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

Definition of the EARTH Model

Kevin B. Theobald

CAPSL Technical Memo 33
October 6, 1999

This report is adapted from Chapter 4 of the author's doctoral dissertation (May 1999), and has been edited for use as a standalone report.

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

Parallel processing has tremendous potential, but has been disappointing in practice. One reason is that parallel computer designers cannot afford to put the same resources into building processors as the makers of commodity chips, who command a much larger market. Therefore, most parallel architects have turned to off-the-shelf microprocessors, which do not support parallel processing well.

Multithreaded systems based on dataflow principles promise a solution to the problems inherent in many of today's parallel machines. Acceptance of such systems in a world ruled by commodity systems requires taking an *evolutionary* approach, in which the multithreaded machine is initially emulated on an existing multiprocessor based on off-the-shelf microprocessors, while still achieving good performance, and then transformed into more powerful versions by gradually replacing the stock components with custom hardware. EARTH (Efficient Architecture for Running Threads) is a multithreading model suitable for such an evolutionary approach.

EARTH can be defined at several layers of abstraction, including a program execution model, an architecture model, and a language definition. This paper defines the program execution model, which describes the most fundamental characteristics of EARTH which are independent of any particular implementation. Separate sections are devoted to the EARTH thread model, the architecture model, and a set of abstract EARTH operations. A final section discusses possible extensions to the existing model.

The material in this report mostly comes from Chapter 4 of the author's doctoral dissertation [24]. The text has been edited slightly for use as a standalone report, and some introductory text from the dissertation's first chapter has been included. Refer to the dissertation for additional background material and experimental work.

Contents

List of Figures

1 Introduction

The last few decades have produced many possible designs for parallel computer systems. In an ideal world, our desire would be to implement one of these, optimizing all components of the system for achieving large-scale parallelism. In the real world, such an undertaking would be difficult and expensive. Parallel machines must compete with modern commodity microprocessors, whose thousands of engineer-years in design time and billions of dollars in capital costs can be amortized over a large sales volume. For this reason, most parallel machines today are based on off-the-shelf processors and other components.

Unfortunately, neither these processors nor the programming systems that typically run on them provide adequate support for features important to parallel machines, such as latency tolerance and interprocessor synchronization [4, 24]. This will limit their effectiveness for many parallel applications. But these processors are unlikely to include efficient hardware support for parallelism in the near future, since they are designed for a highly competitive uniprocessor market. If we want parallel systems to make effective use of these processors, we must address the problems at a different level.

Specifically, we need a parallel programming model which addresses the performance issues important to parallel machines, yet which can be implemented on a computer built with off-the-shelf processors. The performance of such an implementation may not be ideal, for software cannot always compensate for hardware deficiencies. Nevertheless, the establishment of this model can pave the way for the addition of better hardware support for features of the programming model not supported by current processors.

A solution to this problem is to take an *evolutionary* or gradual approach to building a full-scale multiprocessor [13, 24]. The basic plan is to start with a parallel system based on stock hardware, and move step-by-step toward a fully customized implementation. Each step should be a viable, functional system offering improvements over the preceding system, albeit at some cost. Intermediate steps move some of the functions of the architecture to custom chips or dedicated coprocessors, with the remainder being implemented in off-the-shelf hardware. Software for such a system may follow a similar path making use of off-the-shelf packages, since compilers and operating systems also require large investments.

The following are likely phases in this evolution [24]:

1. Use of an existing parallel system, based on off-the-shelf microprocessors, to emulate a multiprocessing model well enough to demonstrate its viability.

2. Construction of a hybrid system, using off-the-shelf microprocessors to perform the regular computations, and custom auxiliary hardware to support the instructions unique to the multiprocessing model. The custom hardware should improve the performance of the machine compared to the first machine (the emulated system).

3. Design of a hybrid chip containing the original core of the stock microprocessor and the extra custom hardware. The combination of the two components on one chip should reduce

communication delays between the two and allow better sharing of common resources, such as caches.

4. Creation of a fully-integrated processor for a parallel system, one which also performs well in a uniprocessor environment.

It is important to keep transition costs down for each step of the way. This includes not only design costs, but also the costs to the users. Moving an application from one machine to a more "evolved" machine should require no more than recompiling. Since this means the application source code should remain the same from one end of the evolutionary path to the other, the essential characteristics of the model should remain constant over this path. (Of course, as performance improves in each step, the programmer may modify the code to take advantage of these improvements, e.g., to generate more parallelism, but this should not be a requirement.)

The challenge is to construct a parallel programming model which is portable along the path without sacrificing efficiency. To elaborate, our goal is to design a model with the following characteristics:

Efficiency: It adequately solves performance issues, particularly latency, bandwidth, and synchronization [4, 24].

Programmability: It allows programmers to express parallelism in their applications efficiently and easily. This should be true both for applications with regular control and data distributions and for "irregular" applications.

Simplicity: It is simple enough to be implemented at the near end of the spectrum (using an existing off-the-shelf multiprocessor) without sacrificing the performance gains that can be realized by moving to custom hardware at the far end of the spectrum.

Flexibility: The machines at the near end of the spectrum will use commodity hardware, and later machines, though custom-designed, may still make use of existing component designs (e.g., processor cores or modules). This architecture should have the flexibility to take advantage of different components at different stages in the path, or even in the same stage to satisfy different price-performance requirements. Therefore, the architecture shall make as few specifications as necessary of such features as the instruction set, clock speed, bus and memory requirements, etc.

This paper presents the *Efficient Architecture for Running THreads* (EARTH), a multithreading model meeting these goals [10, 11, 12, 13, 17]. EARTH falls into a class of hybrid von Neumann/dataflow machines [9, 14, 24]. However, unlike most other designs in this class, EARTH is amenable to the kind of evolutionary design process described above.

It is very important to clarify what EARTH is and what it is not. Since we are proposing an evolving series of computers based on increasing amounts of custom hardware, EARTH does not refer to any specific machine or design in this series. Instead, it refers to a particular model of multithreading, as presented in this paper, and any machine which can adequately implement this model can be called an EARTH computer. Many real and simulated machines have been made to implement the EARTH model at various points in the evolutionary path [10, 24].

An architecture can be presented at several levels. The most common view of an architecture is the *Instruction Set Architecture* (ISA), which gives specific details of instructions, registers, and their interactions, usually including an operational semantics with enough detail that the programmer can accurately predict a given program's behavior at a specified level of detail. The ISA is generally specific to one processor or family of processors. Alternatively, one can talk about the components of a system at a more abstract level, describing, in general terms, the objects visible to the programmer, the operations which can be performed on these objects, and the general method for representing and executing computations on this machine. We use the term *Program Execution Model* (PXM) for this abstraction.[1] Somewhere between these two points lies a high-level abstraction of the components of the machine and the way they interact, which we call an *Architecture Model.*

The PXM for a conventional sequential machine would include an addressable memory component, some of which is divided into frames placed on a stack. Programs are divided into functions; when the machine executes a function, a frame on the stack is assigned to this function for its private use. Instructions are executed in sequential order, except when a branch instruction explicitly redirects execution to another location. A program counter refers to the next instruction in the sequence to execute. The PXM ignores ISA details such as the opcodes of specific instructions, and is thus universal for all modern general-purpose sequential processors. It is this universality that has made standard programming languages based on this PXM, such as C, so portable among ordinary processors.

We present the PXM for EARTH in the next section. The EARTH PXM extends the conventional PXM above with objects and operators specifically for supporting parallel multithreading. We discuss the representation of programs as collections of threads, the synchronization among these threads, and the context of each thread. This section concludes with a discussion of a memory model and a basic set of primitives supported by the PXM. In Section 3, we present an architecture model for EARTH, a general blueprint for all the real implementations of EARTH [10, 24].

Operations in the EARTH PXM are specified at a very general level. This PXM can serve as the basis for a more detailed specification of the behavior of EARTH [24]. The finer specification would serve the same function as an ISA, in as far as being a target for compilation from a higher-level language. However, a detailed ISA is not suitable for this project, given our desire for flexibility and platform portability, since an ISA would bind EARTH to a specific processor family. Instead, the specification would describe an *EARTH Virtual Machine* (EVM). The EVM is a *partial* specification of the instruction set of an EARTH machine. The EVM defines a set of instructions which must be present in any EARTH computer, and defines the semantics

[1]This concept is similar to Valiant's concept of a "bridging model" [27].

of these instructions in relation to the EARTH PXM. It leaves open both how these EARTH instructions are *implemented* (e.g., in software or hardware) and how they are *represented* (e.g., the opcodes used). Since some implementations of EARTH use off-the-shelf processors, pre-specified opcodes would be meaningless to such processors. Illegal-instruction traps could be used, but these are generally inefficient because of the operating system overheads involved in processing the trap. Instead, the EARTH instructions could be signaled using accesses to memory-mapped I/O or, in the case of a pure software implementation, could be converted to instructions native to the off-the-shelf processor. The EVM only specifies which instructions must exist, and describes the semantics of these instructions.[2]

We have deliberately kept EARTH simple to achieve our goal of portability from one end of the evolutionary path to the other. The early papers on EARTH [12, 13] suggested dividing EARTH into "levels" of complexity, in which higher levels support more features. We sought to identify a minimum set of features necessary to support efficient multithreading, so that an implementation using off-the-shelf processors would be feasible. The implementations in this thesis are based on the lowest level. However, many mechanisms have been proposed over the years for expressing parallelism more easily or efficiently. In the last section of this paper, we consider some which could be added to EARTH, particularly further along the evolutionary path where implementations would be more efficient.

2 The EARTH Program Execution Model

The Program Execution Model for EARTH differs from the PXM of a sequential computer in the following important respects:

- Instead of a single program counter, there can be multiple program counters, allowing concurrent execution of instructions from different parts of the program.
- Programs are divided into small sequences of instructions in a two-level hierarchy of threads.
- The execution ordering among threads is determined by data and control dependences explicitly identified in the program, rather than by program order.
- Frames holding local context for functions are allocated from a heap rather than a linear stack.

The following subsections present the EARTH PXM in more detail. The most unique aspect of this PXM is its threading model, which is covered first. Section 2.2 discusses the EARTH memory model. The final subsection gives a list of the fundamental objects of the EARTH PXM and the operations on these objects which are basic to the smooth functioning of EARTH. Our goal in this section is to justify the need for each feature of the PXM, rather than merely present

[2] Refer to the full dissertation [24] for the EVM definitions.

it. Therefore, a simple example, based on computing Fibonacci numbers, is developed during the discussion.

2.1 EARTH Thread Model

The EARTH *thread model* is the most important defining characteristic of EARTH, distinguishing EARTH both from conventional parallel paradigms and from most other multithreaded machines. What this model has in common with other multithreaded machines is the division of a program into multiple sections of code, generally called *threads*. Multithreaded machines based on conventional processing models divide a program into threads to identify computations which can run concurrently; they *parallelize a sequential program*. Multithreading models derived from the dataflow model, such as those in the previous chapter, combine individual instructions into threads to reduce and amortize the overheads of synchronization and improve data locality; they *sequentialize a parallel program*.

Our goal of making EARTH's PXM suitable for off-the-shelf processors has led to a two-layer hierarchy of *fibers* and *threaded procedures*. The following subsections present this two-layer model and describe its individual components. The first defines *fibers*, which are in the lower layer, and shows how their properties are essential for an efficient off-the-shelf implementation. The second part gives an example of a simple parallel program, and shows the difficulties that this program can present for a parallel machine based on a single-layer model. We show how a two-layer model is an effective solution. The remaining parts present the two layers in greater detail.

2.1.1 EARTH Fibers

In ordinary sequential code, the next instruction executed is completely determined by the preceding instructions and the input data. The EARTH model maintains the ordering constraints among instructions within one thread, but loosens the constraints between different threads, allowing the processor to adapt better to runtime conditions such as unpredictable latencies. In the EARTH model, a thread is a *sequentially-executed, non-preemptive, atomically-scheduled* set of instructions. All three qualifiers are basic to the model, and are necessary for efficient execution on conventional processors. Because many other multithreading systems use the term "thread" for entities with different properties, the EARTH PXM introduces the term "fiber" to refer specifically to the type of thread above.[3] We use the term "thread" only generically, to refer to a block of instructions which can run concurrently with other blocks.

Sequentially-executed means that when a fiber is executed, instructions within the fiber are scheduled according to a sequential semantics. In other words, instructions within the fiber are ordered using an ordinary program counter, which increments to the next instruction unless modified by a branch instruction. Both conditional and unconditional branches may be used,

[3]This term, like "thread" itself, comes from the lexicon of textile making. A fiber is typically a short strand of material. It is the smallest unit in the "thread model" of textiles.

but only to destinations within the same fiber. Modern processors perform sequential execution very efficiently, even when there are many dependences among the instructions, and can takes advantage of the data locality which is usually present due to these dependences. Techniques used by modern superscalar processors to increase the instruction issue rate, such as out-of-order execution and branch prediction, may be used to exploit instruction-level parallelism *within* a fiber, so long as the results are the same as executing the instructions in purely sequential order. "Sequentially-executed" in this case does not mean "one instruction per cycle," but simply that the dynamic ordering of instructions within a fiber conforms to the sequential semantics of the code.

EARTH fibers are also *non-preemptive*. Once a fiber begins execution, it remains active in the CPU until the last instruction in the fiber is finished. If the CPU should stall (e.g., due to a cache miss), the fiber will not be swapped out. This is a fundamental design decision based on the goal of using existing processors. At any point in a fiber's execution, there is likely to be some essential *context* (such as live register values). Ordinary processors don't support rapid context switching, so if a fiber is interrupted, the CPU would have to save the live registers and load some registers for the next fiber.[4] An automatic mechanism for fiber suspension, such as one based on interrupts, would have to make conservative assumptions about which registers are live and would probably save a large number of them. This takes time, both for the triggering of the interrupt and the saving and restoring of registers, and the frequent use of such a mechanism would severely limit system performance.

A corollary of non-preemptive execution is *atomic scheduling*. If a fiber cannot be interrupted, then it should not be started until it is guaranteed to finish without any major stalls. The EARTH implementation is responsible for making this determination (using mechanisms discussed in a later section) and deciding when a fiber can start according to this restriction. Borrowing terms from dataflow [4, 8, 14, 20, 24], we say that when the system decides a fiber is ready to execute, it *enables* the fiber. Since the CPU may still be busy with other fibers at that time, there may be a delay between the time a fiber is enabled and the time it starts running. We call the first state *enabled* and the second state *active*. A fiber that is not ready to begin execution is *dormant*.

Figure 1 shows an abstract model of a machine for executing EARTH fibers. There are two pools of fibers, one for dormant fibers and the other for enabled fibers. When fibers are enabled, they are moved from the dormant pool to the enabled pool. When the *Active Fiber Processor* has free resources for executing a fiber, it takes one from the enabled pool, making it active, allocates the resources needed by this fiber and begins execution. When a fiber finishes execution, the processor returns it to the dormant pool and frees the resources. The EVM should specify a special instruction marking the last instruction of a fiber, in order to simplify the implementation of fibers on off-the-shelf processors, which need to know where the fiber "ends."

[4]Fibers may be interrupted for special exceptions such as arithmetic traps, but these should be assumed to be unusual cases and not normal occurrences such as cache misses.

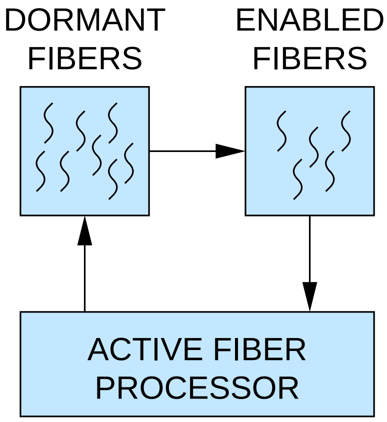


Figure 1: Abstract Fiber Execution Engine

```
int fib(int n) {
    int left, right;
    if (n<2) {              /* S1 */
        return 1            /* S2 */
    } else {
        left = fib(n-1);    /* S3 */
        right = fib(n-2);   /* S4 */
        return left+right;  /* S5 */
    }
}
```

Figure 2: Sequential Fibonacci Example

2.1.2 EARTH Thread Hierarchy

Figure 2 shows a simple recursive program for computing the n^{th} Fibonacci number. Notwithstanding that this is actually a terrible way to compute Fibonacci numbers, it's a good example to illustrate the basic threading model, as well as a good benchmark for measuring multithreading overheads, and will be used throughout this thesis. (More realistic examples of binary recursion are presented in later chapters.) For simplicity, negative inputs aren't checked. Figure 3 illustrates the call graph resulting from calling `fib(4)`.

An obvious way to parallelize this program is to run separate function calls in parallel. For instance, the call to `fib(4)` could spawn separate processes to compute fib(3) and fib(2), and these could run on other processors. But what should be done with the execution of `fib(4)` before `fib(3)` and `fib(2)` have returned their results? If we want to use the processor for some other computation (such as one of the child functions), we must suspend `fib(4)` and switch to another context.

However, this suspension would contradict the non-preemptiveness of EARTH fibers. Fur-

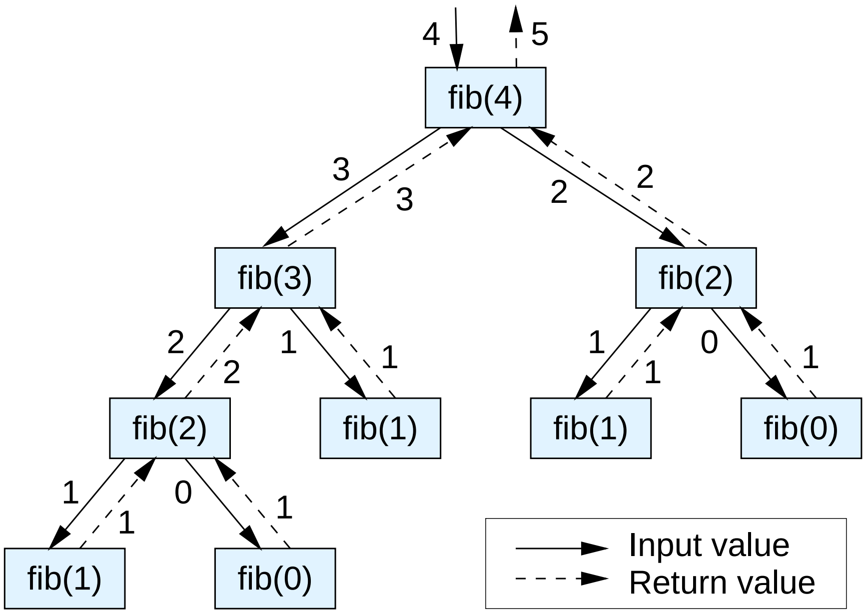


Figure 3: Call Graph for Sequential Fibonacci

thermore, the commodity processors we would like to use in a multithreaded machine have no mechanisms to check that data accessed by an executing fiber is actually valid or to suspend the fiber if it isn't, except for normal register checks such as register scoreboarding. Register checks are enough for sequential computation, because program instructions are in strict sequential order, which is enough to guarantee data dependences are satisfied. However, if `fib(3)` and `fib(2)` run concurrently, their results are returned at indeterminate times, and `fib(4)` must have some way to know that both values have returned before it can add them. Even if `fib(4)` were to call one of the child functions sequentially and fork the other on another processor, it would still need to stall after the sequential return and wait for the return of the value from the parallel function.

The solution is to split the function into several fibers, *each* of which can run non-preemptively. Since the recursive function calls take indeterminate time, the function body should be split after these calls, into two fibers. The first fiber (f_0) executes statements S_1–S_4, that is, tests `n` and either returns 1 or invokes the children. The second fiber (f_1) executes statement S_5, adding the values produced by the children and returning the sum to its parent. Figure 4 shows the threaded version of the call graph for `fib(4)`. Each instance of `fib` has been replaced by a pair of fibers f_0 and f_1.

This example shows a tight coupling between f_0 and f_1. Every instance of f_1 must have been preceded by a corresponding instance of f_0, and most instances of f_0 (except for leaves) lead to a corresponding instance of f_1.[5] Furthermore, paired instances of f_0 and f_1 need to

[5]Throughout this work, an *instance* of a fiber or procedure is a specific dynamic instantiation of a given fiber or procedure, with its own context.

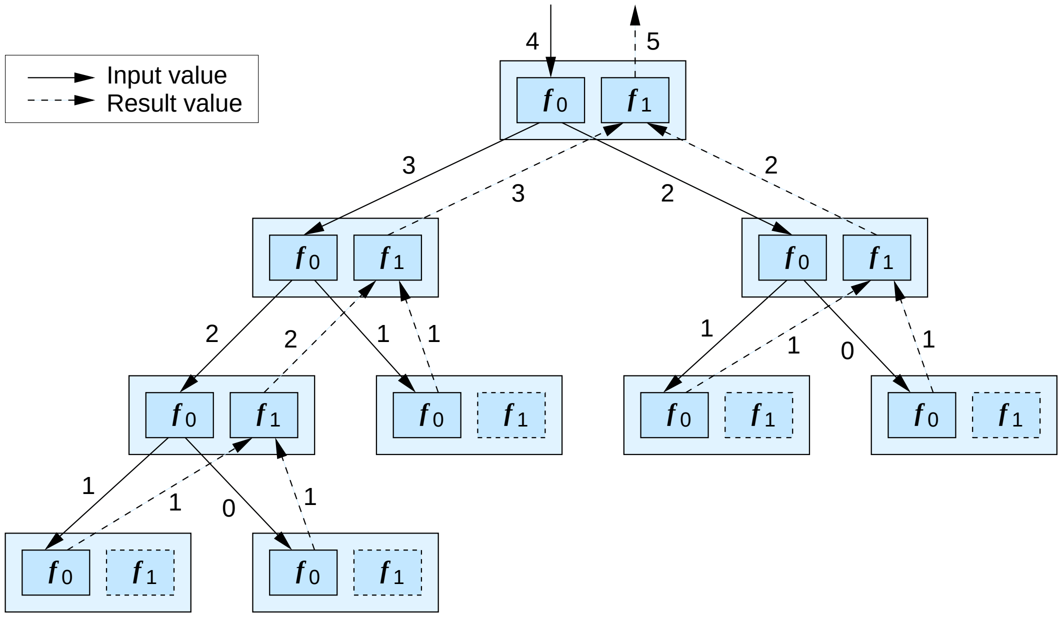


Figure 4: Threaded Fibonacci

share some data. For example, both f_0 and f_1 need access to the partial result variables `left` and `right`. This example shows that the PXM requires something one level higher than an individual fiber.

This need leads to EARTH's *two-level thread hierarchy*. In the Fibonacci example of Figure 4, the `fib` routine is a *threaded procedure*, while f_0 and f_1 are *fibers* within this procedure. Threaded procedures and fibers differ in their contexts, their lifetimes, and their manner of invocation.

The context of an instance of a threaded procedure is similar to the context of a function call in a conventional language such as C.[6] This context includes both local variables and parameters passed to the procedure. Both are accessible by all fibers contained within the threaded procedure. Variables and parameters persist from one fiber to the next, and thus can be used for exchanging values between fibers within the same procedure instance. Fibers have a much smaller context, consisting only of registers and specialized state variables (such as condition codes). Thus, fibers are extremely lightweight, and can be entered and exited quickly, making them suitable for fine-grained tasks where the overheads of normal function context-switching would outweigh the costs of the computation performed. Register values do not persist beyond a fiber's termination, but a fiber can exchange values with other fibers in the same procedure by accessing variables in the procedure context.

Procedures are invoked explicitly by the application program. When the program invokes a procedure, the machine creates a context for this procedure, initializing the input parameters

[6] The term "threaded procedure" is used, rather than "threaded function," because threaded procedures cannot return values, for reasons explained later.

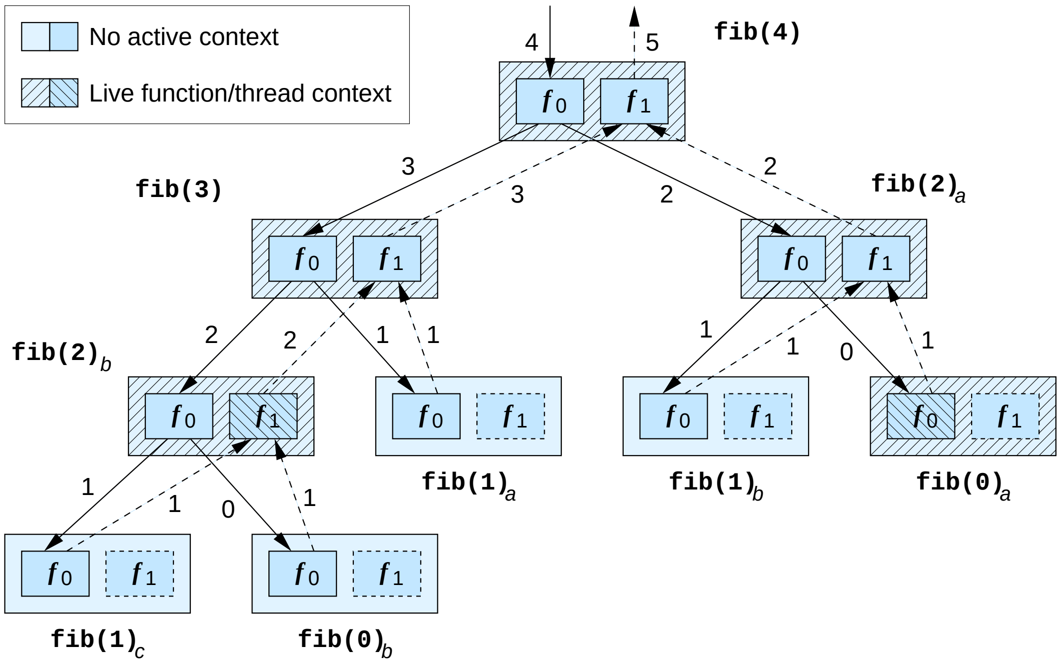


Figure 5: Snapshot of Threaded Fibonacci Context State

with the values passed to this procedure. On the other hand, fibers are started automatically, using a mechanism described in Section 2.1.4. Once a fiber finishes executing, the fiber and its context terminate and are removed from the processor. On the other hand, a threaded procedure instance remains "live" even if none of its fibers are active or enabled; a threaded procedure must explicitly terminate itself.

Figure 5 illustrates context lifetimes. For clarity, procedure instances are labeled, and multiple instances with the same arguments are subscripted a, b, etc. This figure shows one possible state of a two-processor EARTH computer executing `fib(4)`. The topmost instance of `fib` has invoked `fib(3)` and `fib(2)`$_a$. The left child `fib(3)` has invoked its children, and fiber f_1 of its first child `fib(2)`$_b$ is currently running on one processor. Thus, f_1's context (consisting of a few registers) is live. The other child's context is not active, either because it has not been initialized or because it has already run and terminated. Similarly, the other instance `fib(2)`$_a$ has one child whose context is still live and has a fiber f_0 running on the other processor. Thus, two fibers are currently active, and five procedure contexts are live. Three procedure instances currently have no fibers active, but fiber f_1 in each will run once data from their children have produced values.

Some objects may need to exist outside of a particular frame's context. Objects may have a lifetime beyond a single procedure, or may have a size which is dynamic or which can't be determined at the time a frame is allocated, or may be shared among multiple procedures. For this reason, the EARTH PXM includes static variables and a heap for allocating compound data structures such as arrays.

2.1.3 Threaded Procedures

In EARTH, a fiber is always part of an enclosing threaded procedure. All fibers in a procedure share the local variables and input parameters of that procedure. When a threaded procedure is invoked, a new frame is allocated in memory for this particular instance of the procedure. All fibers access this frame through a *frame identifier* (FID), which is part of a fiber's context and is normally kept in a register for quick access. This is similar to the frame pointer found in conventional block-structured languages. Given a frame identifier, it is possible to access any local variable or input parameter in the corresponding procedure instance's context.

However, there are several essential differences between EARTH procedures and conventional functions. The main differences are

1. Frame allocation
2. Invocation and scheduling
3. Parameter passing

Frame allocation: Conventional frames can be kept on a simple linear stack, due to their sequential execution. The state shown in Figure 5 would be impossible for a sequential implementation of Fibonacci because the two functions in the second generation, `fib(3)` and `fib(2)`$_a$, cannot be active at the same time. On a parallel machine, there can be two sibling procedures active at the same time, or even a parent and child procedure running on different processors. A linear stack would not work in either situation.

Therefore, EARTH frames are dynamically allocated from a *heap*. (This was first proposed for dataflow [20] and later used in other multithreaded systems [7, 19].) When a procedure is invoked, the machine must allocate an appropriately-sized block of memory from the heap, and initialize the input parameters with the values passed to the procedure, before any fibers in that procedure instance can start executing. Thereafter, the frame can be accessed via its frame identifier (FID). When a procedure terminates, the block is returned to the heap's free list. Because procedures are explicitly terminated, no garbage collection of frames is needed.

To allow the use of libraries and legacy code, the EARTH PXM also supports conventional *sequential functions*. A sequential function is called from a fiber in the same way it is called from another sequential function. Sequential functions use conventional call/return mechanisms, and a fiber which calls a sequential function suspends until the function returns. While this may appear to violate the non-preemptiveness property, the digression into the sequential code should be viewed as a part of the fiber itself. The sequential code is non-preemptive, and will return control to the calling fiber when completed. To guarantee this, no calls to threaded procedures are allowed within sequential code. Sequential functions require a stack since they may call other sequential functions. The stack is only needed while such sequential code is executing, and can be removed when the current sequential call is terminated. (How the stack is allocated depends on the implementation.)

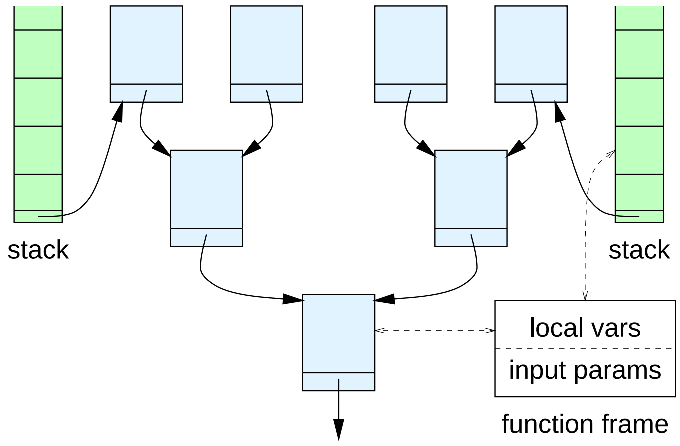


Figure 6: Tree of Procedure Frames with Sequential Stack

Figure 6 illustrates frame allocation for a typical recursive program. Most threaded procedures need to communicate with other procedures, such as their callers. For this, they need references to the other procedures' frames or to variables within the frames. If each procedure has the identifier of the frame of its caller, a tree of procedure frames is generated as shown in the figure. Also shown are two stacks allocated for sequential calls originating from two of the procedure instances, both of which are active. This example shows links from children to parents connecting the call structure together. Most purely recursive divide-and-conquer algorithms will show a similar structure, but the EARTH model is flexible enough to permit an arbitrary interconnection pattern among a set of concurrently executing procedures, such as peer procedures and producer-consumer relationships, as will be illustrated in later chapters.

Invocation and scheduling: When a sequential program calls a function, execution of the caller suspends at the point of the function call and does not continue until the caller returns. When EARTH code invokes a threaded procedure, the fiber invoking the procedure can continue to execute. This is one source of parallelism in EARTH programs, for invoking a procedure can turn one execution stream into two.

A threaded procedure consists of one or more fibers. One fiber in each procedure is called the *initial fiber* and has the special property that it is the first fiber to run when a procedure is invoked. When a procedure is invoked, the system must first allocate and initialize the frame, as described above. Once this is done, the initial fiber is enabled, meaning that as soon as there is spare processing power, the initial fiber can begin executing. All other fibers in the procedure must rely on other mechanisms (described in the next section) to become enabled. Once the initial fiber has finished, it cannot run again in this procedure instance.

Since all fibers within a threaded procedure share data in a single frame, all fibers in a given procedure instance must run on a processor or processors with access to the same context. Typically, locality considerations will require all fibers in a procedure instance to run on the same processor. However, such concerns do not apply to an invoked procedure, which can run on another processor if one is available. The implementation of EARTH (whose programmer interface is specified in the EVM) may allow the caller to select this processor, or the programmer can ask the EARTH system to assign processing resources to the procedure instance automatically. In the latter case, the system will attempt to assign it to lightly-loaded resources, with the goal of achieving a uniform load distribution. The choice of manual or automatic selection of a processor depends on the application; some regular applications have patterns which are easy to balance statically using manual selection, while many irregular applications have a work distribution highly dependent on runtime conditions, requiring dynamic load balancing.

Parameter passing: Threaded procedures, like sequential functions, can pass scalar values to threaded procedures they invoke. However, a threaded procedure cannot return a value to its caller using a conventional value return mechanism. This is because, as stated in the previous section, a fiber which invokes a procedure continues to execute, while the semantics of the return statement imply that the caller waits until the callee finishes so that the value can be used. Therefore, a threaded procedure must send values to its caller using the same mechanism as is used to send data between fibers, which is covered in the next section.

2.1.4 Fibers and Synchronization

In EARTH's two-level thread hierarchy, fibers are the smallest unit of scheduling above individual instructions. Fibers are non-preemptive, so a threaded procedure should be divided into fibers wherever there are likely to be long or unpredictable latencies. This includes threaded procedure invocations, as in the Fibonacci example, and the fetching of data from remote processors.

Each fiber is part of an enclosing threaded procedure. A given instance of a fiber is associated with one particular procedure instance. To maintain this association, the context of a fiber includes a reference to the frame (FID) for this procedure instance, normally kept in a register which is loaded once when the fiber begins execution. The frame identifier (FID) is essential, for it prevents two active instances of a fiber from accessing the same context. For instance, the two active fibers in Figure 5 are supposed to read different copies of the input parameter `n`, and this is guaranteed because they have different frame identifiers.

Within the code of a threaded procedure, each fiber is given a unique identifier called the *instruction pointer* (IP). Each *instance* of a fiber can be uniquely identified by a pair (FID,IP) consisting of a frame identifier, to uniquely identify the frame for the particular instance of the procedure containing this fiber, and an instruction pointer, to specify the fiber within the procedure. As a single unit, the pair (FID,IP) is called a *fiber identifier*.

When a program is divided into procedures and fibers, there will exist (in most cases) data and control dependences among the fibers and procedures. Because the EARTH model can allow fibers to be scheduled in arbitrary sequences, the machine must check and verify dynamically that all data and control dependences have been satisfied before enabling a fiber. This is done explicitly using a mechanism adapted from static dataflow machines [8, 14, 24].

Control and data dependences are made explicit in the EARTH code using *synchronization signals* and *synchronization slots*. A synchronization (or *sync*) signal is sent from one fiber to another, either in the same or another procedure instance, to tell the recipient that a specific control or data dependence has been satisfied. For instance, the sending fiber may have produced data required by the receiving fiber; since the latter could not run before that data was produced, the producer must tell the consumer that the data is now ready. If a fiber depends on more than one datum or control event, it needs to be sure that *all* dependences have been satisfied before it is enabled, since the fiber can't be preempted once started. A counter is used to count the incoming signals so it is known when a fiber is ready to be enabled.

In the Fibonacci example, each non-leaf instance of `fib` receives two integers from procedures that it invokes. Since fiber f_1 adds these integers, it needs both before it can start. Therefore, a count of two is associated with f_1 at the time the procedure instance is initiated. As each datum arrives from a child procedure, the count is decremented. When the count reaches 0, f_1 has all the data it needs and can be enabled.

In static dataflow [8, 14, 24], a count is associated with each actor. Rather than associating a counter with each fiber, the EARTH model separates the counters from the fibers, which allows reuse of the counters for controlling different fibers at different times, or allows several counters to control the same fiber.[7] Counts are maintained in synchronization (or sync) slots. A sync slot is a triple (SC, RC, IP) containing respectively a *sync count,* a *reset count,* and an *instruction pointer*. The instruction pointer (IP) binds the sync slot to one of the fibers in the procedure. The sync count indicates the number of sync signals that have to be received by the sync slot before the specified fiber can be enabled. When a sync signal is received, the sync count is decremented. If the count reaches 0 the fiber specified by the IP is enabled and the sync count is set back to the value of the reset count. The use of a reset count allows fibers to be enabled multiple times, as explained later.

Because sync counts are used for controlling the enabling of fibers, they must persist beyond the lifetime of a fiber, and therefore must be part of the context of a procedure rather than a fiber. They are, in fact, a type of local variable. However, as is shown later, there are some strict ordering constraints between manipulation of sync slots and the sending of sync signals. To ensure portability of the EARTH model across different implementations, sync slots should *not* be accessible to the user, except through special instructions, defined as part of the EVM, which guarantee these ordering constraints are obeyed. Since a sync slot is bound to a particular procedure instance, it can only control fiber instances within that procedure instance. Thus, the third element of a slot only needs to be an IP, rather than a complete fiber identifier, because the FID is fixed.

In the Fibonacci example, each non-root invocation of `fib` produces a value which must be sent to its parent. The sync slot merely counts the number of integers received; there has

[7]One example of the latter is in the use of efficient pipelined producer-consumer programs [24].

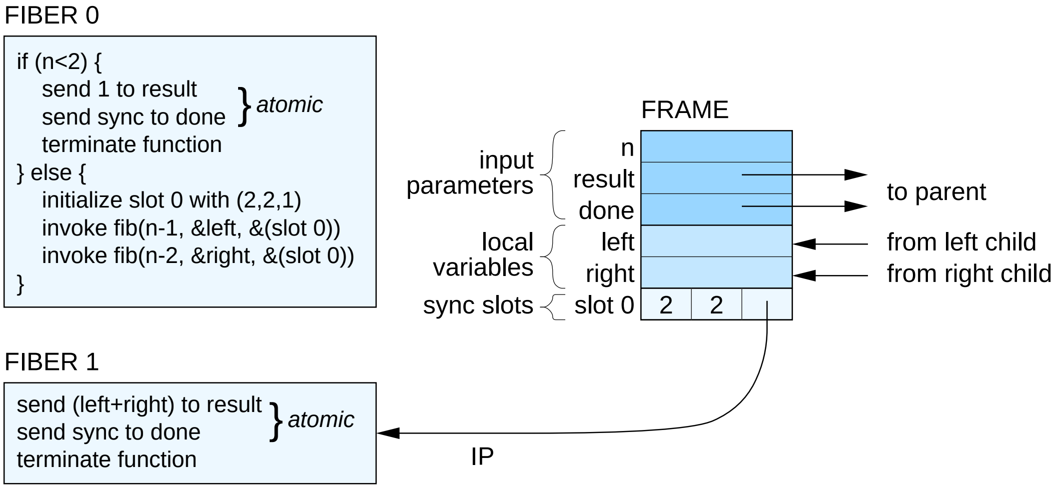


Figure 7: Thread Graph for Fibonacci

been no discussion yet of data transfer. Sending a sync signal is sufficient if both sender and receiver are in the same procedure instance, since local variables can be used to transfer the data from one fiber to another. However, if sender and receiver are in different procedure instances, special mechanisms may be needed to transfer the data, particularly if sender and receiver are on different nodes in a distributed memory machine.

The EARTH PXM provides *atomic* operations for sending data and a sync signal together, which guarantees that the data has been properly transferred before any fibers are enabled as a result of the sync. Such a data-transfer/sync-signal operation may be initiated by the *producer* of data, which sends local data to another location, or by the *consumer* of data, which sends a request for remote data to the system, which retrieves the data and copies it into a local location before signaling the sync slot. The latter operation is called a *split-phase transaction* because the request and data transfer may occur in distinct phases.

Figure 7 shows a graphical representation of an EARTH implementation of parallel Fibonacci, which would produce a call graph like Figure 4. The code is written in a C-like pseudocode since no programming language or EVM for EARTH has yet been presented. The new `fib` procedure takes, in addition to the index parameter `n`, two references `result` and `done`, which refer to an integer and a sync slot, respectively. After `fib(n)` computes the n^{th} Fibonacci number, it sends the answer to the location referenced by `result`, and sends a sync signal to the slot referenced by `done`, bundling the two in an atomic operation. The local variables include two integers for holding intermediate results. if `fib` is not a leaf, it invokes two instances of itself and passes references to these integer locations to the children. A single sync slot is initialized with a count of 2, and a reference to this slot is passed to each child. Thus, when the recursive values are computed, they are stored in the caller's local variables; after the second value arrives, fiber 1 is enabled.

The Fibonacci example has been useful for introducing the EARTH PXM, but it is a trivial

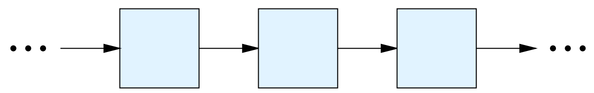


Figure 8: Pipelined Program Structure

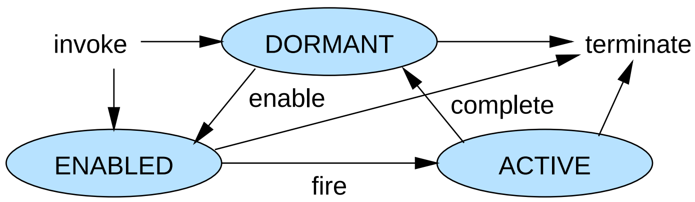


Figure 9: EARTH Fiber States

example which is easily supported by parallel paradigms besides EARTH. The EARTH thread model can be used to partition programs in a way suited to a particular application. Fibonacci is a simple example of binary recursion, in which each fiber is only run once. But many applications require other program structures, such as producer-consumer. For example, the pipeline structure in Figure 8 illustrates an application in which each of a set of modules reads data from its input, transforms it, and sends the result to the next module.

In this application, it is assumed there is a large set of inputs, and parallelism is achieved by running the modules concurrently. Each module needs to run repeatedly in order to process multiple data sets. Otherwise, one or more procedures need to be invoked for each input datum. This could impose large overheads, especially if a lot of state information needs to be passed to a procedure to get it started. It would be better to invoke the procedures once, and allow a fiber to start again whenever new data arrives. Fortunately, this is possible in EARTH.

The complete state diagram for an EARTH fiber is shown in Figure 9. When a procedure is invoked and initialized, all fibers but the initial fiber begin in the dormant state; the initial fiber begins in the enabled state (which is why the "invoke" arc leads to both states). Any active fiber in this procedure instance can initialize sync slots in its frame. Once a sync slot is initialized, any fiber can send sync signals to that slot. If a sync count reaches 0, the fiber whose IP is in the slot is enabled. An enabled fiber is fired (moved to the active state) when processing resources are available to start executing the fiber. After an active fiber is completed, the fiber returns to the dormant state, *where it can be subsequently re-enabled if the sync count returns to 0 again.* Any active fiber can issue a command to terminate the entire procedure, which causes all fibers in that procedure to be removed and the frame deallocated.

It is the ability to re-enable a fiber that has already run that makes producer-consumer synchronizations possible. Fibers can be pipelined in a manner similar to the pipelining used in dataflow graphs [8, 14, 24]. When a fiber has finished processing data and has sent its output to the next module, the fiber terminates; since its sync count was reset, it can be re-enabled when new input data arrives and synchronizes the same sync slot. A real benchmark using this technique is described in a later section.

2.1.5 Benefits of the EARTH Thread Model

The introduction claimed that the EARTH model addresses issues that affect parallel processing: latency, bandwidth, synchronization, programmability, and manufacturability. Having presented the basic model, we can now show in general how features of the EARTH PXM address these issues.

Latency: The multithreading in EARTH does not eliminate interprocessor latencies, but allows computers to *tolerate* latency by performing useful computations while long-latency operations are in progress. This is done by ensuring that the initiators of such operations and the recipients of the data produced by these operations are in different fibers, allowing other fibers to run while these operations are being performed.

Bandwidth: Latency-hiding is not enough to guarantee a high degree of processor utilization. If too many fibers issue remote reads or writes, the network will become saturated and limit overall system performance. While good hardware is important in applications with high communications requirements, the PXM can play a major role as well. The use of frames local to procedure instances encourages writing code with locality in mind. Requiring all dependences to be satisfied before a fiber is enabled may encourage the movement of data in blocks rather than in smaller units, causing network transfer overheads to be amortized. The separation of data communication from the production and consumption of this data removes data transfer from the critical path (because the processor can switch to another fiber), which may give the system's transport layer opportunities to optimize network utilization (e.g., by combining data transfers).

Synchronization: The EARTH PXM separates synchronizations from the computation, allowing synchronizations to be performed independently, possibly by a separate unit. Processors are not blocked waiting for other events to occur, as long as there are other fibers eligible to run. The fiber synchronization mechanism enables the use of non-preemptive fibers, limiting context-switching costs to the boundaries of fibers, where they can be more easily controlled. Unlike dataflow machines, synchronization costs are contained by restricting the instructions within fibers to sequential ordering, where traditional superscalar techniques are effective at achieving limited ILP.

Programmability: Solving the three preceding problems can go a long way toward making parallel machine easier to program [24]. The EARTH PXM further contributes to ease of programming by providing different ways of expressing parallelism, such as divide-and-conquer or producer-consumer, according to what is best for the application. EARTH allows tasks to be divided into an arbitrary and dynamically varying number of procedure instances, and supports automatic load balancing of these instances, which is a boon especially to irregular applications.

Manufacturability: EARTH was designed for efficient execution by off-the-shelf processors. The properties of EARTH fibers (non-preemptiveness, sequential execution, and atomic scheduling) eliminate the need for fine-grain switching between multiple concurrent contexts (as is proposed for some multithreaded processors) and make conventional sequential processors suitable for EARTH.

2.1.6 Summary of the EARTH Thread Model

Under the EARTH model, the instructions of a program are divided into three layers:

1. Threaded procedures,
2. Fibers,
3. Individual instructions.

The upper two layers form EARTH's two-layer thread hierarchy. Each layer defines ordering constraints between components of that layer and a mechanism for determining a schedule which satisfies those constraints.

Individual instructions are at the lowest level. Instructions obey sequential execution semantics, where the next instruction to execute immediately follows the current instruction unless the order is explicitly changed by a branch instruction. Instruction-level parallelism among these instructions is allowed so long as it is consistent with this semantics.

Fibers are collections of instructions sharing a common context, consisting of a set of registers and the identifier of a frame containing variables shared with other fibers. When a processor begins executing a fiber, it executes the designated first instruction of the fiber; subsequent instructions within the fiber are determined by the instructions' sequential semantics. Branch instructions are allowed, but only to other instructions within the same fiber. Calls to sequential procedures are also permitted; such procedure calls are part of the fiber, by definition, even if the code is defined elsewhere. A fiber finishes execution when an explicit fiber-termination marker is encountered. The fiber's context remains active from the start of the fiber to its finish.

Fibers are non-preemptive; once a fiber begins execution, it is not suspended, nor is its context removed from active processing. Thus, fibers are scheduled atomically. A fiber is "enabled" (made eligible to begin execution as soon as processing resources are available) when all data and control dependences have been satisfied. Sync slots and sync signals are used to make this determination. Sync signals (possibly with data attached) tell the recipient that a dependence has been met. A sync slot records how many dependences remain unsatisfied; when this count reaches 0, a fiber associated with this sync slot is enabled, for it now has all data and control permissions necessary to execute. The count is reset to allow a fiber to run multiple times.

Threaded procedures are collections of fibers sharing a common context which persists beyond the lifetime of a single fiber. This context consists of a procedure's input parameters, local variables, and sync slots. The context is stored in a frame, dynamically allocated from the heap when the procedure is invoked. Threaded procedures are explicitly invoked by other procedures. When a threaded procedure is invoked and its frame is ready, the initial fiber is enabled, and can only run once. Other fibers in the same procedure instance may only be enabled using sync slots and sync signals. An explicit terminate command is used to terminate both the fiber which executes this command and its procedure instance, which causes the frame to be deallocated. Since procedure termination is explicit, no garbage collection is needed for these frames.

2.2 EARTH Memory Model

Parallel programming is hobbled by the lack of a universal program execution model [24]. One of the fundamental issues dividing parallel machines is the sharing of memory. Should memory be shared among all processors, allowing a processor to access any memory location in the system, or does each processor have its own private memory, inaccessible to others? To maximize portability, the EARTH model does not take a firm stand on this issue.

Most large multiprocessors use either *distributed memory* or *distributed shared memory*. Distributed memory architectures have separate memories for each processor, or group of processors. These are separate both *physically* and *logically*. A process can access data in remote memory only indirectly, by communicating with a process that has access to that memory. For this reason, they are sometimes called *message passing* machines. Distributed shared-memory architectures have a *global* address space, which allows a processor to access any memory location in the system. Unlike centralized systems, the memory here is distributed among the processors, with some memory physically close to a processor and the rest with other processors. These are also called *Non-Uniform Memory Access* (NUMA) machines. Conventional wisdom says that distributed memory scales more easily than distributed shared memory, while shared memory computers are generally easier to program.

The minimal requirements of the EARTH PXM fall somewhere between these two options. To support the EARTH PXM, a machine's memory system must have the following properties:

1. An active fiber must have direct, low-latency access (through load and store operations) both to its private fiber context and to the frame it shares with other fibers in the same procedure instance.

2. An active sequential function call must have direct, low-latency access both to its local linear stack and to the frame belonging to the fiber which initiated the sequential function call (either directly or through other function calls).

3. Instruction pointers are uniform throughout the system. The code for all threaded procedures and sequential functions is accessible from all processing elements of the machine,

and a given IP value has the same meaning on all such elements. The instruction addresses used in sequential function calls must be the same on all processing elements.

4. All objects in the EARTH system which may be accessible by more than one threaded procedure, including frame identifiers, sync slots, and any data addresses which may be bound with sync signals (as described in Section 2.1.4) must be *globally unique* and accessible by special *EARTH operations.*

The first two requirements are straightforward and ensure rapid execution within a fiber. Requirement (1) would imply, for instance, that on a distributed-memory machine, a threaded procedure instance can run on only one processor. Two fibers from the same procedure instance could not run on different processors because they would not be able to share frame data. Requirement (2) ensures that sequential functions are entered and exited quickly; the second part ensures a rapid return to the initiating fiber.

The third item requires code to be loaded on (or at least available to) all processors. This guarantees that procedures and sequential functions can be invoked on any processing element without any restrictions (other than the first two points above).

The last requirement is critical to the EARTH model, for it allows fibers to communicate with other procedures. Consider the Fibonacci example of Figure 7. When the `fib()` procedure invokes children, it passes references to local variables to its children, and the children write their results to the referred locations using an atomic data-transfer/sync-signal operator. If a child is invoked on another processor, as EARTH allows, then when it does this transfer, the destination argument will refer to a non-local location. This reference must be meaningful on the processor running the child.

Therefore, memory references, at least those passed between procedures, must be globally unique across the machine, and each processor must be able to determine the exact location of any given memory reference. However, note that the requirement above only says that the address must be accessible *to the EARTH operation*; it does not say that the address must be accessible by normal load/store operations. Furthermore, accesses are not expected to be fast, as with the local loads and stores in requirements (1) and (2). These "remote" accesses may have long latencies, but will not necessarily stall the processor because other fibers may run during the accesses.

If loads and store instructions to remote addresses are not required, then it is trivial to form a globally-shared address space by combining a local address with a node identifier. Such a pair can be used whenever an EARTH operator expects a reference. A processor or router that needs to forward the message extracts the node identifier to find the destination of the message, while an SU that accesses memory extracts the local address. Therefore, *if a program, based on the EARTH PXM, accesses non-local memory locations solely through EARTH synchronizing operations, that program can be supported on a distributed memory computer.*

However, it should be emphasized that EARTH's PXM is neither restricted to distributed memory machines nor limited by them. Some distributed shared-memory machines use local caches, driven by complex coherence protocols, to reduce the costs of remote access (*Cache Coherent NUMA* or CC-NUMA architectures). However, even with CC-NUMA machines, there are situations where EARTH's model of fiber synchronization would bring benefits. Shared memory machines still need to enforce data and control dependences, and EARTH provides a simple mechanism which is independent of any memory system. Also, remote latencies will still exist in a shared memory computer even with the addition of local caches.

For example, if the producer of some data signals a remote consumer that the data is available, the producer likely has updated its copy recently, meaning the consumer's cache copy is probably stale. If the consumer tries to read the stale data, the load will miss and stall pending the completion of the remote fetch. If, on the other hand, the producer sends the data to the consumer using an atomic data-transfer/sync-signal operator, the data is guaranteed to be in the consumer's local memory, and no miss is possible. Similarly, explicit fetches by the consumer, if they are likely to miss, can be made on an EARTH machine in a split-phase manner, with the data being used in another fiber. This would allow the local processor to continue execution (or switch to another fiber) in spite of the miss.

The two EVMs presented in the next chapter are designed to work for a distributed memory machine with a globally-shared address space. But these will also work for a shared memory architecture. In fact, shared memory might improve performance by decreasing the fetch time of split-phase transactions.

2.3 EARTH Operations

The previous sections presented an abstract model for representing a program as a hierarchy of instructions, fibers, and threaded procedures whose executions are coordinated using sync slots and sync signals. A model for the execution of a thread-based program was discussed. This execution relies on various operations for sequencing and manipulating the fibers in this hierarchy. These operations perform the following functions:

1. Invocation and termination of procedures and fibers;

2. Creation and manipulation of sync slots;

3. Sending of sync signals to sync slots, either alone or atomically bound with data.

Some of these functions are performed automatically, generally as a result of other EARTH operations. For instance, the sending of a sync signal to a sync slot with a current sync count of 1 causes the slot count to be reset and a fiber to become enabled. Eventually, that fiber becomes active and begins execution. But some operations, such as procedure invocation, are explicitly triggered by the application code. The purpose of this section is to list and define eight explicit (program-level) operations which are essential to a machine implementing the EARTH thread model, and argues why these are essential.

This is still a part of the description of the EARTH PXM, so operations here are still defined at an abstract level. The goal here is to define a common set of primitive operations that must be present in any machine that supports the EARTH PXM. The arguments are defined in general and minimal terms. An EVM, if desired, may include a richer variety of operators or add more arguments to operators to improve efficiency or expressiveness. The next chapter presents two different EARTH Virtual Machines, one of which is used for the experimental studies in this work.

The following data types and functions are used by the eight operators:

FID — A frame identifier is a unique reference to the frame containing the local context of one procedure instance. It is possible to access the local variables, input parameters, and sync slots of this procedure, as well as the procedure code itself, using the FID, in a manner specified by the EVM. The FID is globally unique across all nodes; no two frames, even if on different nodes, have the same FID simultaneously.

IP — An instruction pointer is a unique reference to the start of the code of a particular fiber within a particular threaded procedure. It does not specify a specific *instance* of the fiber. (An FID and IP combined do this.)

PP — A procedure pointer is a unique reference to the start of the code of a particular threaded procedure (but not a specific *instance*). Through this reference, the computer must be able to access all information necessary to start a new instance of a procedure.

SS — This is a reference to a unique synchronization slot, consisting of a sync count (SC), reset count (RC), and fiber (F). The first two are non-negative integers and the third is an IP. The expression $SS.SC$ refers to the sync count of SS, etc. However, this is for descriptive purposes only; these fields should not be manipulated by the application program except through the special EARTH operators listed below. Each SS is associated with a specific instance of a procedure, hence a particular FID, which can be referred to as $FID\_of(SS)$.

The SS type must include enough information to identify a single sync slot which is unique across all nodes. How much information is required depends on the operator and the EVM. In some cases, the sync slot may be restricted to a particular frame, which means that only a number, identifying the slot within that frame, is needed. In other cases, a complete global address is required (such as a pair consisting of a frame ID and a sync slot number).

T — In the list of EARTH operators, type T means an arbitrary object, either scalar or compound (array or record). This class of objects can include any of the reference data types listed above (FID, IP, PP, SS), so that these objects can also be used in EARTH operations (e.g., they can be transferred to another procedure instance). Type T can also include any instance of the reference data type that follows.

reference-to-T — For each object of type T, there is a reference to that object, of type reference-to-T, through which that object can be accessed or updated. In accordance with the memory requirements in Section 2.2, this must be globally unique and all processing elements must be able to access the object of type T using the reference. The term "reference" is used, instead of "pointer" or "address," to prevent any unwarranted assumptions about the kinds of operations that can be performed with these references.

The following lists the eight operations, describing the role of each operation, why it is important, and the behavior which must be supported by the EVM. The list also suggests options which might be added in the EVM. In the list, the "current fiber" is the fiber executing the operation, and the "current frame" is the FID corresponding to the current fiber.

2.3.1 Thread Control Operations

Thread control operations control the creation and termination of threads (fibers and procedures) based on the EARTH thread model described in the previous section. The primary operation is procedure invocation, as illustrated in the Fibonacci example. There must also be operators to mark the end of a fiber and to terminate a procedure. No explicit operators to create fibers are needed, as fibers are enabled implicitly. One fiber is enabled automatically when a procedure is invoked, and others are enabled as a result of sync signals.

A program compiled for EARTH must designate one procedure which is automatically invoked when the program is started. Only one instance of this procedure is invoked, even if there are multiple processors. Other processors remain idle until procedures are invoked on them. This distinguishes EARTH from parallel models such as SPMD, where identical copies of a program are started simultaneously on all nodes.

INVOKE(PP $proc$**, T** $arg1$**, T** $arg2$**, ...)**

This operator invoke procedure $proc$. It allocates a frame appropriate for $proc$, initializes its input parameters to $arg1, arg2$, etc., and enables the IP for the initial fiber of $proc$. The EVM may set restrictions on what types of arguments can be passed, such as scalar values only. The system must guarantee that the frame contents, as seen by the processing element which executes $proc$, are initialized before the execution of $proc$ begins.

TERMINATE_FIBER()

This terminates the current fiber. The processing element which ran this fiber is free to reassign the processing resources used for this fiber, and to begin execution of another enabled fiber, if one exists. (If there are none, the processing element waits until one becomes available, and begins execution.)

TERMINATE_PROCEDURE()

This is similar to TERMINATE_FIBER(), but it also terminates the procedure instance corresponding to the current fiber. The current frame is deallocated. This description does not specify what happens to any other fibers belonging to this instance if they are active or enabled, or what happens if the contents of the current frame are accessed after deallocation. The EVM may define behavior which occurs in these cases, or define such an occurrence as an error which is the programmer's responsibility to avoid.

2.3.2 Sync Slot Control Operations

Sync slots are used to control the enabling of fibers and to count how many dependences have been satisfied. They must be initialized with values before they can receive sync signals. It would be possible to make sync slot initialization an atomic part of procedure invocation. But our experiences with programming multithreaded machines have shown that the number of dependences may vary from one instance of a procedure to the next, and may depend on conditions not known at compile time (or even at the time the procedure is invoked). Therefore, it is preferable to have an explicit operation for initializing sync slots. (Of course, a particular *implementation* of EARTH may optimize by moving slot initialization into the frame initialization stage if the initialization can be fixed at compile time.)

INITIALIZE_SLOT(SS *slot*, int *SC*, int *RC*, IP *F*)

This initializes sync slot *slot*, giving it a sync count of SC, a reset count of RC, and an IP of F. Only sync slots in the current frame can be initialized (hence, no FID is required). Normally, sync slots are initialized in the initial fiber of a procedure. However, an already-initialized slot may be re-initialized, which allows slots to be reused much like registers. (Reusability is explored in a related architecture project which is based on EARTH [18].)

There is the potential for race conditions between the initialization or re-initialization of a thread and the sending of sync signals to that thread. The EVM and implementation should guarantee sequential ordering between slot initialization and slot use within the same fiber. For instance, if an INITIALIZE_SLOT operator which initializes *slot* is followed in the same fiber by an explicit sending of a sync signal to *slot*, the system should guarantee that the new values in *slot* (placed there by the initialization) are in place before the sync signal has any effect on the slot. On the other hand, it is the programmer's responsibility to avoid race conditions *between* fibers. The programmer should also avoid re-initializing a sync slot if there is the possibility that other fibers in the system may be sending sync signals to that slot.

INCREMENT_SLOT(SS *slot*, **int** *inc***)**

This operator increments *slot.SC* by *inc*. Only slots in the local frame can be affected. The ordering constraints for the INITIALIZE_SLOT operator apply to this operator as well.

Although not mandatory for the EARTH thread model, this is a very useful operation for procedures where the number of dependences is not only dynamic, but cannot be determined at the time a sync slot would normally be initialized. An example is traversing a tree where the branching factor varies dynamically, such as searching the future moves in a chess game, where the number of moves to search at each level is determined at runtime. Such an application could be coded in the following way (pseudocode used):

Initial fiber

```
...
INITIALIZE_SLOT(s,1,1,Fiber_2);
...
```

Fiber 1

```
...
m = 0;
for (all potential moves) {
  if (potential move is legal) {
    INCREMENT_SLOT(s, 1);
    INVOKE(search_move,
           &results[m++], s, ...);
  }
}
INCREMENT_SLOT(s, -1);
...
```

In this code, an array is allocated for holding result data, and each child is given a reference to a different location to which the results of one move are sent. Each children sends a sync signal to sync slot `s`, and fiber 2, which chooses a move from among all the sub-searches, should be enabled when all children are done. Since the number of legal moves varies from one instance to the next, the total number of procedures invoked is not known when the slot is initialized in the initial thread. The INCREMENT_SLOT operation is used to add one to the sync count in `s` before invoking a child. If, after the first child is invoked, the child sends a sync signal back before the loop in fiber 1 performs another INCREMENT_SLOT, the count `s`.*SC* could decrement to 0, prematurely enabling fiber 2. To avoid this possibility, the count should start at 1, ensuring that the count is always at least 1 provided the slot is incremented before the INVOKE occurs. When all increments have been performed, it is safe to remove this offset, after which the last child to send a sync signal back will trigger fiber 2. An INCREMENT_SLOT with a negative count (-1) does this. (Actually, a SYNC operation, covered next, would have the same effect, and might be more efficient since it doesn't need an argument.)

2.3.3 Synchronizing Operations

The synchronizing operators give EARTH the ability to enforce data and control dependences between procedures, even those not directly related, enabling the programmer to create many parallel control structures besides simple recursion. Thus, the programmer can tailor the control structures to the needs of the application.

These operators manipulate sync slots as described in Section 2.1.4. Each of the following operators sends a sync signal to a specified slot, which is used to count dependences. However, there may be fibers which only need to wait for one datum or control event, which would imply a sync slot with a reset count of 1. For such cases, the EVM may define special versions of the operators which enable the fiber directly rather than going through a sync slot, saving time and sync slot space. These are optional, however, as the same effect can be achieved with regular sync slots.

The static dataflow machines which inspired EARTH's synchronization mechanism had to avoid the erroneous condition in which two tokens try to exist on the same arc at the same time. There are analogous race conditions in EARTH. One example is enabling a fiber while another instance of the same fiber in the same procedure instance is active or enabled. This could result from having initialized the sync slot with too low a sync count, or, conversely, having generated too many sync signals for that slot. Another error is sending a sync signal to a slot before the slot has been initialized. The exact behavior is left undefined at this level. A specific EVM may specify behavior, or even provide a non-erroneous interpretation for such an occurrence, or indicate to what extent the programmer is responsible to avoid such conditions.

The first case is not, strictly speaking, an error under the EARTH PXM, but will work properly only under special conditions. Figure 10 illustrates the situation arising from having two instances of the same fiber in the same procedure instance simultaneously active. Technically, each fiber has its own context, so it would be possible for the two to run concurrently without interfering with each other. However, note that they still share the same frame, and any input data they require must come from this frame, either directly (the data is in the frame itself) or indirectly (a reference to the data is in the frame), since all local fiber context except the FID itself come from the frame. If both fibers copy the same data and references, they will operate redundantly. If each loads its initial register values from values in the frame and then updates the frame values, it is possible for the fibers to work concurrently on independent data. The figure shows each fiber working with a different element of an array `x`; the snapshot shows the state after each fiber has copied the reference to register `r2`. But correct operation of this code under all circumstances requires two conditions:

1. If the hardware allows the two fibers to run concurrently, it must support atomic access to the frame variable `i`, e.g., a fetch-and-add primitive. These are not universal among

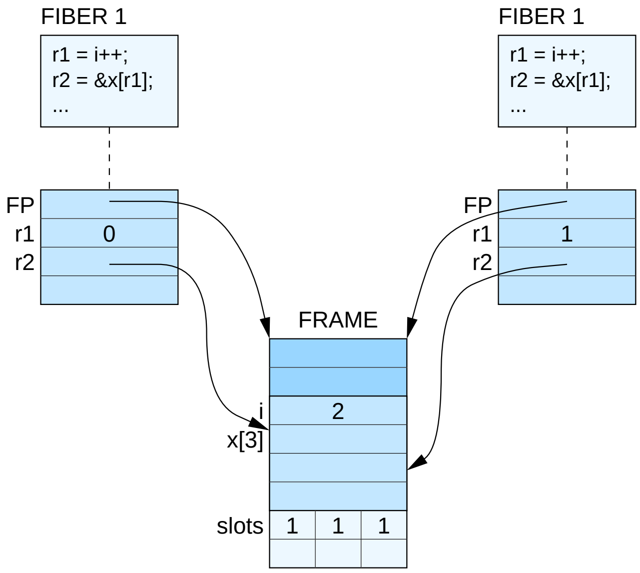


Figure 10: Two Instances of Same Fiber

processors and may require expensive OS solutions.

2. If the fibers were triggered by separate sync signals bound with atomic data transfers (note the first slot in the frame has a count of 1 and triggers fiber 1), the two producers of the data (assume in this case that it is sent to `x[]`) must be programmed to send the two values to separate locations in `x[]`.

While it is possible to program around these restrictions, it would be better if these situations were handled explicitly in the PXM. Section 4 discusses some simple extensions to the EARTH PXM which would simplify this example.

Three basic synchronizing operations are offered by the EARTH PXM: synchronization alone, and both producer-oriented and consumer-oriented versions of synchronization bound with data transfers.

SYNC(SS *slot***)**

This is the basic synchronization operator. The count of the specified sync slot ($slot.SC$) is decremented. If the resulting value is 0, the fiber ($FID\_of(slot), slot.F$) is enabled, and the sync count is updated with the reset count $slot.RC$. Otherwise, the sync count is updated with the decremented value. The implementation must guarantee that the test-and-update access to the SC field is atomic, relative to other operators that can affect the same slot (including the slot control operators).

SYNC_WITH_DATA(T *val*, reference-to-T *dest*, SS *slot*)

Section 2.1.4 mentioned the importance of binding data transfers with sync signals, to avoid a race condition in which a sync signal indicates the satisfying of a data dependence and enabled a fiber *before* the data in question has actually been transferred. This binding is done in EARTH by augmenting a normal SYNC operator with a datum and a reference. The system copies the datum *val* to the location referenced by *dest*, then sends the sync signal to *slot*.

The system must guarantee that the data transfer is complete before the sync signal is sent to *slot*. More precisely, the system must guarantee that, at the time a processing element starts executing a fiber enabled as a direct or indirect result of the sync signal sent to *slot*, that processor sees *val* at the location *dest*. (A direct result means that the sync signal decrements the sync count to 0, while an indirect result means that a subsequent signal to the same slot decrements the count to 0.) The system must also guarantee that, after the sync slot is updated, it is safe to change *val* (this is mostly relevant if *val* is passed "by reference," e.g., as is usually done with arrays).

SYNC_WITH_FETCH(reference-to-T *source*, reference-to-T *dest*, SS *slot*)

The final operator of the EARTH set also binds a sync signal with a data transfer, but the direction of the transfer is reversed. While the previous operator takes a value as its first argument, which must be locally available, the SYNC_WITH_FETCH specifies a location which can be anywhere, even on a remote node. A datum of type T is copied from the source to the destination. The ordering constraints are the same as for SYNC_WITH_DATA, except that *val* (in the previous paragraph) now refers to the datum referenced by *source*.

This operator is primarily used for fetching remote data through the use of *split-phase transactions*. If a procedure needs to fetch data which is likely to be on another node, the fiber initiating the fetch should not wait for the data, which may take a long time. Instead, the consumer of the data should be in another fiber, with a SYNC_WITH_FETCH used to synchronize a slot and enable the consumer when the data is received.

This operation is considered "atomic" only from the point of view of the fiber initiating the operation. In fact, the operation typically occurs in two phases: the request is forwarded to the location of the source data (on a distributed-memory machine), and then, after the data has been fetched, it is transferred back to the original fiber. The SS reference is bound to both transfers, so that the system guarantees the data is copied to *dest before* any fibers begin execution as a direct or indirect result of the sync signal sent to *slot*.

3 The EARTH Architecture Model

The previous sections presented an abstract threading model and a set of operations performing the necessary functions of this model. This section considers the structure and organization of a real machine that can support this model. Since our goal is to support a range of machines containing varying amounts of off-the-shelf hardware, we can only present an *abstract* model of an architectural implementation of EARTH. This section also clarifies terms which were intentionally left vague in the previous section to avoid unwarranted or overly-restrictive assumptions about the hardware architecture.

According to the model, an EARTH computer consists of one or more EARTH *nodes* connected by a network. Each node has the following five essential components:

1. An *Execution Unit* (EU) for executing active fibers;
2. A *Synchronization Unit* (SU) for scheduling and synchronizing fibers and procedures, and handling remote accesses;
3. Two queues, the *Ready Queue* (RQ) and *Event Queue* (EQ), through which the EU and SU communicate;
4. Local memory, shared by the EU and SU;
5. An interface to the interconnection network.

This division allows the implementation of multithreading architectures with off-the-shelf microprocessors mass-produced for uniprocessor workstations [13], one of our primary goals. An EARTH computer is shown in Figure 11.

3.1 Execution Unit

The EU executes individual fibers. As stated in Sections 2.1.1 and 2.1.6, the instructions within fibers are ordered according to sequential semantics. Modern uniprocessors are very efficient at sequential execution. Mechanisms such as bypassing or forwarding [15, 25] reduce hazards due to sequential dependences, while small-scale instruction-level parallelism can be achieved with register renaming, out-of-order execution, register scoreboarding, and branch prediction [22]. Processors using these techniques would make good execution units for EARTH machines.

Most of the instructions executed in the EU are the normal arithmetic, load/store, and branch instructions that come with stock processors. To ensure flexibility, the EARTH model does not specify a particular instruction set. Instead, ordinary arithmetic and memory operations use whatever instructions are native to the EU processors. The context of a fiber is whatever registers are provided by the hardware. The EARTH operations specified by the PXM and EVM are for synchronization and communication. These are unlikely to exist in any form in ordinary processors, but these can be mapped to native EU instructions according to

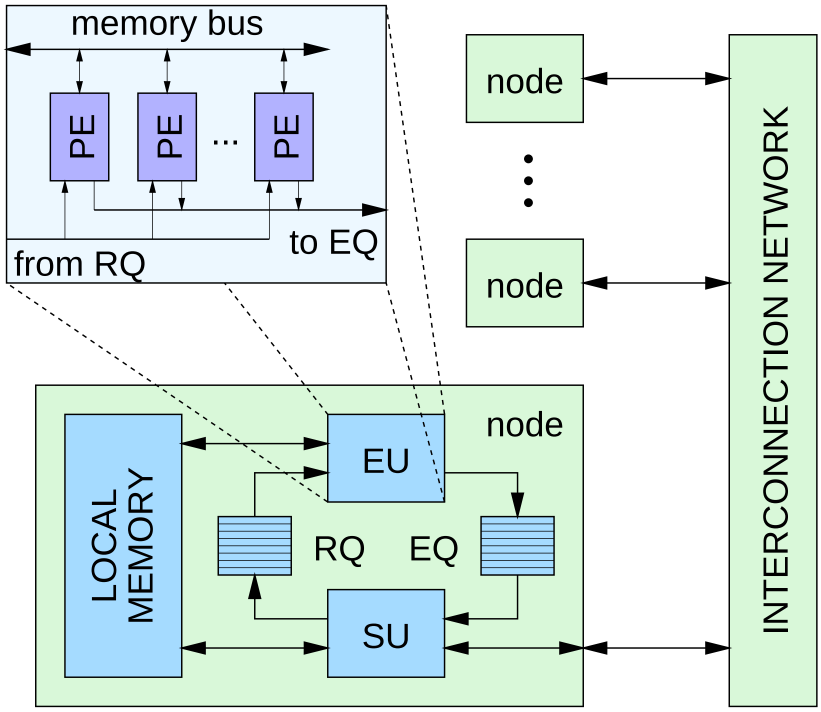


Figure 11: EARTH Architecture

the needs of the specific architecture. For instance, on a machine with ASIC SU chips, the EU EARTH instructions would most likely be converted to accesses to memory-mapped addresses, which would be recognized and intercepted by the SU hardware.

Many details of the EU's behavior have been left unspecified in the EARTH PXM in order to leave considerable latitude in the design of the EU. Nothing in the preceding discussions of the EU should be taken to mean the EU is a single uniprocessor or superscalar executing one fiber at a time, even though this *would* be a correct implementation. In fact, the EU in the model can have processing resources for executing one *or more* fibers simultaneously. This is shown in Figure 11 as a set of parallel *Processing Elements* (PEs). These elements have the following properties:

1. Each PE runs a separate fiber independent of the other PEs.
2. Each PE has a *logically* private context for that fiber.
3. PEs share access to the memory system.
4. All PEs share access to the same Event Queue and Ready Queue, though each PE accesses the queues asynchronous with the other PEs.

Consequently, programmers should avoid unwarranted assumptions about concurrent fiber execution.

The drawing of the EU in Figure 11 is only an abstraction. Any structure may be used so long as it can adhere to this abstraction. For instance, the Execution "Unit" could be a *set* of conventional processors, each executing one fiber at a time (EARTH has been implemented on a system consisting of clusters of Sparc processors [16]). Another possible choice for an EU would be a processor which holds multiple register contexts simultaneously and interleaves between them [1, 2, 21]. Even though such a processor would seem to eliminate the need for a non-preemptiveness requirement, such processors still have only a limited number of register contexts, and can't afford to have too many of them inactive while waiting for hazards to be resolved. The EARTH model would prevent such a situation from happening while taking advantage of an interleaving processor's ability to tolerate short-term latencies, such as floating-point operations. Finally, several processor organizations have been proposed in which multiple execution pipelines feed shared functional units. For instance, Simultaneous Multithreading [26] has demonstrated an ability to use multiple functional units efficiently by sending several threads of instructions to superscalar-like instruction scheduling units feeding the functional units, provided there are enough independent threads to hide instruction latencies. A processor based on such principles would be an excellent platform for EARTH, because EARTH's PXM can provide all the threads needed while EARTH's fiber scheduling policies would reduce the need for these threads.

3.2 Synchronization Unit and Queues

Off-the-shelf processors are suitable for dynamic scheduling of instructions within a single fiber. But they are not good for general-purpose multiprocessing, since they do not adequately address the performance issues discussed in Section 1: latency, bandwidth and synchronization. EARTH's solution is to separate the tasks not supported well by existing processors into a separate unit, called the Synchronization Unit, leaving the EU free to perform the tasks it does best. This has its roots in the Argument-Fetch Dataflow Architecture [8, 14, 24].

The EU and SU messages to each other are buffered using queues called the *Ready Queue* (RQ) and *Event Queue* (EQ). These may be implemented using off-the-shelf devices such as FIFO chips, incorporated into a custom SU, or implemented in software. Various implementations are discussed elsewhere [24].

The EU executes fibers as if they were normal sequential code. The SU does not execute any user code, so all the EARTH operators in Section 2.3 are executed within active fibers running on the EU. When the EU encounters one of these operators, it forwards the operator to the SU, which actually handles the request. The EU writes the request to the EQ, where it stays until being read by the SU.

The SU processes each event it reads from the EQ. Some events will lead to fibers being enabled. Enabled fibers (each fiber is a pair (FID, IP)) are placed in the RQ. When a PE in the EU terminates a fiber, it reads the next entry from the RQ and begins executing that fiber. If the RQ is empty, then the PE becomes idle and must wait until the SU writes a fiber into the RQ. If there is enough parallelism in the program, there will usually be at least one fiber in the RQ, and the EU will never be idle.

3.3 Node Memory

In the EARTH model, each node has memory which is accessible by the SU and all Processing Elements within the EU. Whether this memory is physically and logically local to the node or shared with other nodes depends on the implementation (memory issues are discussed in Section 2.2). There may also be memory which is private to the SU, but some of its functions (such as data transfers) require that it use the memory which is common with the EU.

Memory in most small-scale multiprocessors (as might be used for a *single* EARTH node) is sequentially consistent, because it is easy to support in a single-bus system with snooping caches.[8] However, an EARTH node can get by with a weaker consistency model, so long as it runs correct code which conforms to the EARTH thread model, avoiding the race conditions and other operation errors mentioned in Section 2.3. The use of synchronizations to order fibers according to data and control dependences ensure that fibers can run *only* when explicit sync signals tell the SU that these dependences have been satisfied. Therefore, it is only necessary that data accesses be ordered relative to these synchronization operations (as described in Section 2.3.3), which makes the minimal consistency requirements more like weak ordering.

3.4 Network Interface

On a multinode machine, the final component of an EARTH node is the interface to the network connecting the node to other nodes. To maintain the portability of EARTH applications across machines with different network interfaces, the EARTH model does not provide for direct access to the network by the application code (i.e., the active fibers). As shown in Figure 11, all network traffic goes through the SU.

All the synchronizing operators in Section 2.3.3, as well as the INVOKE operator of Section 2.3.1, can specify remote destinations. Some operations may involve both local and remote references, but such operations will have multiple phases, and the message in each phase will have a specific destination. For instance, a SYNC_WITH_FETCH operation used for a split-phase transaction will usually specify a remote source and a local destination. A message must first go to the remote node where the source is located; this node will send back a message containing the data.

As with other components of the EARTH architecture, the behavior of the network is minimally specified to broaden design choices. However, several generalizations about the network can be made. First, to allow nodes to operate asynchronously, there should be no synchronization required between sender and receiver. For instance, a sender should be able to send a message to another node even if the other node is busy and not explicitly waiting for a receive. This implies a reasonable amount of buffering within the network and the network

[8] This refers only to consistency within a single node.

interface, and some kind of automatic flow control. Second, the implementation (either in the SU or in the network itself) should guarantee reliable end-to-end transmission. Third, to improve performance, some networks may have multiple paths between a given pair of nodes or message priorities allowing some messages to overtake others. Applications programs should *not* assume that messages are received in the same order as sent, and should strictly rely on the EARTH fiber synchronization operations to guarantee correct program behavior.

3.5 Functions of the Synchronization Unit

The SU's primary task is to support the EARTH threading model by implementing the operations listed in Section 2.3. To implement these operations, the SU must perform the following functions:

1. EU and Network Interfacing,
2. Event Decoding,
3. Sync Slot Management,
4. Data Transfers,
5. Fiber Scheduling,
6. Procedure Invocation and Load Balancing.

This section summarizes each of these functions.

EU and Network Interfacing The SU's interfaces with the EU are the Event and Ready Queues. The SU reads the Event Queue regularly and dequeue events from this queue as quickly as possible, in order to prevent the EQ from becoming full, which might cause the EU to stall (either by blocking the write or causing an exception in the EU to be generated, depending on implementation). On the other side, the SU tries to keep at least one enabled fiber in the Ready Queue at all times so that the EU never goes idle.

The SU also needs to maintain contact with the network, since all network traffic must go through the SU. The SU monitors the network interface regularly and reads any incoming messages promptly, and sends outgoing messages to remote nodes whenever the network is available. The interface does whatever protocol conversions are required by the network.

Event Decoding All events (operators) received from either the EQ or the network are decoded to determine what operation is needed. The SU needs to know if the destination is on its node or a remote node. (For split-phase transactions, the SU determines the specific destination of the first uncompleted phase.) For operations with remote destinations, messages are sent to the network interface with the destination node indicated. Local operations are queued for local processing.

Sync Slot Management To support fiber synchronization, the SU manages sync slots, being responsible for their allocation, initialization, and maintenance. The SU decrements slots in response to synchronizing operations, and decides when fibers are enabled.

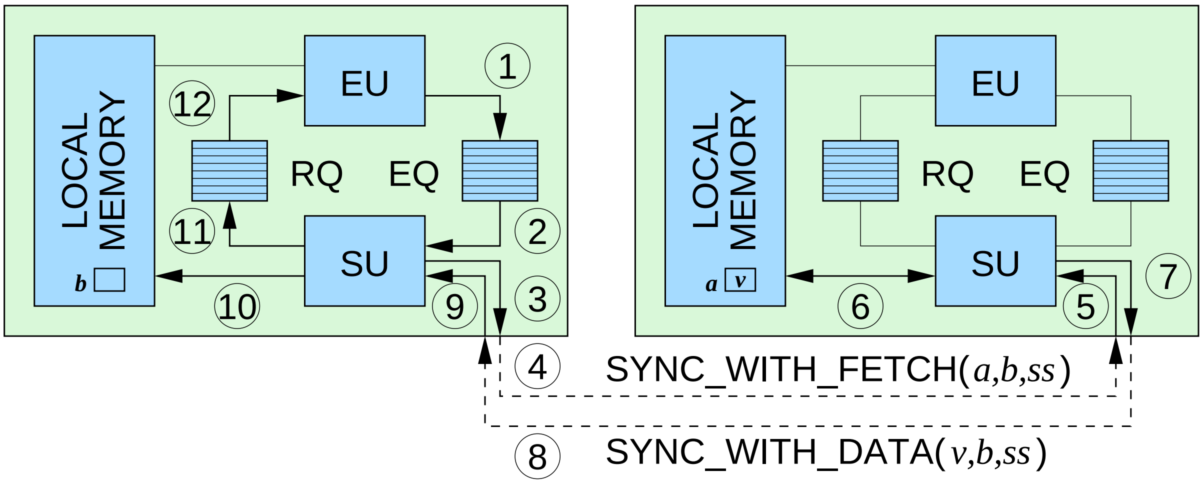
Data Transfers The data transfer operations (SYNC_WITH_DATA and SYNC_WITH_FETCH) require reading and writing local memory, and transferring scalar data and blocks of data over the network. The SU must ensure that each transfer is properly bound with its synchronization signal, so that the ordering constraints in Section 2.3.3 are obeyed.

One benefit of the division of labor between the EU and SU is that split-phase transactions (remote fetches) do not require the intervention of the EU on the remote node. When an SU receives a SYNC_WITH_FETCH request from its local EU, the SU sends a message to the remote node. The remote SU decodes the message, fetches the requested memory contents, and sends the data back to the originating SU. This is illustrated in Figure 12.

Fiber Scheduling The Ready Queue pictured in Figure 11 suggests a simple FIFO between EU and SU for enabled fibers. However, the only restrictions imposed by the EARTH model on fiber scheduling are the synchronization rules implemented by the sync slots. A fiber can begin execution any time after the sync count in its corresponding sync slot reaches 0. While the simplest scheduling policy is a FIFO scheduling in which newly-enabled fibers are placed at the end of the RQ and the EU always read from the beginning, more elaborate policies are possible in implementations that allow efficient random access to this queue [13]. For instance, fibers can be prioritized to favor fibers known to be on critical paths, or fibers can be scheduled in LIFO order to benefit from register locality.

The need for register spills and reloads can be reduced by taking advantage of temporal locality in the scheduling of fibers. This means that the SU can favor (prioritize) fibers in the RQ which are immediately related to the currently executing fiber. (In the Threaded Abstract Machine [7], this would refer to fibers in a *quantum*.) For instance, if the SU determines that a fiber just enabled uses the same frame as a currently-executing fiber, and the registers corresponding to these frame values are the same, it can put the new fiber at the head of the RQ and advance the IP past the (now-redundant) register-loading section of the fiber. If the processor has multiple register sets (as in some current off-the-shelf processors), a *Register Use Cache* (RU-Cache) can keep track of which register set is assigned to which procedure instance. For example, a register file with n register sets will have an RU-Cache of n entries. The RU-Cache would be visible to the SU so that when a new fiber is enabled, the RU-Cache would be associatively searched for a frame identifier which matches the FID of the new fiber. A match in the cache would indicate that that thread should be prioritized. Moreover, the register set to which it is assigned should be included in the RQ entry so that when that fiber identifier is read by the EU, the proper register set would be used.[9]

[9] This is a simple form of the *registering process* in the register-cache of the Super-Actor Machine [9, 14].



1. Local EU sends SYNC_WITH_FETCH(a, b, ss) to EQ
2. Local SU reads event from EQ
3. Local SU determines location of a, sends request there
4. Network transfers message
5. Remote SU reads message from network
6. Remote SU reads value v from reference a in its memory
7. Remote SU sends message to network
8. Network transfers message
9. Local SU reads message from network
10. Local SU writes value v to reference b in its memory
11. Local SU decrements sync count of ss, places fiber in RQ
12. Local EU reads fiber from RQ

Figure 12: Steps in Split-Phase Transaction

Procedure Invocation and Load Balancing The final task of the SU is procedure invocation. Upon receiving an INVOKE request, the SU allocates a frame from the heap, initializes the frame with the procedure's input parameters, and enables the procedure's initial fiber, taking care to obey the ordering constraints given in Section 2.3.1.

That section only stated that procedures are invoked, but did not talk about *where* they are invoked, leaving the details to the specification of the EVM. Since the initial procedure begins on only one node, there must be some way to invoke procedures on other nodes. Two straightforward choices are to let the programmer choose the node or let the EARTH's runtime system or hardware decide. The first choice is highly effective for applications with predictable load distributions, for the programmer has explicit control over where each procedure runs.

However, for many applications, especially irregular ones, the programmer may not have a good understanding of how to distribute data and the workload of the computations, or explicit programmer decision may simply be inconvenient. (An example would be the Fibonacci example used throughout this paper.) For these cases, it is best to let the EARTH system decide where to invoke the procedure. The system uses various heuristics based on the current state of the nodes to choose what it thinks is the best node to run the procedure.

4 Extensions to EARTH

EARTH was originally designed to be a programming model that could be easily implemented on parallel machines based on off-the-shelf processors. For this reason, the operation set was kept to a minimum to reduce implementation complexity, while including enough operators to support efficient parallel processing. While experimental results [10, 24] show that we have largely achieved our goal, there is room for improvement. Our original position papers on the design of EARTH [12, 13] suggested dividing EARTH into "levels" of complexity, in which higher levels support more features. The PXM presented so far in this paper reflects, with minor changes, level 0 in the original concept. Certain features were seen as having potential benefits, but were postponed pending construction of the base machine.

Experience with implementations of our base model has shown that while most parallel algorithms can be represented in EARTH, some operations which are common in parallel programs require somewhat complex or convoluted programming in the EARTH PXM. Some of these were anticipated in our original paper, while others were discovered while programmers were attempting to port specific programs to EARTH. The following extensions to the EARTH PXM involve no fundamental changes to EARTH and could be integrated into the current model.

4.1 Mutual Exclusion and Parallel Reduction

The deterministic nature of EARTH execution makes reduction operations (e.g., adding a set of numbers or finding the max of the set) difficult. Consider the problem of finding the maximum of n floats, each of which is generated by a separate instance of procedure P_1, and using the maximum in a procedure P_2. There are several ways to do it:

1. Allocate an array of n floats in P_2's frame, and pass to each instance of P_1 a reference to a different element of that array. Create a sync slot in P_2 with a count of n. Each P_1 sends its value to its element and syncs the slot in P_2. Then P_2 finds the max in a sequential loop. This is shown in Figure 13.

   This does no more work than any other method ($n-1$ comparisons are needed in any case), but this can be a bottleneck if getting the final max answer in P_2 is in a critical path (as in the tomcatv benchmark [24], in which the reduction in P_2 is part of a barrier synchronization). It can also be problematic if n is unknown or unbounded at compile time.

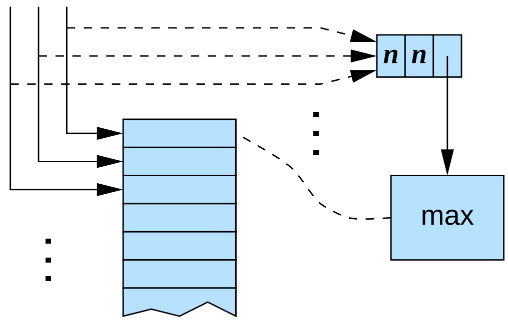


Figure 13: Reduction with Single Fiber

2. Create a binary compare tree with $n/2$ leaves (a fragment is shown in Figure 14). If some of the instances of P_1 send values before others, then there will be some work to do in the tree while other work is still outstanding (assuming there are spare processing resources). This will shorten the amount of time needed by the sequential part which occurs when the last (N^{th}) instance of P_1 returns a value. However, this is complex and requires each call of P_1 to point to one of the leaves of the binary tree. This implies transferring pointers to the instances of P_1 through P_2. The effort might be worthwhile if this reduction is performed repeatedly, but would be a large overhead for a single event. This technique also has difficulties if n changes from one reduction to the next.

3. Traditional methods for guarding critical sections of code (e.g., semaphores) can be employed in the application code. But these impose high overheads on fine-grain applications and are an extra burden to the programmer.

All three methods have been successfully used for benchmarks in EARTH. Some of the programming effort could be removed by providing special reduction library routines. But given the other disadvantages cited above, a more natural and efficient solution would look something like the abstract dataflow graph of Figure 15, in which all values are sent to a single fiber, which saves the internal state (the largest value found so far) until all values have been sent.[10]

The current PXM cannot support this type of graph, for two reasons:

1. If two values are sent to the same place at about the same time, one could overwrite the other before the latter has been used.

2. The general EARTH Architecture Model allows Execution Units to have multiple processing elements. Therefore, there is no way to guarantee atomic access to the critical resource (in this case, the partial maximum).

[10]This non-deterministic merging solution would not work for numerical analysis problems in which the order of floating-point operations is important to the correctness of the algorithm. In such cases, one of the other solutions can be used.

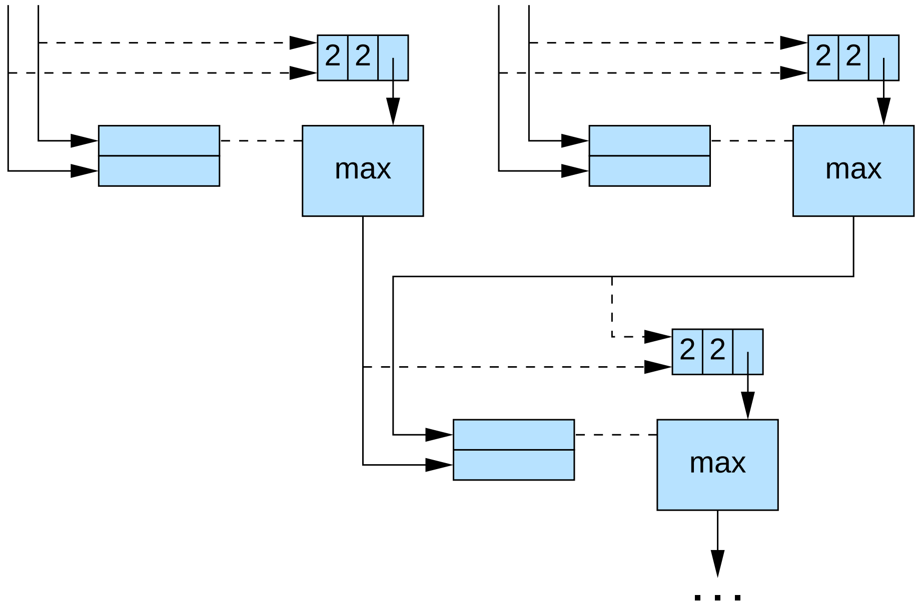


Figure 14: Reduction with Binary Tree

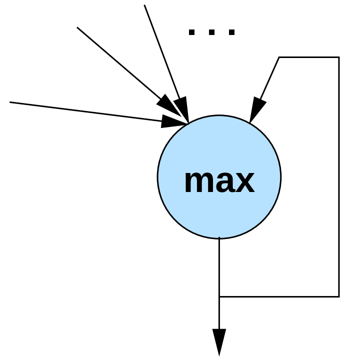


Figure 15: Reduction with Mutual Exclusion

An enhanced EARTH PXM should include some support for mutual exclusion. Fortunately, this is possible with little change.

One of the higher "levels" in the first EARTH design paper [13] allows sync slots to be designated mutually exclusive. If the count in such a slot is decremented and it reaches 0, the fiber is fired as usual, *but the sync count is not reset.* This "locks" the fiber, which should have exclusive access to the critical resource (this is the programmer's responsibility). Any subsequent SYNC_WITH_DATA transfers involving the same sync slot are placed in a queue without updating the memory. When the fiber terminates, the slot's sync count is reset, "unlocking" the fiber. When this occurs, any pending SYNC_WITH_DATA transfers to the same slot may continue.

4.2 Support for Speculative Execution

Many non-numerical programs, and even some numerical programs, have the potential for big improvements in speed if some computation is performed *speculatively*, i.e., before its need has been determined [23]. There are also many applications in which the program can search down many paths in the search space simultaneously, but a success in any subsearch satisfies the search conditions, i.e., an OR condition. The subsearches can often be done in parallel. The generation of parallel searches and speculative procedures and fibers is easy enough, but combining the results in a meaningful way is more complex, because nondeterminism is required. An efficient parallel search creates a race between the subsearches; the (temporally) first subpath to return a successful result is chosen as the solution to the next higher level of the search.

The previous extension for supporting mutual exclusion can be further modified to support this kind of merging. The mutex mechanism allows all incoming SYNC_WITH_DATA messages to be processed eventually. A modified mechanism would permit only one, or perhaps a predetermined number, of such signals to be accepted, causing all others to be discarded.

This would be far more efficient if there were a mechanism for killing computations whose results are no longer needed (e.g., parallel subsearches still in progress). A procedure that wished to kill some of the procedures that it invoked could send a signal to those procedure instances, which in turn would have to kill any processes they created, propagating down the call tree until all descendents are killed. Most of this could be done using the basic PXM, e.g., adding a special "kill" fiber to each threaded procedure. However, the more such features are built into the PXM, the better the hardware support for such operations can be made, leading to fewer redundant or unnecessary computations as superfluous fibers are killed faster.

4.3 Other Extensions

Section 3.5 suggested that the SU could make some fibers jump to the front of the Ready Queue in order to take advantage of register locality. This would be transparent to the programmer. However, priorities could also be made explicit to the programmer (or compiler) by introducing a more general priority mechanism, in which each thread (fiber or procedure instance) is given a priority level. Thus, one could identify critical paths in the code that need to be run quickly, and other parts that can be deferred until processor load is light. Such a priority scheme would also assist in the speculative execution raised in the previous section.

I-structures [5] have been proposed as a convenient way of decoupling producers and consumers, so that consumers could request data without knowing if the data has been produced yet, so long as the consumers know where the data will be when it finally *is* produced. The semantics of I-structures are similar to the EARTH SYNC_WITH_FETCH operator; the consumer requests data in a split-phase transaction. The key difference is that the SYNC_WITH_FETCH operator assumes the data is already present at the specified source location. An I-structure read operation, on the other hand, checks to see if the data has been produced, and does not return data from that location until it is known to be valid. Presense or absence of the data is determined by a full/empty bit attached to each I-structure location. This bit starts empty and is set when a producer writes data to that location. Multiple reads from the same location can be pending; when the location is finally written, the pending split-phase transactions are completed and the data is sent to each requester. A recent extension [6] allows these objects to be reused.

I-structures would simplify some programming, particularly in the beginnings of programs where parallel data and control structures are initialized. Several programs written for EARTH could benefit from I-structures. The semantics of I-structures are close enough to the EARTH SYNC_WITH_FETCH operator that the latter could be easily modified to support the former. The remote SU would simply have to hold onto the SYNC_WITH_FETCH until the data is written. (The PXM should require that the write be done with a SYNC_WITH_DATA operator rather than an ordinary store, so that the SU is alerted when the write occurs.) In fact, I-structures have already been implemented on EARTH in software [3], so it is not essential to add them to the PXM. However, including them in the PXM would mean that they would be supported in the EARTH hardware or system software, and this support would be more efficient than applications-level code.

Acknowledgments

EARTH was a team effort, and would not have been possible without the work of many other people. The people most relevant to this paper (covering the fundamental models of EARTH) are Herbert Hum and Guang R. Gao, who (along with the author) originated the EARTH concept. This was based on earlier work done in the Advanced Compilers, Architectures and Parallel Systems (ACAPS) group at McGill. Olivier Maquelin contributed some important features to the EARTH model, most notably the use of tokens, and also coded the main parts of the EARTH runtime system and compiler almost singlehandedly.

Many others in the ACAPS group at McGill, and the CAPSL group at the University of Delaware, participated in the implementation of EARTH on various systems, work on the compilers, and studies of applications. The National Sciences and Engineering Research Council (NSERC) of Canada provided much of the initial funding for the EARTH project.

References

[1] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, Seattle, Washington, May 28–31, 1990. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News,* 18(2), June 1990.

[2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Conference Proceedings, 1990 International Conference on Supercomputing*, pages 1–6, Amsterdam, The Netherlands, June 11–15, 1990. ACM. *Computer Architecture News,* 18(3), September 1990.

[3] José Nelson Amaral and Guang R. Gao. Implementation of I-structures as a library of functions in Portable Threaded-C. CAPSL Technical Note 04, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, June 1998.

[4] Arvind and Robert A. Iannucci. A critique of multiprocessing von Neumann style. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 426–436, Stockholm, Sweden, June 13–17, 1983. *Computer Architecture News,* 11(3), June 1983.

[5] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[6] P. S. Barth, R. S. Nikhil, and Arvind. M–structures: Extending a parallel, non-strict, functional language with state. CSG Memo 327, Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, Massachusetts, March 1991.

[7] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, Santa Clara, California, April 8–11, 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News,* 19(2), April 1991; *Operating Systems Review*, 25, April 1991; *SIGPLAN Notices,* 26(4), April 1991.

[8] Jack B. Dennis and Guang R. Gao. An efficient pipelined dataflow processor architecture. In *Proceedings of Supercomputing '88*, pages 368–373, Orlando, Florida, November 14–18, 1988. IEEE Computer Society and ACM SIGARCH.

[9] Herbert H. J. Hum and Guang R. Gao. A high-speed memory organization for hybrid dataflow/von Neumann computing. *Future Generation Computer Systems*, 8(4):287–301, September 1992.

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

[11] 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.

[12] Herbert H. J. Hum, Kevin B. Theobald, and Guang R. Gao. Definition of the Multi-Threaded Architecture (MTA) model. ACAPS Technical Note 40, School of Computer Science, McGill University, Montréal, Québec, October 1993.

[13] Herbert H. J. Hum, Kevin B. Theobald, and Guang R. Gao. Building multithreaded architectures with off-the-shelf microprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 288–294, Cancún, Mexico, April 26–29, 1994. IEEE Computer Society.

[14] Herbert Hing-Jing Hum. *The Super-Actor Machine: a Hybrid Dataflow/von Neumann Architecture*. PhD thesis, McGill University, Montréal, Québec, May 1992.

[15] Peter M. Kogge. *The Architecture of Pipelined Computers*. McGraw-Hill Book Company, New York, New York, 1981.

[16] Cheng Li. EARTH-SMP: Multithreading support on an SMP cluster. Master's thesis, University of Delaware, Newark, Delaware, April 1999.

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

[18] Andrés Márquez, Kevin B. Theobald, Xinan Tang, and Guang R. Gao. A super-strand architecture. CAPSL Technical Memo 14, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, December 1997. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[19] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 156–167, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 20(2), May 1992.

[20] James Edward Rumbaugh. A parallel asynchronous computer architecture for data flow programs. Technical Report MIT/LCS/TR-150, MIT Laboratory for Computer Science, Cambridge, Massachusetts, May 1975. PhD thesis.

[21] Burton Smith. The architecture of HEP. In Janusz S. Kowalik, editor, *Parallel MIMD Computation: HEP Supercomputer and its Applications*, Scientific Computation Series, pages 41–55. MIT Press, Cambridge, Massachusetts, 1985.

[22] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 135–148, Minneapolis, Minnesota, May 12–14, 1981. IEEE Computer Society and ACM SIGARCH.

[23] Kevin B. Theobald, Guang R. Gao, and Laurie J. Hendren. Speculative execution and branch prediction on parallel machines. In *Conference Proceedings, 1993 International Conference on Supercomputing*, pages 77–86, Tokyo, Japan, July 20–22, 1993. ACM.

[24] Kevin Bryan Theobald. *EARTH: An Efficient Architecture for Running Threads.* PhD thesis, McGill University, Montréal, Québec, May 1999.

[25] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[26] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 23(2), May 1995.

[27] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.