcc`, is present at `http://www-acaps.cs.mcgill.ca/info/EARTH/earth-manna/etcc.html`. The options supported by `etcc` are similar to those typically supported by other compilers. The relevant options for compiling Threaded-C programs on the IBM SP machines are as given below.

`etcc [-target target-arch][-prof][-lb dual|spn|nop|snd|range|his|shis] [-h][v][-keep][-O[level]][-o file][-c][-S][-E] [-Dname[=def]][-Uname]` file ...

`-target target-arch` Specifies the target architecture. Code may be generated for the following IBM SP machines.

`rs6000-seq` : Generate a sequential executable that runs on a single IBM RS6000 CPU.

`rs6000-tb2` : Generate code for the Argonne SP-2 cluster of workstations. This implementation directly accesses the tb-2 network card for better performance.

`rs6000-tb3` : Generate code for the Caltech CACR IBM SP-2 cluster of workstations. This version also directly accesses the tb-3 network card.

`-prof` Link with a different version of the portable run-time system that gathers profiling information and prints it after the program has terminated.

`-lb dual|spn|nop|snd|range|his|shis` Selects one of the possible dynamic load balancers. the spn balancer is used by default, except for the sequential targets which use the nop load balancer.

For example, to run the program `fib.c`, the following steps are to be followed after setting the environment variable `EARTH_HOME`. Consider the platform to be IBM SP-3, load balancer to be dual, and the profiling option included.

```
etcc -target rs6000-tb3 -prof -lb dual -O2 -o fib.dual.prof fib.c
```

Alternatively, the applications may be compiled using the `compile` script. Usage information is printed by typing `compile`.

`compile` [cflag] [bal] [prof] progname

- `cflag`: Any of the C optimization flags - O, O2,O3.

- `bal`: name of any of the load balancers, listed above.

- `prof`: Include profiling data in output.

- `progname`: Application name, without .c extension

To submit the executable for execution,

`submit` num_nodes max_time program [arguments...]

- `num_nodes`: Number of nodes requested.

- `max_time`: Maximum time that the application may take to execute.

- `program`: Name of the application exceutable

- `arguments`: Any application arguments may be included here

Usage information is printed by typing `submit`.

On the CACR SP-2, programs can be executed with the `tb3run` script.

```
tb3run -n 4 progname
```

- `n`: Number of nodes

The resulting output gives the concatenated output of all nodes in one file. The profiling code if included, helps provide statistical information as to the number of threads generated, number of remote communications, number of tokens migrated etc.

Fig. 30 shows a segment of the sample output for the Fibonacci Threaded-C program. The `fib` threaded function from Fig. 1 is compiled with the dual balancer and the profiling options. The executable is run with 32 nodes on the IBM SP-2 at the Cornell Theory Center. The sample output in Fig. 30 shows the program result, and some profile information on node `0`. The actual output includes profiling information for various idle periods when the node is idle, and this sequence is repeated for all 32 nodes that participated in the execution. An explanation for the profile statistics is provided in Appendix A.

```
0:### Start (32 nodes, dual)
0:fib(33) = 5702887
0:ELAPSED Time is 31.467594 s
0:### Elapsed time: 31.468582 s
0:---------- PROFILING DATA for 0/32 ---------
0:Tokens consumed:  4317400
0:Tokens generated:  4894583
0:Threads running:  20040574
0:# of balancing activities:  1154366
0:# of requests sent:  0
0:# of requests received:  577183
0:# of tokens sent:  577183
0:# of tokens received:  0
0:# of remote communications: 808304
0:# of idle periods:  12
0:Total idle time:  1435.0414 (1433.6824) us
```

Figure 30: Sample Output for the Fibonacci - fib(33) on 32 nodes

6 Performance Review

The performance of the portable EARTH system is reviewed in this section. The timing costs of various Threaded-C language and EARTH operations on EARTH SP-2[2] are studied here. The same study, as relevant to EARTH SP-2[3], is included in the appendix B (the architecture at this site is updated to IBM SP-3, recently).

The EARTH SP-2 (EARTH on RS6000 IBM SP-2 system) at CACR, has 9 nodes (Power2 CPU) running at different individual clock speeds of 67MHz, 77MHz, and 135 MHz respectively. The L1 cache is of 32KB, and memory is of either 128MB or 512MB, depending on the node's classification. Network interface is through the tb-3 card. The point-to-point peak bandwidth on the 135MHz wide nodes is rated at 90.41MB/s and one way node-to-node latency is approximately 19 μs.

The overhead for implementing various Threaded-C instructions on the CACR EARTH SP-2 is shown in Table 1. The timing costs in executing these instructions locally and remotely are computed by allowing the operations to execute on local and remote data structures and

[2] Located at the CACR, Caltech.
[3] Located at the Argonne National Laboratories, Oregon.

isolating the cost for the instruction. The instruction is executed a certain `count` number of times, taking care to wait between consecutive executions to allow for polling.

EARTH Operation	Local Operation	Remote Operation	
	EU Costs	Local Costs	Remote Costs
SYNC	140 ns	2349 ns	896 ns
SPAWN	138 ns	2061 ns	1009 ns
END_THREAD	456 ns	NA	NA
INCR_SYNC	159 ns	1921 ns	898 ns
DATA_SYNC	163 ns	1920 ns	1127 ns
GET_SYNC	165 ns	1720+1685 ns	3143 ns
INVOKE(1)	117 ns	1901 ns	1101 ns
END_FUNCTION(1)	567 ns	NA	NA
INVOKE(5)	111 ns	1684 ns	1213 ns
END_FUNCTION(5	646 ns	NA	NA
INVOKE(9)	120 ns	1952 ns	1194 ns
END_FUNCTION(9)	691 ns	NA	NA
INVOKE(18)	152 ns	1842 ns	1265 ns
END_FUNCTION(18)	702 ns	NA	NA

Table 1: Overhead for Threaded-C instructions on EARTH-SP2

The first entry in Table 1 is for the `SYNC` instruction. The local execution unit costs are attributed to the time spent to invoke the SYNC instruction, and the time taken for the execution of RTS functions (that implement the SYNC functionality). Profiling code is inserted before and after the SYNC instruction. When initiating Threaded-C instructions on remote nodes, the time spent on the current node in invoking the remote operation, is shown under the "Local Costs" of a remote operation. The time time taken by the Threaded-C instruction for its execution on the remote node is shown under "Remote Costs" of remote operation.

Obtaining local costs for local operations is quite simple when compared to the costs for remote operations. For instance, to find local costs for invoking SYNC instruction locally, the sync slot corresponding to a thread in this Threaded-C function is used as an argument to the SYNC instruction. On generating the appropriate number of sync signals, profiling code in the thread ready for execution (recently enabled) measures the time taken for executing the SYNC instruction locally, by dividing the elapsed time by the number of sync signals generated.

While computing overhead costs for remote operations, care is taken to ensure that the remote node is busy. The Threaded-C instruction is executed remotely under normal working conditions, by placing it in the ready queue (RQ) of the remote node. This is done by executing a simple `while` loop on a remote node that is terminated only after completing execution of the Threaded-C instruction. To allow for the execution of a thread containing the remote Threaded-C instruction that is being timed, the `while` loop is split into two threads, as described in section 3.12. The thread with the Threaded-C instruction under consideration, is placed in the ready queue along with two threads corresponding to the `while` loop. This creates an environment similar to those found in normal application programs. While computing the remote costs of remote operations, the time spent on executing Threaded-C instructions is dealt with separately from the time spent on executing the loops. This is easy, as the remote Threaded-C instruction and loops are the only computation load belonging to the application on the remote node. The time spent on loops is computed by determining the time spent on executing each loop and the total number of loops executed at the end of execution.

The cost for SYNC construct in Table 1 can be explained as the time taken to execute the corresponding RTS function, `etc_sync`. After determining the node on which the relevant sync slot is located, the associated thread is placed in the ready queue if the sync count is one (it would be zero on decrementing). Otherwise, the sync count is decremented.

The `SPAWN` construct is less expensive than the `SYNC` construct, as checking for sync count is avoided and the thread directly placed in the ready queue. The `END_THREAD` primitive indicates the end of the current thread, and therefore preprocessed into a '`return` statement at the end of the C function representing the thread. The END_THREAD primitive pertains to the current thread and acts more like a directive rather than as an instruction. Computing remote costs for it is unnecessary, as the costs would be same on local and remote nodes. The cost of END_THREAD is the same as that of terminating a C function.

The `INCR_SYNC` instruction costs more than the `SYNC` instruction because of an incrementing operation done prior to normal SYNC operation. A `DATA_SYNC` operation places the data at destination and sends a sync message to associated sync slot. Hence it costs more than a sync operation. A `GET_SYNC` determines the source node and a makes a call to the RTS function corresponding to DATA_SYNC, with the source and destination locations, and the sync slot as arguments. Understandably, it costs more than a DATA_SYNC operation.

The `INVOKE` instruction launches the Threaded-C function directly on the node specified. Considerable effort is spent on building up the environment before making the RTS call to `etc_invoke`. This includes, turning the arguments into global values, and setting up the variable parameter passing mechanism. The arguments have to be set up at the calling INVOKE instruction. At the receiving end in the invoked function, the parameters have to be downloaded from the parameter pointer. Thus it is interesting to see the overheads involved in having different number of arguments to the INVOKE instruction. The timing costs for `1`, `5`, `9`, and `18` arguments for the INVOKE instruction are studied. As expected, they show an increasing trend as the number of arguments increases.

The `END_FUNCTION` instruction indicates the end of a Threaded-C function (similar to the END_THREAD primitive which signifies the end of a thread). The dynamic memory allocated for the activation frame is returned, as may be seen in the preprocessed code. To create normal working conditions, Threaded-C functions with `1`, `5`, `9`, and `18` parameters having just a single instruction - END_FUNCTION are invoked, and the results observed. The timing costs increase with the increase in number of parameters, as these parameters have to be obtained from the parameter frame. The costs represent the time for obtaining the parameters from the parameter frame, returning memory to heap, and terminating the C function representing the last thread in the Threaded-C function.

The timing requirements for important EARTH operations like synchronization, thread spawning, data communication, and function invocation, are studied in the table shown in Table 2. The time taken for the entire EARTH operation is observed with respect to the manner, in which the operations are issued - *sequential* and *pipelined.* In sequential issue, a single EARTH operation is initiated, and its timing recorded. In pipelined issue, multiple instances of a single EARTH operation are started. The motivation, for this was to observe any possible improvements in timing with the pipelined issue, as certain book-keeping operations may need to be performed only once for all the multiple instances of the EARTH operation. This issue is implemented in the Threaded-C program, and is over and above any possible instruction-level parallelism in the underlying architecture.

The results have confirmed timing improvements with pipelined execution, as shown in Table 2. For each EARTH operation, the time required to execute it in sequential fashion, is shown for local and remote executions. Similarly, local and remote executions are done for the pipelined fashion.

One uniform trend visible among all operations is that, the sequential execution costs far exceed those of corresponding pipelined execution. The timing overheads include the time for setting up the EARTH operation (`for` loops, if any) and the completion of the operation. This is unlike the timing overheads seen in Table 1, where the time is measured strictly for executing the Threaded-C instruction. This might be the reason for the slightly higher costs for local pipelined execution of EARTH operations than the local costs for Threaded-C instructions.

The local sequential execution has higher costs than the local pipelined costs, because of the time savings available when certain part of the RTS code is executed only once for all instances of the EARTH operation in the pipelined execution. For example for the `SYNC` operation, the RTS code that resets the sync count and places the thread in the ready queue is executed only for the last instance in the pipelined execution. In contrast, this is done for every instance in the sequential execution.

As seen in Table 1, writing a word takes (`data_sync`) lesser time than reading a word (`get_sync`). This same behavior is visible even in Table 2. The costs for the entire operation of reading/writing word is costlier than individual read/write instructions in Threaded-C because the former includes the costs for the `for` loop, and enabling of threads over which the loop is spread over.

The function call operations show increasing time overheads as the number of parameters increases. This is understandable, as the time spent in executing exra RTS code for uploading/downloading each parameter to/from the parameter frame.

Depositing data at local/remote locations by child computations is a crucial step before synchronization in multithreaded architectures. Hence, it is interesting to note the time costs in moving data among local/remote destinations. The overhead costs and throughput achieved by

Operation	Local Seq.	Remote Seq.	Local Pipe.	Remote Pipe.
Sync Thread:	661 ns	20921 ns	178 ns	3147 ns
Spawn Thread:	639 ns	20888 ns	NA	NA
Read Word:	762 ns	41031 ns	241 ns	5563 ns
Write Word:	689 ns	41011 ns	201 ns	5642 ns
Fun. Call (1):	1445 ns	42165 ns	922 ns	6183 ns
Fun. Call (5):	1528 ns	43045 ns	986 ns	6879 ns
Fun. Call (9):	1631 ns	43141 ns	1060 ns	6917 ns
Fun. Call (18):	1686 ns	43225 ns	1154 ns	6527 ns

Table 2: Overhead costs for EARTH operations on EARTH-SP2

using the `GET_SYNC_X` and `DATA_SYNC_X` operations are listed in Table 3. The sending/receiving of data is performed to/from local and remote destinations. The entries in Table 3 show the overhead in nanosecs, and the throughput in MB/secs, as a result of local/remote data transfers. Data of all sizes are transferred - `byte, short, long, double`, of sizes 1, 2, 4, and 8 bytes respectively. The difference between the times of start and end of a `GET_SYNC_X/DATA_SYNC_X` is the time overhead, and the amount of data transferred, divided by the elapsed time is the throughput achieved.

Initially, the source array of data is initialized on node `0`, by a threaded function (say Th. A). The destination array of data is then passed as pointer to another threaded function (say Th. B), where its contents are reset to zero and its pointer updated with the address of a local array variable. For initiating local data transfer, the second threaded function (Th. B) is invoked on node `0`, whereas for remote operations, it is invoked on node `1`. All data transfer happens between the source array (declared in Th. A), and the destination array (declared in Th. B) pointing to dynamic memory allocated in second threaded function (Th. B).

Operation	Local		Remote	
	Overhead	Throughput	Overhead	Throughput
DATA_SYNC_B	249 ns/op	4.02 MB/s	5632 ns/op	0.18 MB/s
DATA_SYNC_S	240 ns/op	8.33 MB/s	5616 ns/op	0.36 MB/s
DATA_SYNC_L	228 ns/op	17.55 MB/s	5591 ns/op	0.72 MB/s
DATA_SYNC_D	212 ns/op	37.74 MB/s	5454 ns/op	1.47 MB/s
GET_SYNC_B	286 ns/op	3.50 MB/s	5615 ns/op	0.18 MB/s
GET_SYNC_S	287 ns/op	6.97 MB/s	5614 ns/op	0.36 MB/s
GET_SYNC_L	272 ns/op	14.72 MB/s	5620 ns/op	0.71 MB/s
GET_SYNC_D	296 ns/op	27.01 MB/s	5475 ns/op	1.46 MB/s

Table 3: Overhead costs for GET_SYNC operation on EARTH SP-2

Remote memory access is costlier than local memory access (approx. 20 times), as in any

NUMA. The overhead costs are almost same for data of all sizes (though they show a negligible decreasing trend from byte to double for DATA_SYNC_X operation), the throughput goes on increasing from byte to double, as the amount of data transferred is increasing while the time taken is almost same.

Performancewise, moving blocks of data is more advantageous than multiple datum transfers. The saving in synchronization costs is realized through servicing a single sync signal, as against multiple sync signals. The bandwidth resulting from block movement of data to local/remote destinations is shown in Table 4. Typically, block moves of data result in a throughput of around 230MB/s (local) and 90.41MB/s (remote) for 0 byte aligned data block movement.

Align	Local		Remote	
	Single	Dual	Single	Dual
0	230.06 MB/s	230.99 MB/s	90.41 MB/s	98.04 MB/s
16	230.63 MB/s	230.61 MB/s	90.21 MB/s	97.61 MB/s
8	227.89 MB/s	229.67 MB/s	89.71 MB/s	96.19 MB/s
4	229.84 MB/s	230.11 MB/s	91.29 MB/s	97.44 MB/s
1	216.74 MB/s	217.38 MB/s	91.78 MB/s	97.35 MB/s

Table 4: Bandwidth for Blockmove operations on EARTH-SP2

The results are obtained by varying destinations among local/remote, single/dual blocks to transfer, and the byte alignment of data. In the single block transfer, data is transferred between one pair of source-destination during the time observed. With double block transfer, data is transferred between two pairs of source-destinations. In addition, the data is `0`, `1`, `4`, `8`, `16` bytes aligned. For example, the first entry in Table 4 shows transfer of zero aligned data, i.e. the data from the very first byte of array at source location is transferred. In the local block transfer, both source and destination are on the local memory whereas in the case of remote block transfer, remote data is transferred to local destination. Elapsed times are measured while transferring single and dual blocks of data. If 20 sync signals are to be generated to enable a consumer thread to act on the blocks of data transferred, the 20 signals may be generated after transferring single block data 20 times, or they may be split up between two block transfers (each generating 10 sync signals). Thus the entries in Table 4 show bandwidth in MB/s achieved by moving appropriately aligned data, stored in single/dual blocks among local source/destinations and remote source to local destinations.

Local block transfer, as expected, achieves better bandwidth (approx. 2.6 times) than remote block transfer. This is understandable as the overhead costs associated with remote memory write and sync operations are quite high. In fact, the single block transfer does very well, considering the high disparity in overhead costs (approx. 28 times) between remote and local pipelined memory word write operations. This confirms that blocking of data is profitable, when compared to multiple word transfers.

For any alignment, the dual block transfer achieves higher bandwidth than the corresponding single block transfer. This suggests lesser time required for dual block transfer than single block transfer. The difference between dual block transfer and a corresponding single block transfer is quite pronounced in the CACR SP-2 machine, while it is almost nonexistent on the ANL SP-2 machine. This might possibly be due to the faster clock/network speeds on the CACR machine, which allows faster communication and synchronization.

7 Conclusion

The purpose of this document is to study the realization of parallel constructs in Threaded-C, and finally the implementation strategies for the run-time system (RTS). The programming model is addressed from the RTS perspective, and an effort has been made to appreciate the data structure requirements to support such a model. A study of the preprocessed code for the parallel constructs of the Threaded-C leads to an understanding of techniques for realizing a multithreaded language with C language. In this regard, all facets of multithreaded systems - parallel thread launching, scheduling of threads, inter- thread communication/synchronization, and global memory management are examined. The RTS is described with emphasis on mechanisms employed for supporting multithreaded computation, and substantiated by sufficient code extracts. Two specific cases of multithreaded computation are closely studied in the final section to understand the RTS behavior. Finally, the performance measurements are documented to show reasonably low latencies associated with EARTH operations. A quick glimpse through the current version of RTS concerning its directory structure and usage ends the document.

8 Acknowledgments

The authors would like to thank current and former members of the ACAPS Lab, McGill University, and the CAPSL Lab, University of Delaware for their insight, ideas and help. The quality of this document has improved immensely, thanks to the many comments and useful suggestions of Dr. Jose Nelson Amaral. Thanks are due to Sean Ryan for reviewing this document and suggesting corrections. The authors thank the CACR, Caltech and the Argonne High-Performance Computing Research Facility for making this project possible by providing us access to their IBM-SP2 systems. The initial EARTH work was partially supported by the Natural Sciences and Engi neering Research Council (NSERC) of Canada.

References

[1] Hisham Petry. *Earth Threaded-C Programming manual*, March, 1996.

[2] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Xinan Tang, Guang R. Gao, Phil Cupryk, Nasser Elmasri, 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 Lubomir Bic, Wim Böhm, Paraskevas Evripidou, and Jean-Luc Gaudiot, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 59–68, Limassol, Cyprus, June 27–29, 1995. ACM Press.

[3] Laurie J. Hendren, Guang R. Gao, Xinan Tang, Yingchun Zhu, Xun Xue, Haiying Cai, and Pierre Ouellet. *Compiling C for the EARTH Multithreaded Architecture*,ACAPS Techinical Memo 101, March 18,1996.

[4] Olivier Maquelin, Guang R. Gao, Herbert H.J. Hum, Kevin B. Theobald, and Xin-Min Tian. *Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling*, Proceedings of the 23rd Annual International Symposium on Computer Architecture, Philadelphia, Pennsylvania, May 22-24, 1996, pp. 179-188.

[5] Kevin B. Theobald, Jose Nelson Amaral, Gerd Herber, Oliver Maquelin, Xinan Tang, Guang R. Gao. *An Overview of the Portable Threaded-C Language*, CAPSL Technical Memo 19, University of Delaware, March, 1998.

[6] Xinan Tang, Guang R. Gao, and Olivier Maquelin. *Design and Implementation of the EARTH Threaded-C Preprocessor.* Technical Report, CAPSL, Univ. of Delaware, March, 1998.

[7] Xinan Tang, Olivier Maquelin, Guang R. Gao and Prasad Kakulavarapu. *An Overview of the Portable Threaded-C Language*, ACAPS Tech Note 2, October 1997.

[8] Olivier C. Maquelin *Load Balancing and Resource Management in the ADAM Machine*, in Advanced Topics in dataflow Computing and Multithreading (Guang R. Gao, Lubomir Bic, and Jean-Luc Gaudiot, eds), pp. 307-323, IEEE Computer Society Press, 1995.

[9] Haiying Cai, Olivier Maquelin, Prasad Kakulavarapu, and Guang R. Gao. Design and Evaluation of Dynamic Load Balancing Schemes under a Fine-grain Multithreaded Execution model. In *Proc. of the Multithreaded Execution Architecture and Compilation Workshop, Orlando, Florida*, January 1999.

[10] Haiying Cai. *Dynamic Load Balancing on the EARTH-SP System.* Masters Thesis, McGill University, May 1997.

[11] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently*, Ph. D Thesis, MIT, September 1995.

[12] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. *Active Messages: A Mechanism for Integrated Communication and Computation*, in the Proceedings of the 19th International Symposium on Computer Architecture, ACM Press, Gold Coast, Australia, 1992.

A Appendix - Profiling support in the EARTH RTS

To monitor the performance of Threaded-C program execution, profiling support is provided in the EARTH-SP2 RTS. This profiling support is based on certain profiling parameters which besides indicating the program execution time, also provide a glimpse into the RTS behavior. The profile data identifies the number of different individual RTS operations that are required to execute a single Threaded-C program. This is later useful in constructing a cost model for program execution on the EARTH RTS, and also identify possible areas (both within the RTS and the application) for further optimization.

The profile code is mainly present in the files `prof.h` and `prof.c`. As may be noted from the commandline options of the `etcc` (EARTH Threaded-C Compiler), the `prof` option allows this profiling code to be linked in making the final executable. The profiling data is declared in `prof.h` and initialized in `prof.c`. Throughout the RTS, whenever a RTS operation is performed, relevant profile data is updated. The code to update this data is conditionally compiled along with the rest of the RTS (based on whether the `prof` option is included in the `etcc` commandline).

A list of the items collected under the profile data is provided below. Currently, work is under progress to identify and implement extensions to existing profile data, to get more clearer picture of the RTS behavior.

- **Tokens Consumed**: The number of tokens consumed on this node.
- **Local Tokens Consumed**: The number of locally generated locally consumed tokens.
- **Tokens Generated**: The number of tokens generated on this node (by executing the function call `etc_token`).
- **Threads Running**: The number of threads executed on this node. It may be noted that the number of threads may not equal the number of tokens (the former is generally higher).
- **Number of Balancing Activities**: The total number of dynamic load balancing activities during the execution. This is the number of load balancing related communication activities (does not include the CPU time spent on load balancing).

  No. of Load Balancing Activities = No. of Requests Sent + No. of Requests Received + No. of Tokens Sent + No. of Tokens Received
- **Number of Requests Sent**: The number of requests for tokens sent by the load balancer on this node to remote nodes.
- **Number of Requests Received**: The Number of requests received by this node from remote nodes.

- **Number of Tokens Sent**: The number of tokens sent by the load balancer on this node to remote nodes (maybe as response to earlier received requests or because of the sender/hybrid policy of the load balancer).

- **Number of Tokens Received**: The number of tokens received on this node (maybe as response to earlier sent requests, or because of sender/hybrid load balancer policies like `Snd`, `his`, etc.. or maybe because this node is an intermediate node as part of the logical ring topology, while the token is on its way to destination).

- **Number of Remote Communications**: The total number of remote communiations involved in implementing global memory management in remote memory access, synchronization, and spawning of threads `etc_spawn`. These are the communication activities spent in ensuring that certain RTS operations are performed at designated nodes, i.e. the intermediate message transfers until the message reaches the destination node (for the relevant handler to be invoked to implement the RTS operation). These communications don't include those required in implementing `INVOKE` and `TOKEN` and load balancing operations.

- **Total Idle Period**: The total time that this node has been idle, i.e. without any application threads to execute.

- **Number of Idle Periods**: The number of idle periods throughout the application execution, that sum up to make the total idle time.

- **Idle Period Type**: The idle periods are of two types - Application and Balancer. When a node has no threads to execute (both Ready Queue and Token Queue empty), it idles while polling the network. A thread may be ready for execution in two ways - a token may be received as response for request sent earlier, or a token may be generated on this node itself. If the token is received as a result of load balancing, then the idle period is classified as *balancer* related idle period, whereas if the token is generated locally, the idle period is called *application* idle period.

- **Application Elapsed Time**: The total time for the execution of the application, from the start of first thread on node 0, till the end of execution of last thread in the application.

B Appendix - EARTH SP-2 Performance Overview

The performance of the EARTH-SP2[4] is reviewed here. The performance figures shown below, are based on the SP-2 machine with 56 nodes running at 62.5MHz, 128 MB memory, and an instruction/data cache of 32 KB each. The tb-2 card provides the network interface. Peak point-to-point bandwidth is 35 MB/s, and one way node-to-node latency is 30 microsecs.

The portable Threaded-C programs implemented to obtain the timing information of various EARTH operations are the same as those, described in 6. The performance is relatively better on the EARTH-SP2[5], than on the EARTH-SP2 reviewed here, because of faster clock speed, less network latency, and larger memory on the former, not withstanding the lack of homogeniety among it's nodes.

EARTH Operation	Local Operation	Remote Operation	
	EU Costs	Local Costs	Remote Costs
SYNC	301 ns	2486 ns	1895 ns
SPAWN	460 ns	2530 ns	2031 ns
END_THREAD	992 ns	NA	NA
INCR_SYNC	334 ns	2496 ns	1864 ns
DATA_SYNC	427 ns	2849 ns	2282 ns
GET_SYNC	550 ns	2664+2796 ns	4527 ns
INVOKE(1)	413 ns	3167 ns	2736 ns
END_FUNCTION(1)	1402 ns	NA	NA
INVOKE(5)	506 ns	2993 ns	2468 ns
END_FUNCTION(5)	1603 ns	NA	NA
INVOKE(9)	503 ns	2897 ns	2571 ns
END_FUNCTION(9)	1789 ns	NA	NA
INVOKE(18)	611 ns	3732 ns	2849 ns
END_FUNCTION(18)	1982 ns	NA	NA

Table 5: Overhead for Threaded-C instructions on EARTH-SP2

[4] Located at Argonne National Laboratories, Portland, USA
[5] Located at CACR, Caltech, USA

Operation	Local Seq.	Remote Seq.	Local Pipe.	Remote Pipe.
Sync Thread	1773 ns	28699 ns	464 ns	4047 ns
Spawn Thread	1561 ns	28595 ns	NA	NA
Read Word	2156 ns	56742 ns	690 ns	7249 ns
Write Word	1934 ns	56121 ns	609 ns	7077 ns
Fun. Call (1)	3863 ns	60094 ns	2365 ns	8364 ns
Fun. Call (5)	4102 ns	60713 ns	2603 ns	8593 ns
Fun. Call (9)	4354 ns	61077 ns	2863 ns	8761 ns
Fun. Call (18)	4853 ns	63651 ns	3325 ns	9659 ns

Table 6: Overhead costs for EARTH operations on EARTH-SP2

Operation	Local		Remote	
	Overhead	Throughput	Overhead	Throughput
DATA_SYNC_B	810 ns/op	1.24 MB/s	7071 ns/op	0.14 MB/s
DATA_SYNC_S	817 ns/op	2.45 MB/s	7101 ns/op	0.28 MB/s
DATA_SYNC_L	811 ns/op	4.93 MB/s	7066 ns/op	0.57 MB/s
DATA_SYNC_D	918 ns/op	8.71 MB/s	7156 ns/op	1.12 MB/s
GET_SYNC_B	890 ns/op	1.12 MB/s	7095 ns/op	0.14 MB/s
GET_SYNC_S	896 ns/op	2.23 MB/s	7081 ns/op	0.28 MB/s
GET_SYNC_L	908 ns/op	4.40 MB/s	7088 ns/op	0.56 MB/s
GET_SYNC_D	967 ns/op	8.28 MB/s	7159 ns/op	1.12 MB/s

Table 7: Overhead costs for GET_SYNC operation on EARTH SP-2

Align	Local		Remote	
	Single	Dual	Single	Dual
0	69.96 MB/s	69.93 MB/s	34.09 MB/s	39.46 MB/s
16	69.83 MB/s	69.88 MB/s	34.09 MB/s	39.23 MB/s
8	69.83 MB/s	69.75 MB/s	34.12 MB/s	39.72 MB/s
4	71.16 MB/s	71.16 MB/s	32.90 MB/s	33.69 MB/s
1	66.32 MB/s	66.29 MB/s	32.65 MB/s	33.65 MB/s

Table 8: Bandwidth for Blockmove operations on EARTH-SP2