Multithreaded Systems

Krishna M. Kavi†, Ben Lee‡ and Ali R. Hurson*

† Department of Computer Science and Engineering
The University of Texas at Arlington
Arlington, Texas 76019-0015

‡ Electrical and Computer Engineering Department
Oregon State University
Corvallis, OR 97331

*Computer Science and Engineering Department
The Pennsylvania State University
University Park, PA 16802

This reseach is supported in part by NSF Grants, MIP-9622593 and MIP-9622836.

1. Introduction

The past couple of decades have seen tremendous progress in the technology of computing devices, both in terms of functionality and performance. It is predicted that over the next five years, it will be possible to fabricate processors containing billions of transistor circuits operating at GigaHertz speeds [DARPA97]. While there has been a continuing growth in the density of DRAM memory chips, improvements in the access times and I/O bandwidth of memory parts have not kept pace with processor clock rates. This has widened the relative performance of processors and memory. The memory latency problem is further compounded by complex memory hierarchies which need to be traversed between processors and main memory. In Symmetric Multiprocessors (SMPs), which have become dominant in commercial and scientific computing environments, contention due to the shared bus located between the processor’s L2 cache and the shared main memory subsystem adds additional delay to the memory latency. The delays becomes even more severe for scalable Distributed Shared Memory (DSM) systems that span the spectrum; from systems with physically distributed memory and hardware support for cache coherency, to Networks of Workstations (NOWs) interconnected by a LAN or WAN and software support for shared-memory abstraction. In either case, a miss on the local memory requires a request to be issued to the remote memory, and a reply to be sent back to the requesting processor. Stalls due to the round-trip communication latency are and will continue to be an aggravating factor that limits the performance of scalable DSM systems.

Memory latency, while growing, is not a new phenomenon. There have been varied efforts to resolve the memory latency problem. The most obvious approach is to reduce the physical latencies in the system. This involves making the pathway between the processor requesting the data and the remote memory that contains the data as efficient as possible, e.g., reducing the software overhead of sending and receiving messages and improving the connectivity of networks. The second approach is to reduce the frequency of long latency operations, by keeping data local to the processor that needs it. When data locality cannot be exploited, prefetching or block transferring (as opposed to cache-line transfers) of data can be used. Caches are the most prevalent solution to the problem of memory latency. Unfortunately, they do not perform well if an application’s memory access patterns do not conform to hard-wired policies. Furthermore, increasing cache capacities, while consuming an increasingly large silicon areas on processor chips, will only result in diminishing returns.

Although the aforementioned approaches reduce latency, they do not eliminate it. *Multithreading* has emerged as a promising and exciting avenue to tolerate the latency that cannot be eliminated. A multithreaded system contains multiple "loci of control" (or threads) within a single program; the processor is shared by these multiple threads leading to higher utilization. The processor may switch between the threads to not only to hide memory latency but other long latency operations, such as I/O latency, or interleave instructions on a cycle-by-cycle basis from multiple threads to minimize pipeline breaks due to dependencies among instructions within a single thread. Multithreading has also been used strictly as a programming paradigm on general purpose hardware to exploit thread parallelism on SMPs and to increase applications' throughput and responsiveness. However, lately, there is an increasing interest in providing hardware support for multithreading. Without adequate hardware support, such as multiple hardware contexts, fast context-switch, non-blocking caches, out-of-order instruction issue and completion, register renaming, we will not be able to take full advantage of the multithreading model of computation. As the feature size of logic devices reduces, we feel that the silicon area can be put to better use by providing support for multithreading.

The idea of multithreading is not new. Fine-grained multithreading was implicit in the dataflow model of computation [Lee94]. Multiple hardware contexts (i.e., register files, PSWs) to speed up switching between threads were implemented in systems such as Dorado [Pier83], HEP [Smith85], and Tera [Alverson90]. Some of these systems were not successful due to a lack of innovations in programming languages, run-time systems, and operating system kernels. There is, however, a renewed interest in multithreading primarily due to a confluence of several independent research directions which have united over a common set of issues and techniques. A number of research projects are underway for designing multithreaded systems that include new architectures, new programming languages, new compiling techniques, more efficient interprocessor communication, and customized microkernels. Some of these projects have produced substantial improvements over single threaded abstractions. The success of multithreading as a viable computational model depends on the integration of these efforts.

This article is organized as follows: Section 2 discusses multithreading in terms of user-level programming models, such as TAM [Culler93], Cilk [Blumofe95], and Cid [Nikhil94]. Section 3 reviews the execution models and run-time support of multithreading. Thread libraries and kernel-level thread support will be the main focus of this section. Section 4 discusses the architectural support for multithreading with emphasis on reducing the cost of context switching. Section 5 provides an overview of various multithreaded architectures along with their key features. The survey includes Tera MTA, StarT, EM-X, Alewife, M-Machine, and Simultaneous Multithreading. Section 6 presents analytical models for studying the performance of multithreading. Finally, Section 7 concludes the article with a brief discussion of future developments and challenges in multithreading.

2. Programming Models

Multithreading has become increasingly popular with programming language designers, operating system designers, and computer architects as a way to support applications. In this section we will concentrate on multithreaded models as seen from a programmer perspective. Concurrency can be supported by a programming languages in many ways. It can be achieved by providing user-level thread libraries to C and C++ programmers, whereby the programmer can insert appropriate calls to these libraries to create, invoke, and control threads. A variety of such libraries have been available to programmers, including C-threads, Pthreads, and Solaris Threads. We will discuss these libraries in the next section.

Some programming languages provide concurrency constructs as an integral part of the language. ADA-95 permits users to create and control concurrent programming units known as tasks [ISO 94]. Synchronization among tasks can be achieved using either shared-memory (protected objects) or message-passing (rendezvous using select and accept statements). Consider the following function which forks (recursively) threads to compute Fibonacci numbers. Forking of tasks is accomplished by allocating a pointer type that points to a task type. Each new task spawns two additional tasks to compute Fib(N-1) and Fib(N-2), and waits for the results from the spawned tasks.

In most implementations, individual ADA-95 tasks of a program are bound to threads provided by the system (either kernel-level or user-level threads). ADA-95 facilitates a various means for creating, initiating, and managing synchronization among tasks. Single tasks are scheduled as soon as the block in which they are defined is entered. Variables of task types are enabled for execution as soon as the body containing the variable declarations is entered. Access (pointer type) variables to tasks types become enabled when allocated. Task cease to exit when they complete execution, and only when all their child tasks complete executions. Tasks can be explicitly aborted. The primary synchronization in ADA is the rendezvous mechanism using select and accept statements. Entry points can be guarded. In ADA-95, the concept of protected objects is introduced to implement monitors and conditional waiting inside a monitor.

```
Function Fibonacci (N : In Integer) Return Integer Is

Task Type Fib Is
---
--- This is the task specification (prototype).
--- Task type is declared here with two entry points. Tasks can rendezvous at these entry points
----
          Entry Get_Input  (N: In Integer);
          Entry Return_Result (Result : Out Integer);
End Fib;

Type Fib_Ptr Is Access Fib;      -- A pointer to the task type is defined here

Function Create_Fib_Task Return Fib_Ptr Is
Begin
---
--- This function is used to create and spawn new tasks of type Fib by allocating the pointer type.
--- The function is needed to eliminate recursive definition inside the task body below.
----
          Return New Fib;  --- The construct New allocates the task
End Create_Fib_Task;

Task  Body Fib Is
---
--- This is the task body for the task Fib
---

Input, Result_N, Result_N_1, Result_N_2 : Integer;
Fib_N_1, Fib_N_2 : Fib_Ptr;

Begin
          Accept Get_Input (N : In Integer) Do
          ---
          --- This entry point is used to receive the argument
          ---
                    Input := N;
```

```
    End Get_Input;

    If (Input <= 2) Then
        Result_N := Input;
    Else
        Fib_N_1 := Create_Fib_Task; -- create a new thread to compute Fib (N-1)
        Fib_N_2 := Create_Fib_Task; -- create a new thread to compute Fib (N-2)
        Fib_N_1.Get_Input (Input - 1); -- The spawned task Fib(N-1) receives the argument n-1
        Fib_N_2.Get_Input (Input - 2); -- The spawned task Fib(N-2) receives the argument n-2

        Fib_N_1.Return_Result (Result_N_1); -- Receive the result from task Fib(N-1)
        Fib_N_2.Return_Result (Result_N_2); -- Receive the result from task Fib(N-2)
        Result_N := Result_N_1 + Result_N_2;

        Accept Return_Result (Result : Out Integer) Do
        ---
        --- This entry point is used to return the result to the parent
        ---
            Result := Result_N;
        End Return_Result;
    End If;
End Fib;

--- This is the main procedure that contains the task Fib declaration.

Result : Integer;
Fib_N := Fib_Ptr;

Begin
    Fib_N := Create_Fib_Task;
    Fib_N.Get_Input (N);
    Fib_N.Return_Result (Result);
    Return Result;
End Fibbonnacci;
```

Java programming language supports multithreading by defining classes for creation and synchronization of threads [Berg 95]. Consider the following Java implementation of Fibonacci numbers. As in the ADA-95, Java threads are blocking (and coarse-grained). The parent thread that created and started two new threads to compute Fibonacci(N-1) and Fibonacci(N-2) must wait for the threads to complete using a barrier synchronization “Join”. Java is based on C++. In the example below, a class Fibonacci is defined as thread class. In the body of the class, two new threads for computing Fibonacci of N-1 and N-2 are created recursively. The parent will wait (using Join) the two child threads complete execution; and the values returned by the child threads are added. Some of the characteristic of Java threads are listed in the next section.

```
public class Fibonacci extends Thread
```

```
{       int fib;
        Fibonacci(int n)
        {       fib = n;
        }
        public void run()
        {       if(fib == 0 || fib == 1)
                {       fib = 1;
                }
                else
                {       Fibonacci thread1 = new Fibonacci(fib-1);  // create a child thread for N-1
                        Fibonacci thread2 = new Fibonacci(fib-2); // create a child thread for N-2
                        thread1.start(); // execution of created thread starts here.
                        thread2.start();// execution of created thread starts here.
                        try
                        {       thread1.join(); // wait for child threads
                                thread2.join();// wait for child threads
                                fib = thread1.getFib() + thread2.getFib();
                        }
                        catch( InterruptedException e)
                                // Java requires this section to handle exceptions
                                {       e.printStackTrace();
                                }
                }
        } // end of run()
        public final int getFib()
        {       return fib;
        }
        public static void main(String arg[])              // this is the main program
        {       Fibonacci fib;
                int n = new Integer(arg[0]).intValue();
                fib = new Fibonacci(n);
                fib.start();
                try
                {       fib.join();
                        System.out.println("The Fibonacci for "+ n + " is: "+ fib.getFib());
                }
                catch( InterruptedException e)
                {       e.printStackTrace();
                }
        }
}
```

Programming languages with support for multithreading normally permit coarse-grained and blocking threads. The blocking nature requires synchronization among the threads using such common techniques as mutual exclusion using semaphores or mutexes, condition variables, events, rendezvous, guards and monitors. They provide for thread scheduling constructs such as yield, suspend, detach, abort or terminate. Some functional programming languages such as Multilisp [Halstead85] and Id90 [Nikhil91] have proposed a different attack on multithreading, often supporting fine-grained threads. In such languages, actions that traditionally block or synchronous are made nonblocking and asynchronous. For example, in traditional von Neumann languages, function calls are synchronous: when a function is invoked, the thread of control is transferred to the called function (blocking the execution of the caller) and the control is returned to the caller upon its completion. In Multilisp, function calls (called futures) are nonblocking so that several futures can be invoked without waiting for their completion. Likewise languages can be designed with other asynchronous or nonblocking actions. In general, a multithreaded programming language may permit programs where even conditional statements can be made asynchronous. Languages based on data-driven model of synchronization support fine-grained and nonblocking threads. In such systems, a thread is not ready for execution until all its synchronizations requirements are satisfied; and once initiated, the thread executes to completion with no further synchronization requirements. In the remainder of this section, we will introduce three such languages.

2.1 Threaded Abstract Machine (TAM)

TAM [Culler93] has its roots in the dataflow model of execution, but can be understood independently of dataflow. A language called Threaded Machine Language, TL0, was designed to permit programming using the TAM model. TAM recognizes three major storage resources—code-blocks, frames, and structures—and the existence of critical processor resources, such as registers. A program is represented by a collection of re-entrant code-blocks, corresponding roughly to individual functions or loop bodies in the high-level program text. A code-block comprises a collection of threads and inlets. Invoking a code-block involves allocating a frame—much like a conventional call frame—depositing argument values into locations within the frame, and enabling threads within the code-block for execution. Instructions may refer to registers and to slots in the current frame: the compiler statically determines the frame size for each code-block and is responsible for correctly using slots and registers under all possible dynamic thread orderings. The compiler also reserves a portion of the frame as a continuation vector, used at run-time to hold pointers to enabled threads. The global scheduling pool is the set of frames that contain enabled threads.

Executing a code-block may fork several frames concurrently, since the caller is not suspended as in a conventional language. Therefore, the set of frames in existence at any time form a tree (the activation tree) rather than a stack, reflecting the dynamic call structure. This is shown in Figure 1. To allow greater parallelism and to support languages with non-strict function call semantics, the arguments to a code-block may be delivered asynchronously. An activation is enabled if its frame contains any enabled threads. At any time, a subset of enabled activations may be resident on processors.

Threads come in two forms, synchronizing and non-synchronizing. A synchronizing thread specifies a frame slot containing the entry count for the thread. Each *fork* to a synchronizing thread causes the entry count (synchronization count) to be decremented, but the thread executes only when the count reaches zero, indicating that all synchronization requirements were met. A non-synchronizing thread is ready for executing immediately. Synchronization occurs only at the start of a threads: once successfully initiated, a thread executes to completion. Fork operations may occur anywhere in a thread, causing additional threads to be enabled for execution. An enabled thread is identified by a continuation—its instruction pointer and frame pointer. A thread ends with an explicit *stop* instruction, which causes another enabled thread to be scheduled. Conditional flow of execution is supported by *switch*, which forks one of two threads based on a boolean input value. The compiler is responsible for establishing correct entry counts for synchronizing threads. This is facilitated by allowing a distinguished initialization thread in each code-block, which is the first thread executed in an activation of the code-block. Long latency operations, such as I-Fetch or Send, implicitly fork a thread that resumes when the request completes. This allows the processor to continue with useful work while the remote access is outstanding.

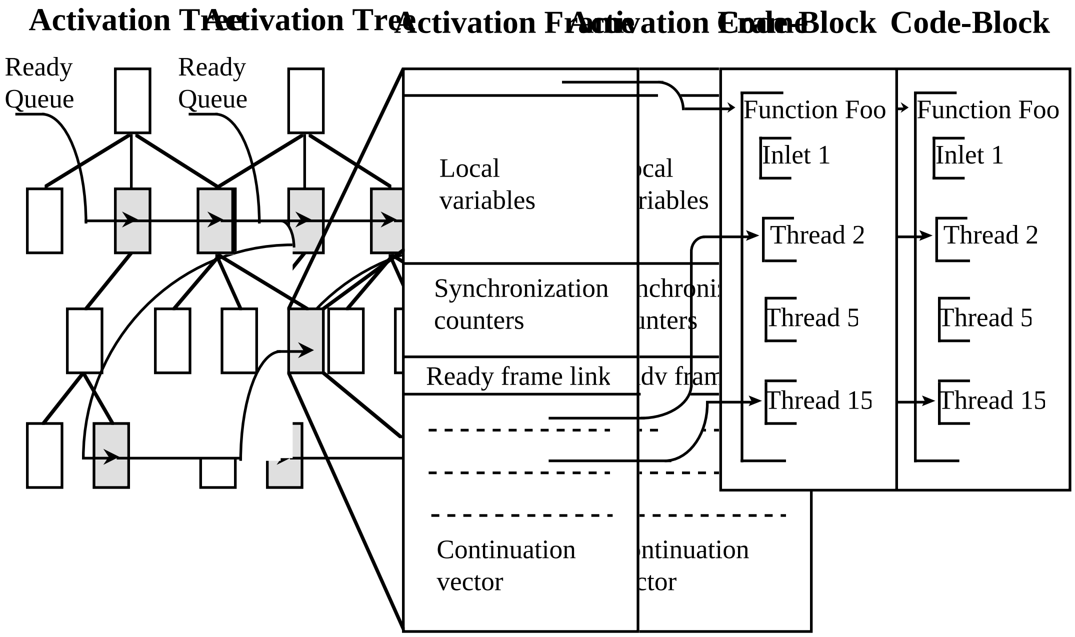


Figure 2: TAM activation tree

The storage hierarchy is explicit in TAM. In addition, scheduling is explicit and reflects the storage hierarchy. In order to execute threads from an activation, the activation must be made resident. When an activation is made resident on a processor, it has access to processor registers. Furthermore, it remains resident and executing until no enabled threads for the activation exist. The set of threads executed during a single residency is called a quantum.

The following is an implementation of the Fibonacci program in TL0:

```
FRAME_BODY RCE = 3                  -- defines a frame with 3 arguments
        islot1.i, islot1.i, islot2.i        -- one argument and two results
        pfslot1.pf, pfslot2.pf              -- frame pointers for recursive calls
        sslot0.s                            -- synchronization variable for thread 6
        pfsloto.pf, jsloto.j                -- parent's frame pointer and inlet
REGISTER                                    -- Registers used
        breg0.b, ireg0.i                    -- boolean and integer temps
INLET 0                                     -- recv parent frame ptr, retrun inlet and argument
        RECEIVE pfslot0.pf, jslot0.j, isloto.i
        FINIT                               -- initialize frame
        SET_ENTER 7, t                      -- set enter-activation thread
        SET_LEAVE 8, t                      -- set leave-activation thread
        POST 0.t
        STOP
INLET 1                                     -- receive frame pointer of first recursive call
        RECEIVE pfslot1.pf
        POST 3.t
        STOP
INLET 2                                     -- receive result from first recursive call
```

```
        RECEIVE islot1.i
        POST 5.t
        STOP
INLET 3                                     -- receive frame pointer of second recursive call
        RECEIVE pfslot2.pf
        POST 4.t
        STOP
INLET 4                                     -- receive result from second recursive call
        RECEIVE islot2.i
        POST 5.t
        STOP
THREAD 0                                    -- test argument against 2
        LT brego.b = islot0.i 2 i
        SWITCH breg0.b 1.t 2.t
        STOP
THREAD 1                                    -- if argument <2, return argument
        MOVE ireg0.i = 1.i
        FORK 6.t                            -- thread 6 retuns this value
        STOP
THREAD 2                                    -- allocate frames for recursive calls
        MOVE sslot0.s = 2.s                 -- set synchronization counter
        FALLOC 1.j = FIB.pc
        FALLOC 3.j = FIB.pc
        STOP
THREAD 3                                    -- send n-1 to first recursive call
        SUB ireg0.1 = islot0.1 1. i
        SEND pfslot1.pf[0.i] <-fp.pf 2.j ireg0.i
        STOP
THREAD 4                                    -- send n-2 to second recursive call
        SUB ireg0.1 = islot0.1 2. i
        SEND pfslot2.pf[0.i] <-fp.pf 4.j ireg0.i
        STOP
THREAD 5                                    -- waits for results from both calls
        SYNC sslot0.s
        ADD ireg0.i = islot1.1 islot2.i     -- add the two results
        FORK 6.t
        STOP
THREAD 6                                    -- send result to parent
        SEND pfslot0.pf[jslot0.j] <- ireg0.i
        FREE fp.pf
        SWAP                                -- swap to next activation
        STOP
THREAD 7                                    -- enter point for this activation
        STOP
THREAD 8                                    -- leave this activation
        SWAP
        STOP
```

Here, Thread 0 checks if the argument received is less than 2. If the value is greater than 2, two new fibonacci activations are allocated (corresponding to the recursive calls). The allocation of frames is performed by Thread 2. It is possible to indicate that the activations be executed either on local or a remote processor. The arguments n-1 and n-2 for the two recursive calls are computed and

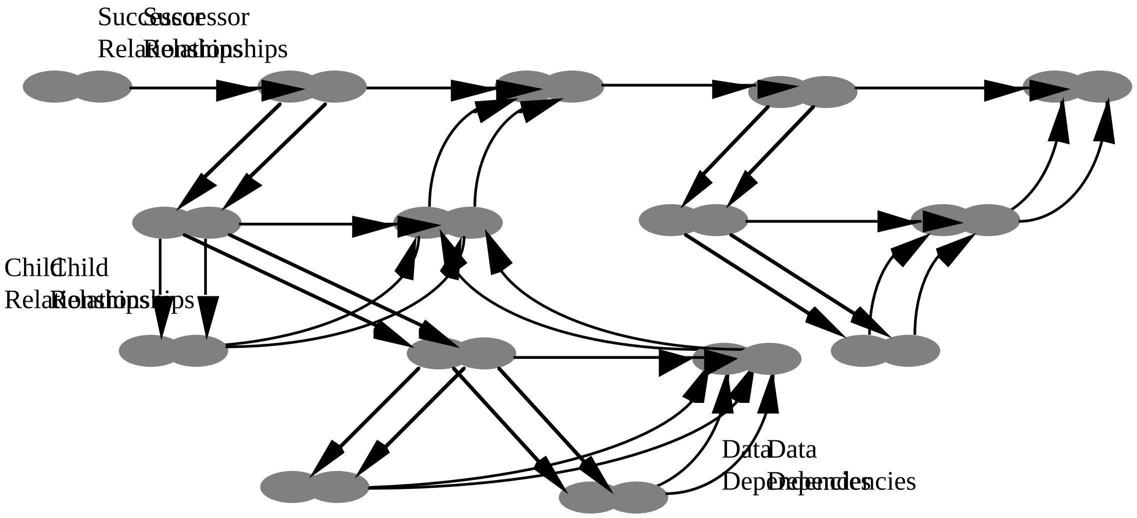


Figure 3: An example of a Cilk program.

sent by Thread 3 and Thread 4, respectively. Thread 5 waits for two results from the spawned activation frames (indicated by the synchronization counter value of 2). The two received values are added, and the result is sent to the parent by Thread 6. If the argument is less than 2, Thread 1 calculates the base value (=1), and thread 6 returns this value to the parent.

There are four Inlets, two to receive the frame pointers for the recursive calls, and two to receive results from the spawned frames. The synchronization counter of a thread is decremented when either a thread or an inlet "posts" to that thread.

2.2 Cilk

Cilk [Blumofe95] language is an extension of C, providing an abstraction of threads in explicit continuation passing[1] style. The Cilk run-time supports "work stealing" for scheduling threads and achieves load balancing across a distributed processing environment. A Cilk program consists of a collection of procedures, each in turn consists of threads. These threads of a Cilk program can be viewed as the nodes of a directed acyclic graph as shown in Figure 2. Each horizontal edge represents a creation of a successor thread, a downward vertical edge represents the creation of child threads while the curved upward edges represents data dependencies.

[1] More recent implementation of Cilk (e.g., Cilk 5) have deviated from Continuation Passing style, and chose shared memory for passing arguments.

Like TAM threads, Cilk threads are non-blocking. This requires the creation of successor threads which receive results form child threads. The successor thread is blocked until the necessary synchronization events (or release conditions) arrive. Cilk threads can spawn child threads to execute a new procedure. The child threads normally return values or synchronize with the successor threads created by their parent thread.

The run-time system keeps track of the active threads and threads awaiting initiation. The data structure used for thread management is called a "Closure". A closure consists of a pointer to the code of the thread, a slot for each of the input parameters for the thread, and a join counter indicating the number of missing values (or synchronization events). The closure (hence the thread) becomes ready to execute when the join counter becomes zero; otherwise the closure is known as waiting. The missing values are provided by other threads using "continuation passing" which identifies the thread closure and the argument position in the thread closure. The following shows a Cilk program segment for computing the Fibonacci numbers.

```
thread fib (cont int k, int n)
{
    if (n<2)
            send_argument (k, n)
    else{
            cont int x, y;
            spawn_next sum (k, ?x, ?y);  /* create a successor thread
            spawn fib (x, n-1);          /* fork a child thread
            spawn fib (y, n-2);          /* fork a child thread
            }
   thread sum (cont int k, int x, int y)
   send_argument (k, x+y);               /* return results to parent's successor
}
```

The program consists of two threads, `fib` and its successor `sum` (which waits for the recursive `fib` calls to complete and provide the necessary values to `sum`). The `fib` thread tests the input argument `n`, and if it is greater than 2, it spawns the successor thread `sum` by passing the continuation `k`, and the indication that sum requires two inputs `x` and `y` before becoming enabled. It also spawns two (recursive) child threads with `n-1` and `n-2` as their arguments, as well as the slot where they should send their results (specified by the `cont` parameter). The statement `send_argument` sends

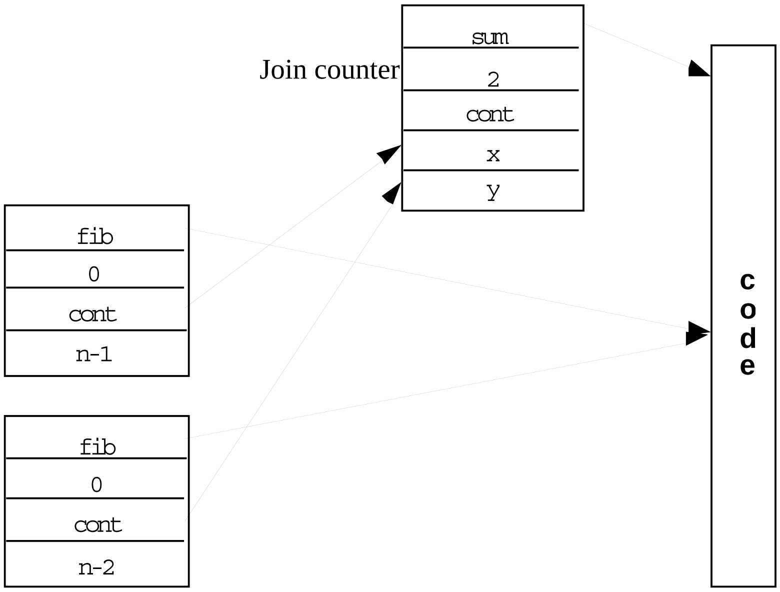


Figure 5: The closures for the Fibonacci program.

the results to the appropriate continuation. The closures for the above Fibonacci program is shown in Figure 3. The similarities between the Cilk run-time system and the continuation passing methods used in dynamic dataflow systems should be clear to the reader.

Cilk run-time system uses an innovative approach to load distribution known as "work stealing". In short, an idle worker randomly selects a heavily loaded processor, and steals a portion of its work. Note that only ready to execute threads are stolen, to avoid the complications that could result in locating the continuation slots of the stolen threads.

2.3 Cid

Unlike TAM and Cilk, Cid threads can block waiting for synchronization [Nikhil94]. Each Cid thread can be viewed as a C function with appropriate mechanism to specify synchronization. The simplest type synchronization is based on Join (and join variables). Consider the following Cid implementation of the Fibonacci function.

```
int fib(int n)
{   int fibN1, fibN2;
   cid_initialized_jvar(joinvariable);
           if (N<2) return n
           else
           {   cid_fork(joinvariable;) fibN1=fib(n-1); fibN2=fib(n-2);
               cid_jwait(&joinvariable);
                return fibN1+fibN2; } }
```

When the value of N is greater than 2, two new threads are forked using `cid_fork` to compute `fib(n-1)` and `fib(n-2)`. The `cid_fork` also indicates that these computations synchronize using join on the `joinvariable` specified. The parent thread will wait for the completion of the child threads and then returns the sum of `fib(n-1)` and `fib(n-2)` and signals appropriate `joinvariable`. Note that the Cid system is responsible for initializing the joinvariable (as indicated by `cid_initialized_jvar`).

As can be seen from the description of the various programming models shown above, concurrency using multithreading is becoming prevalent in modern programming languages. Traditional imperative languages support coarse-grained threads, where the thread synchronization is based on locks, rendezvous or monitors (protected object of ADA-95). Functional and data-driven languages often permit fine-grained and non-blocking threads, using continuation passing and synchronization counters. Thread libraries can be used with languages such as C and C++ to interleave different sections of the program, mimicking concurrency. We feel that the popularity of Java will only increase the interest in multithreading at programming level, and more programming languages will introduce constructs for the creation and management of multithreaded programs.

3. Execution Models

In this section we describe how the underlying system can support multiples threads. We will only concentrate on Operating System level or run-time support for threads. Section 4 will discuss architectural level support for multithreading. The notion of threads evolved from a need for an execution model that supports cooperating activities within a process. A thread can be viewed as an unit of execution that is active within a process, sharing certain resources such as files and address space with other threads in the process space. However, each thread is associated with its own execution status. This notion of threads or lightweight processes was originally supported in Mach [Boykin93]. The main advantage of such a threaded model is to permit programming applications

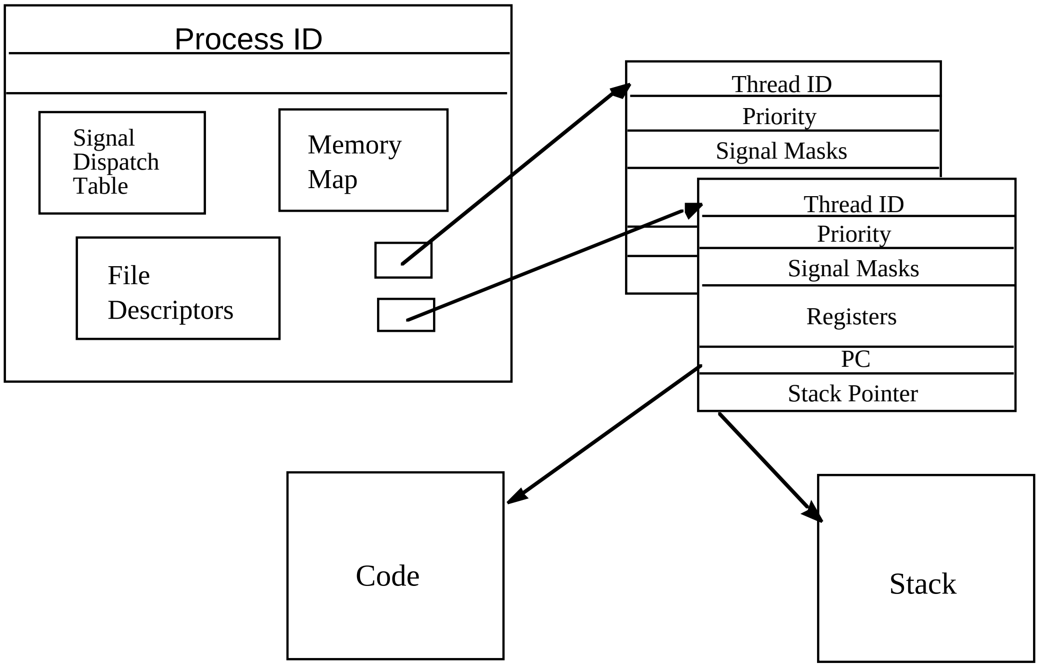


Figure 4. Processes and Threads

using "virtual processes" such that a process can continue execution even when one or more of its threads are blocked. Figure 4 illustrates the concept of threads as related to conventional Unix like processes.

Multithreaded programming model is becoming very common since most modern operating systems (including DEC Unix, Solaris, Windows, Windows-NT, Rhapsody) support threads. In addition, standardized user-level libraries are being provided by numerous vendors. Such packages permit users to create and manage threads. It should be noted that OS threads and user-level thread packages normally support coarse-grained threads that are blocking.

3.1 Design Issues

The execution models for multithreading can be distinguished from several view points: implementation (user-level vs. kernel-level), scheduling (preemptive vs. non-preemptive, binding of threads to processors and LWP's) and thread management functions (mutual exclusion, barriers, etc.). Threads can be implemented either at the kernel-level or user-level (see Figure 5). *User-level threads* ([Bershad90], [Eykholt92], [Freeley92], [Stein92]) are created and managed entirely at the user-level,

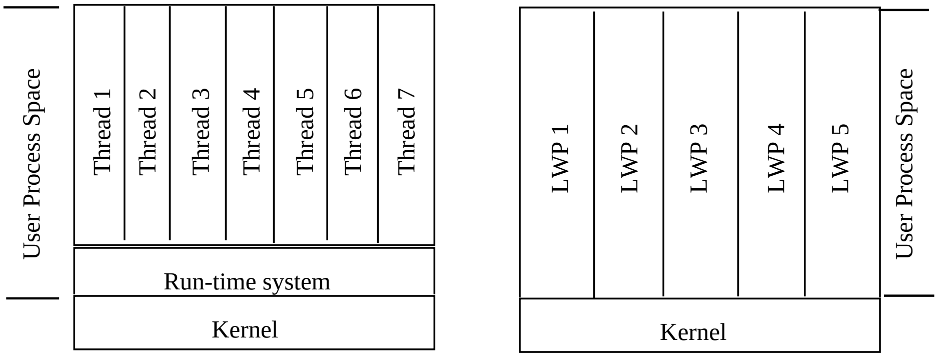


Figure 5. User Level and Kernel Level Threads

and the kernel has no knowledge of the existence of these threads. Such packages can be implemented on top of any operating system with or without kernel-level threads. The run-time system will intercept any calls made by user-level threads that could potentially block. The run-time system will not make the system call, but suspends the thread and schedules a new user thread. The required call is made if it results in no blocking or when there are no runnable user threads. The major advantage of such threads is efficiency in implementing thread functionality. It has been found that user-level thread management functions are as much as two orders magnitude faster than kernel-level thread management functions. This in turn permits each user-level process with a larger number of threads, leading to more user-level concurrency. However, since the kernel is unaware of the existence of such threads, when one user-level thread is blocked in the kernel, the entire process is blocked, thus nullifying the benefits of multithreading. Other disadvantages include a lack of control for preempting threads, or the ability to directly notify a thread of kernel events.

Kernel-level threads are essentially lightweight processes (LWP) which have the same address space as the “parent” process (see Figure 4). Hence it is less expensive to create threads than processes, and less expensive to switch between threads than between processes. However, kernel-level thread management functions are more expensive than those for user-level threads. Moreover, since each thread requires some kernel resources, the number of threads that can be supported within a process space is limited. These factors dictate that only coarse grained concurrency be used to exploit multithreading using kernel-level threads.

More recently, thread packages are becoming available that multiplex several user-level threads onto one or more kernel-level threads (or LWPs), resulting in *hybrid threads*. Each user process can have multiple LWP's, and the run-time system can bind user-level threads to these LWPs. In such systems, scheduling occurs at two levels. The multiplexing of use-level threads onto LWP's is under the control of the run-time system, while the scheduling of LWP onto physical processors is under the control of the kernel. The hybrid model was originally implemented in Scheduler Activations [Anderson91]. When a user-level thread (scheduled on a LWP) is blocked, the kernel notifies (***upcall***) the run-time system and provides sufficient information about the event that caused the block. The run-time system will then schedules another user-level thread (possibly on another LWP). When the blocking event is cleared, the kernel notifies the run-time system, which either schedules the blocked thread or starts a new thread.

Thread implementations can also be distinguished based on the scheduling control given to the user:

Non-preemptive scheduling. In such systems, a thread runs until it is blocked on a resource request or completes its execution, before releasing the processing resources. In some recent implementations, it is possible for a thread to voluntarily "yield" the processing resources. Such non-premptive scheduling is possible only for user-level thread packages, since the kernel cannot permit run-away threads that do not relinquish their resources. For well behaved programs, this model is very efficient since very little run-time scheduling is involved. Another advantage of this model is that it reduces the reliance on locks for synchronizing threads, since the running thread knows when it is giving up control of the processor. The reduced use of locks will reduce the overhead due to thread synchronization functions. The major drawback of this model is that, for some CPU-intensive applications, very little performance gains can be obtained using multithreading.

Preemptive scheduling. When threads can be preempted, we can consider various scheduling approaches to dynamically schedule runnable threads, including priority scheduling and time-sliced (round-robin) scheduling. The priority based scheduling can also be used with non-preemptive model, where the selection of a new thread to run occurs when the running thread blocks or yields. In most systems, the thread priority is fixed and assigned statically. When time-slicing is used, the running thread is preempted when its time-slice expires, and it awaits its turn in the round-robin queue. Kernel-level threads often permit pre-emption of running threads on interrupts or when a higher priority thread becomes runnable. In most systems, the kernel attempts to prevent the starvation of lower priority threads, by periodically increasing their priority.

Thread packages also differ in how user-level threads can be bound to processing resources. In many-to-one model, all user-level threads are bound to a single processing resource (or kernel-level LWP). This is the only model feasible when the kernel does not support threads. In one-to-one model, each user-level thread is bound to a different kernel-level LWP. Many-to-many model is the most flexible since it allows different number of user-level threads to be bound to each kernel LWP. Solaris systems support all of the above models, while DEC Unix 3.0 and WIN32 threads support one-to-one model.

In addition to the differences in the design decisions described above, thread implementations differ in the thread management and synchronization functions they provide. The following table summarizes the thread functions supported by Pthreads, WIN32, and Solaris threads. Java threads are included for completeness sake, even though Java threads are a language feature, and they are either supported using threads provided by the underlying run-time and/or kernel threads, or simulated with interleaved execution of threads.

The multithreaded model of execution is becoming popular with programmers since user-level thread packages and kernel threads are becoming readily available, along with debugging and analysis tool ([Catanzaro94], [Kleiman96]). A majority of the systems provide reasonable control on the creation and management of threads. They differ in the flexibility of synchronization primitives, control over a thread’s priority, stack size for a thread, and ability to share the kernel resources across multiple processing units. There are experimental systems currently being developed that permit even greater control over threads. Such systems will allow the microkernel functionality to be customized for a specific application, by specifying the actions to be performed in response any thread function. Such systems (e.g., SPIN [Bershad94], Exo Kernel [Engler95]) are beyond the scope of this paper.

Features	*Java*	*POSIX*	*SOLARIS*	*WIN32* Features
Java	POSIX	SOLARIS	WIN32	

Java

User-Kernel Space	K	N/A	K and U	K
Cancellation	No	Yes	No	No
Priority Scheduling	Yes	Yes	Yes	Yes
Priority Inversion	?	Yes	Yes	Yes
Mutex Attributes	No	Yes	Yes	No
Shared and Private Mutexs	Yes	Yes	Yes	No
Thread Attributes	No	Yes	Yes	No
Synchronization	Yes	Yes	Yes	Yes
Stack Size Control	No	Yes	Yes	Yes
Base Address Control	No	Yes	Yes	No
Detached Threads	Yes	Yes	Yes	No
Joinable Threads	Yes	Yes	Yes	No
Condition Variables	Yes	Yes	Yes	?
Semaphores	Yes	Yes	Yes	Yes
Thread ID Comparison	Yes	Yes	Yes	No
Call-Once Functions	Yes	Yes	Yes	No
Thread Suspension	Yes	No	Yes	Yes
Specify Concurrency	?	No	Yes	Yes
Reader / Writers Share Locking	Yes	No	Yes	No
Processor Specific Thread Allocation	No	No	No	Yes
Fork All Threads	Yes	No	Yes	No
Fork Calling Thread Only	Yes	Yes	Yes	No

K = Kernel-Level; U = User-Level

The following describes the various characteristics listed in the above table.

Base Address Control - Allows identification of where the thread will reside in physical memory.
Call-Once-Functions - An ability to limit execution of a particular function/rountine only once. Subsequent call will return without execution and error.
Cancellation - Killing threads from within the program.
Detached Thread - A flag not to join a thread(s) at creation time.
Fork All Threads - A flag, which forces all thread-creation calls to be, forks with shared memory.
Joinable Threads - The ability to merge threads into a single execution context.
Kernel Level Threads - Threads that are handled/scheduled by the kernel.
Mutex - Mutex Exclusion. A Mutex can lock specific section of memory using access flags.
Priority Inversion - As threads get I/O blocked, provides a re-prioritization of threads.
Priority Scheduling - Programmatically identifying the order, priority, or next threads to execute.
Processor Specific Thread Allocation - The ability to designate a specific thread to a specific processor. Useful for processors that handle special things like interrupts or exclusions.
Reader / Writer Locking - In Solaris, threads can have one writer and several readers at the same time.
Semaphores - A pair of functions that lock data sets, p() and v() (lock and unlock).
Shared / Private Mutexes - Having separate spaces for mutexes.
Specifying Concurrency - The ability to identify which threads will be multiprocessed.
Stack Size Control - The ability to limit, resize or check the thread's stack usage.
Synchronization - Ensures multiple threads coordinate their activities.
Thread - The smallest context of execution.
Thread Suspension - Temporarily halting execution of a thread.
User Space Threads - Threads that are handled/scheduled within a single task by special libraries.

4. Architectural Support for Multithreading

The previous sections discussed multithreading support from a purely software point-of-view. This section presents the hardware mechanisms used to support multithreading. Hardware support needed for multithreading varies depending on whether thread execution blocks on long latency operation (i.e., blocked scheme) or is interleaved on a cycle-by-cycle basis (interleaved scheme). Both schemes, however, require support for multiple hardware contexts (i.e., states) and context switching, their implementations differ.

In the blocked scheme, the simplest way to support multiple contexts is to provide a register file with each context. This will reduce the cost of context-switching, however, these register partitions are fixed and inflexible, making it difficult to utilize effectively when the number of registers required per thread varies dynamically. This problem can be alleviated by allowing the contexts to share a large register file, but will likely to increase the register file access time.

Once multiple threads exist in the processor, it must decide when to context-switch. A context-switch can occur when there is a cache miss. This will require additional logic to signal cache misses. A processor probably will not context-switch on a L1 cache miss since the latency to fetch the cache line from L2 cache is small. Whether to context-switch on a L2 cache will depend on the cost of context switching, the thread run-length, and the latency of a L2 cache miss. The context switching cost depends on how much support is provided in the hardware, while thread run-length depends on the miss rate. Latency of L2 cache misses depends on the organization of a node. A single processor node will have lower latency than a node in a SMP. Finally, context switching will be necessary for misses on local memory that require requests to be sent to remote node.

Once the need to context-switch is detected, a number of possibilities exist for scheduling the next available thread. A simple technique that can be used is to select the next thread using round-robin scheduling. This can be implemented by a bit vector with warp-around indicating which threads are ready to be scheduled. Having selected the next thread to schedule, a context-switch is performed by saving the PC of the first uncompleted instruction from the current thread, squashing all incomplete instructions from the pipeline, save the control/status registers from the current thread, switch the control to the register file for the new context, restore the control/status registers from the new thread, and start executing instructions from the PC of the new thread.

5. Example Multithreaded Systems

Multithreading is desired when the performance of a parallel machine suffers from the latencies involved in the communication and synchronization. Multithreaded architectures provide various software and hardware features in order to support multithreading, including lightweight synchronization, fast context switching mechanism, effective and intelligent management of threads, efficient communication mechanism, and shared-memory model for ease of programming. This section provides an overview of various multithreaded architectures and discusses some of the software and hardware features that represent the past and the current research efforts in the multithreading community. The architectures included in the discussion are Tera [Alverson90], MIT’s StarT project ([Ang92], [Chiou95], [Nikhil92]), Electrotechnical Lab’s EM-X ([Kodama95], [Sakane95]), MIT's Alewife [Agrawal95], M-Machine [Filo95], and Simultaneous Multithreading [Tullsen95, 96].

5.1. Tera MTA

Tera MTA (MultiThreaded Architecture) computer is a multistream MIMD system developed by Tera Computer Company [Alverson90] It is the only commercially available multithreaded architecture that will become available in 1997. The designers of the system tried to achieve the following three goals: (1) provide high-speed, highly-scalable architecture, (2) be applicable to a wide variety of problems, including numeric and non-numeric problems, and (3) ease the compiler implementation.

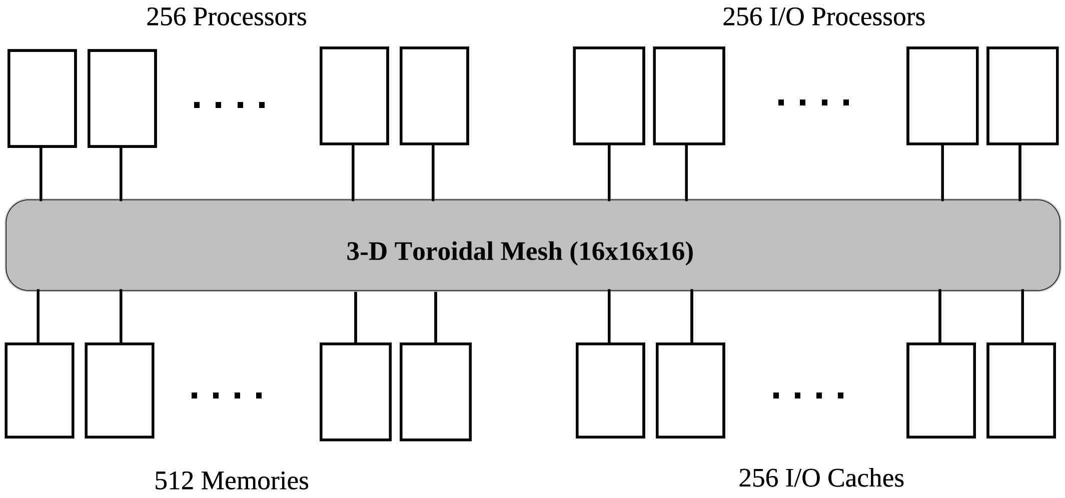


Figure 6: The organization of Tera MTA

The interconnection network of Tera is composed of pipelined packet-switching nodes in a three-dimensional mesh with a wrap-around. Each link is capable of transmitting a packet containing source and destination addresses, an operation, and 64-bit data in both directions simultaneously on every clock cycle. For example, a 256 processor system consists of 4096 switching nodes arranged in 16×16×16 toroidal mesh, among which 1280 nodes are attached to 256 processors, 512 data memory units, 256 I/O cache units, and 256 I/O processors as shown in Figure 6. In general, the number of network nodes grows as a function of $p^{\frac{3}{2}}$, where p is the number of processors in the system.

Each processor in Tera can simultaneously execute multiple instruction streams from one to as many as 128 active program counters. On every clock cycle, the processor logic selects an instruction stream that is ready to execute and a new instruction from a different stream may be issued in each cycle without interfering with the previous instruction. Each instruction stream maintains the following information: one 64-bit Stream Status Word (SSW), 32 64-bit General Purpose Registers (R0-R31), and eight 64-bit Target Registers (T0-T7). Thus, each processor maintains 128 SSWs, 4096 General Purpose Registers, and 1024 Target Registers, facilitating context switching on every clock cycle. Program addresses are 32 bits long, and the program counter is located in the lower half of the its SSW. The upper half is used to specify the various modes (e.g., floating-point rounding), trap mask, and four recently generated condition statuses. Target Registers are used for branch targets, and the computation of a branch address and the prediction of a branch are separated, allowing the prefetching of target instructions. A Tera instruction typically specifies three operations: a memory reference operation, an arithmetic operation, and a control operation. The control operation can also be another arithmetic operation. Thus, if the third operation specifies a arithmetic operation, it will perform a memory and two arithmetic operations per cycle.

Each processor needs to execute on the average about 70 instructions to maintain the peak performance by hiding remote latencies (i.e., the average latency for remote access is about 70 cycles). However, if each instruction stream can execute some of its instructions in parallel (e.g., two successive loads), less than 70 streams are required to achieve the peak performance. To reduce the required numbers of streams, Tera architecture introduced a new technique called *explicit-dependence lookahead* to utilize instruction-level parallelism. The idea is that each instruction contains a three-bit lookahead field that explicitly specifies how many instructions from this stream will be issued before encountering an instruction that depends on the current instruction. Since seven is the maximum possible lookahead value with three bits, at most eight instructions can be executed concurrently from each stream. Thus, in the best case only nine streams are needed to hide 72 clock cycles of latency, compared to 70 different streams required for the worst case.

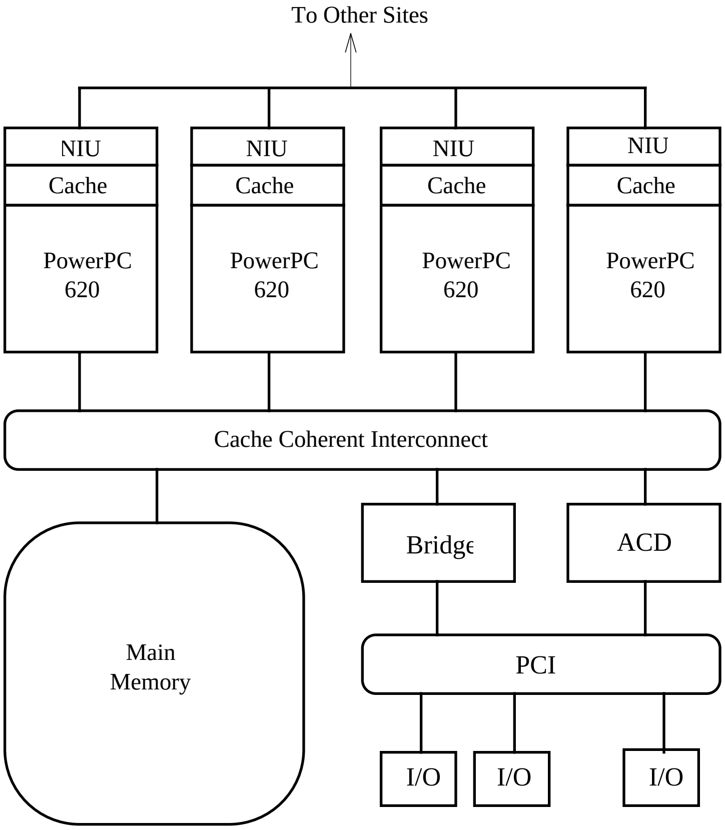
A full-size Tera system contains 512 128-Mbyte data memory units. Memory is 64-bit wide and byte-addressable. Associated with each word are four additional *access state bits* consisting of two data trap bits, a forward bit, and a full/empty bit. The trap bit allows application-specific use of data breakpoints, demand-driven evaluation, run-time exception handling, implementation of active memory objects, stack limit checking, etc. The forward bit implements invisible indirect addressing, where the value found in the location is interpreted as a pointer to the target of the memory reference rather than as the target itself. The full/empty bit is used for lightweight synchronization. Load and store operations use the full/empty bit to define three different synchronization modes along with the access control bits defined in the memory word. The values for access control for each operation is shown below.

value	LOAD	STORE
0	read regardless	write regardless and set full
1	not used	not used
2	wait until full and then read	wait until full and then write
3	read only when full and then set empty	write only when empty and then set full

For example, if the value of the access control field is 2, LOAD and STORE operations wait until the memory location is full (i.e., written) before proceeding. When a memory access fails, it is placed in a retry queue and memory unit retries the operation several times before the stream that issued the memory operation results in a trap. Retry requests are interleaved with new memory requests to avoid the saturation of the communication links with the requests that recently failed.

5.2 StarT

The StarT project attempts to develop general-purpose scalable parallel systems while using commodity components. StarT-NG (Next Generation) is the first effort in developing such a system [Ang92]. Based on a commercial PowerPC 620, a 64-bit, 4-way superscalar processor with a dedicated 128-bit wide L2 cache interface and a 128-bit wide L3 path to memory, StarT-NG is a SMP system that supports user-level messaging and globally-shared cache coherent memory.



StarT-NG has 4 processor card slots, where one to four slots are filled with Network-Endpoint-Subsystem (NES) cards. Each NES contains a single PowerPC 620 processor with 4 MBytes of L2 cache and a Network Interface Unit (NIU) as depicted in Figure 7. Each site has an Address Capture Device (ACD) on the NES board, which is responsible for bus transactions. When an access to global shared-memory is necessary, one of the processors is dedicated to servicing the ACD and is called a service processor (sP). On the other hand, when a processor is used for running application, the processor is called an application processor (aP).

StarT-NG is built on a *fat-tree* network using MIT's Arctic routers connected to NIU [Boughton94]. The NIU's packet buffers are memory-mapped into an application's address space enabling users to send and receive messages without kernel intervention. The arrival of a message can be signaled either by polling or interrupt. Generally, PowerPC 620 polls the NIU by reading a specified location of the packet buffer, resulting in lower overhead. An interrupt mechanism can also

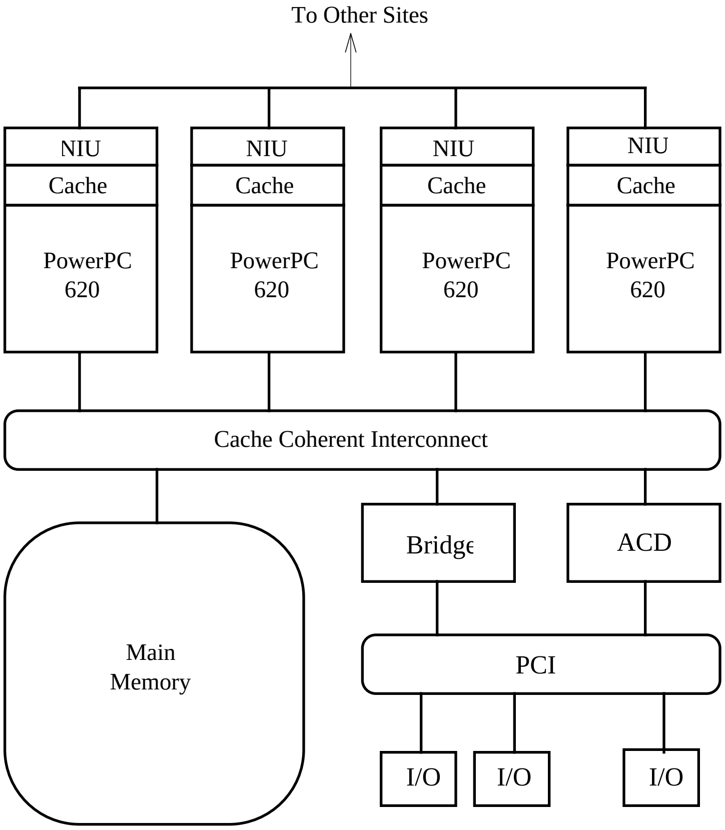


Figure 7: A site structure of StarT-NG

be used either for a kernel message or a user message when the frequency of the message arrival is estimated to be low, to minimize the overhead of polling. The dual ported buffer space of NIU is divided into four regions allowing receiving and transmitting of messages with both high and low-level priorities.

Cache-coherent distributed shared-memory in StarT-NG is implemented in software by programming the ACD and sP. This allows the designers of StarT-NG to experiment with various cache-coherence protocols, such as cache-only memory architecture.

Influenced by the predecessor *T [Nikhil92], multithreading in StarT-NG relies heavily on software support. The instruction `fork` creates a thread by pushing a continuation specified in registers onto a continuation stack. For thread switching, the compiler is required to generate `switch` (branch) instructions in the instruction stream. Also, the compiler needs to generate the necessary save/restore instructions to swap the relevant register values from the continuation stack, resulting in a large context-switching cost. StarT-NG examines how the multithreaded codes can run on a stock processor and emphasizes the importance of cache-coherent global shared-memory supported by efficient message-passing.

StarT-Voyager, which replaces StarT-NG, is based on dual-PowerPC 604 SMP system [Ang96]. Each SMP uses a typical PC/workstation class motherboard with two processors cards, but one of the processors cards is replaced with an NES card. Each NES card is then attached to the Arctic network to facilitate a scalable architecture. The NES has been programmed to support S-COMA coherent shared memory that allows local DRAM to act as a cache for global data. A two-node StarT-Jr system [Hoe96], consisting of Pentium Pro processors connected by a network interface attached to their PCI buses, was demonstrated at Fall Comdex95 in Las Vegas. StarT-Jr provides much of the same functionality of StarT-Voyager at a lower development cost and lower performance. A four-node StarT-Voyager system is expected to be completed in 1998.

5.3. EM-X

The EM-X parallel computer, which is a successor to EM-4 architecture [Sato92], is being built at Electrotechnical Laboratory in Japan ([Kodama95], [Sakane95]). EM-X architecture is based on the dataflow model that integrates the communication pipeline into the execution pipeline by using small and simple packets. Sending and receiving of packets do not interfere with the thread execution. Threads are invoked by the arrival of the packets from the network or by matching two packets. When a thread suspends, a packet on the input queue initiates the next thread. EM-X also supports direct matching for synchronization of threads, and the matching is performed prior to the buffering of the matching packets. Therefore, one clock cycle is needed for pre-matching of two packets, but the overhead is hidden by executing other threads simultaneously.

The EM-X consist of EMC-Y nodes interconnected by a circular Omega Network with virtual cut-through routing scheme. The structure of its single chip processor EMC-Y is depicted in Figure 8. The Switching Unit is a 3-by-3 crossbar connecting input and output of network and the

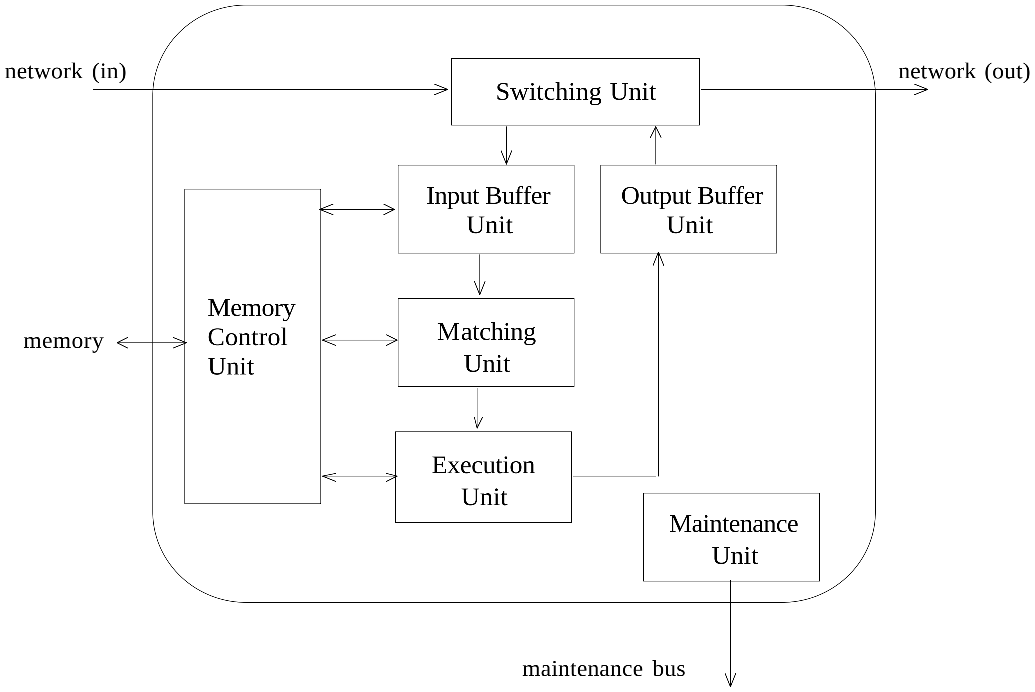
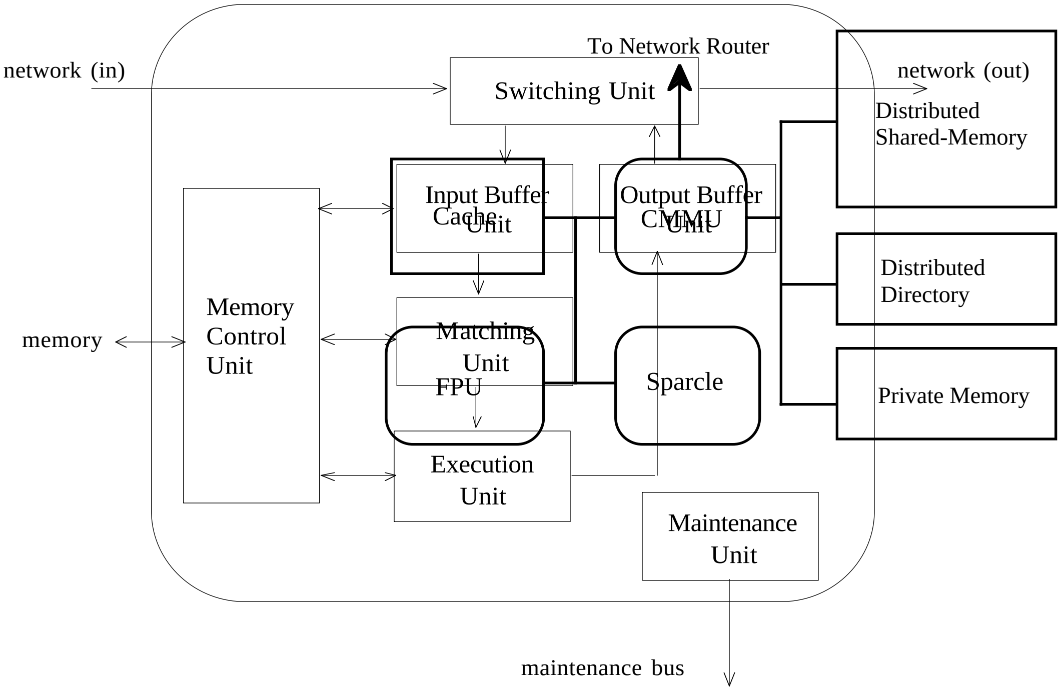


Figure 8: A structure of EMC-Y.

processor. Packets arriving at a processor are received in the Input Buffer Unit (IBU). The IBU has a on-chip packet buffer which holds a maximum of 8 packets. When the on-chip buffer overflows, packets are stored in data memory, and brought back to on-chip buffers when space becomes available.

EM-X implements a flexible packet scheduling by maintaining two separate priority buffers. Packets in the high priority buffer are transferred to the Matching Unit (MU), and the low priority packets are transferred to MU only when the high priority buffer is empty. The MU prepares the invocation of a thread by using the direct matching scheme [Kodama95]. This is done by first extracting the base address of the operand segment from the incoming packet. The operand segment is basically an activation frame which is shared among threads in a function, and holds the matching memory and local variables. Next, the partner data is loaded from the matching memory specified in the packet address, and the corresponding presence flag is cleared. Then, a template (i.e., a code frame) is fetched from the top of the operand segment, and the first instruction of the enabled thread



is executed on the Execution Unit (EXU). The EXU is a RISC-based thread execution unit with 32 registers. The EXU provides four SEND instructions for invoking a thread, accessing remote memory, returning the result after the thread execution, and implementing variable size operand segments or a block access of remote memory [Kodama95].

EM-X performs a remote memory access by invoking packet handlers at the destination processor, and the packets are entirely serviced by hardware which does not disrupt the thread execution in the execution pipeline. The round trip distances of the Omega Network in EM-X are 0, 5, 10, and 15 hops for request/reply sequences with the average of 10.13 hops requiring less than 1μsec on a unloaded network. On a loaded network, the latency is 2.5μsec on the average with random communication of 100 Mpackets/sec.

5.4. Alewife

MIT’s Alewife machine improves scalability and programmability of modern parallel systems by providing software-extended coherent cache, global memory space, integrated message-passing, and support for fine-grained computation. Underneath the Alewife's abstraction of globally shared memory, each PE has a physically distributed memory managed by a Communication and Memory Management Unit (CMMU). This memory hardware manages the locality by caching both private and shared data on each node. A scalable software-extended scheme called LimitLESS maintains the cache coherence [Chaiken91]. The LimitLESS scheme implements a full-map directory protocol which can support up to five read requests per memory line directly in hardware and by trapping into software for more widely-shared data.

Each Alewife node, shown in Figure 9, consists of a Sparcle processor, 64 Kbytes of direct-mapped cache, 4 Mbytes of data and 2 Mbytes of directory, 2 Mbytes of private unshared memory, a floating-point unit, and mesh routing chip. The nodes communicate via two-dimensional mesh network using wormhole routing technique.

Sparcle is a modified SPARC processor that facilitates block multithreading, fine-grained synchronization, and rapid messaging. The register windows of SPARC are modified to represent four independent contexts in Sparcle: one for trap handlers and other three for user threads. A context-switch is initiated when the CMMU detects a remote memory access and causes a synchronous memory fault to Sparcle. The context switching is implemented by a short trap handler that saves the old program counter and status register, switches to a new thread by restoring a new program counter and status register, then returns from the trap to begin execution in the new context.. Currently, the context-switching takes 14 clock cycles, but it is expected to be reduced to four clock cycles.

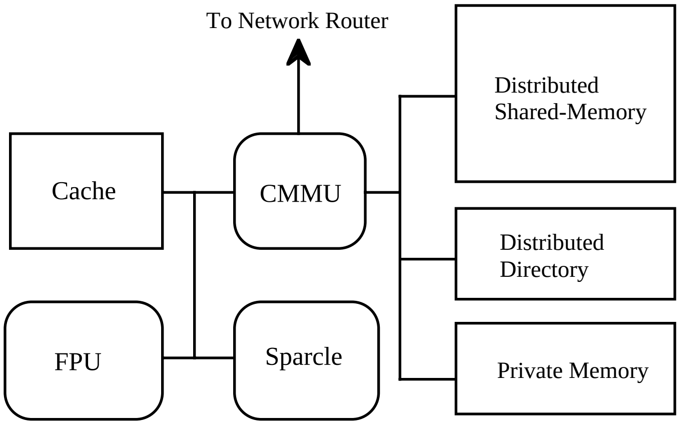


Figure 9: The Organization of Alewife Node

Sparcle also provides new instructions that manipulates the full/empty bits in memory for data-level synchronization [Agrawal93]. For example, `ldt` (read location if full, else trap) and `stt` (write location if empty, else trap) instructions can be used to synchronize on an element-by-element basis. When a trap occurs due to a synchronization failure, the trap handler software decides what must be done next.

Fast message handling is implemented via special instructions and memory-mapped interface to the interconnection network. To send messages, Sparcle first writes a message to the interconnection network queue using `stio` instruction, and then `ipillaunch` instruction is used to launch the message into the network. A message contains the message opcode, the destination node address, and data values (e.g., content of a register or address and length pair which invokes DMA on blocks from memory). The arrival of a message invokes a trap handler that loads the incoming message into registers using `ldio` instruction or initiate a DMA sequence to store the message into memory.

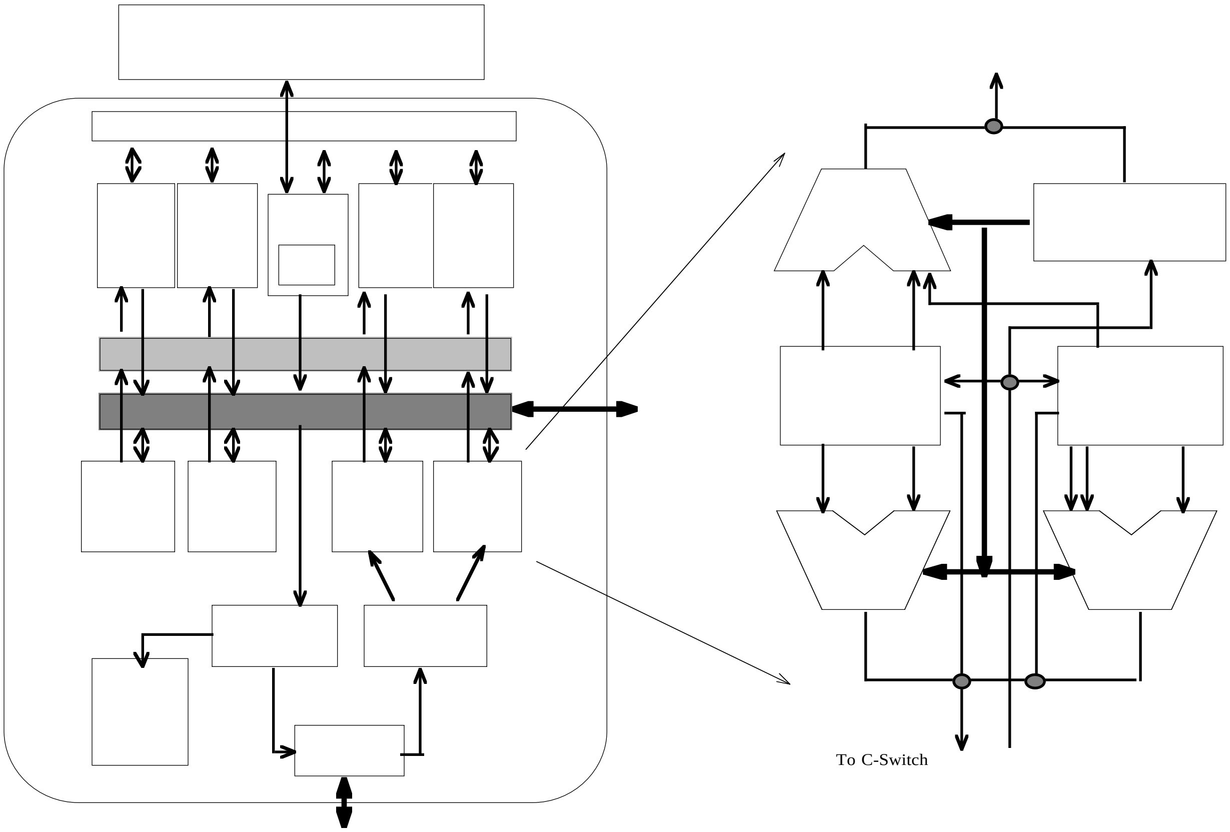
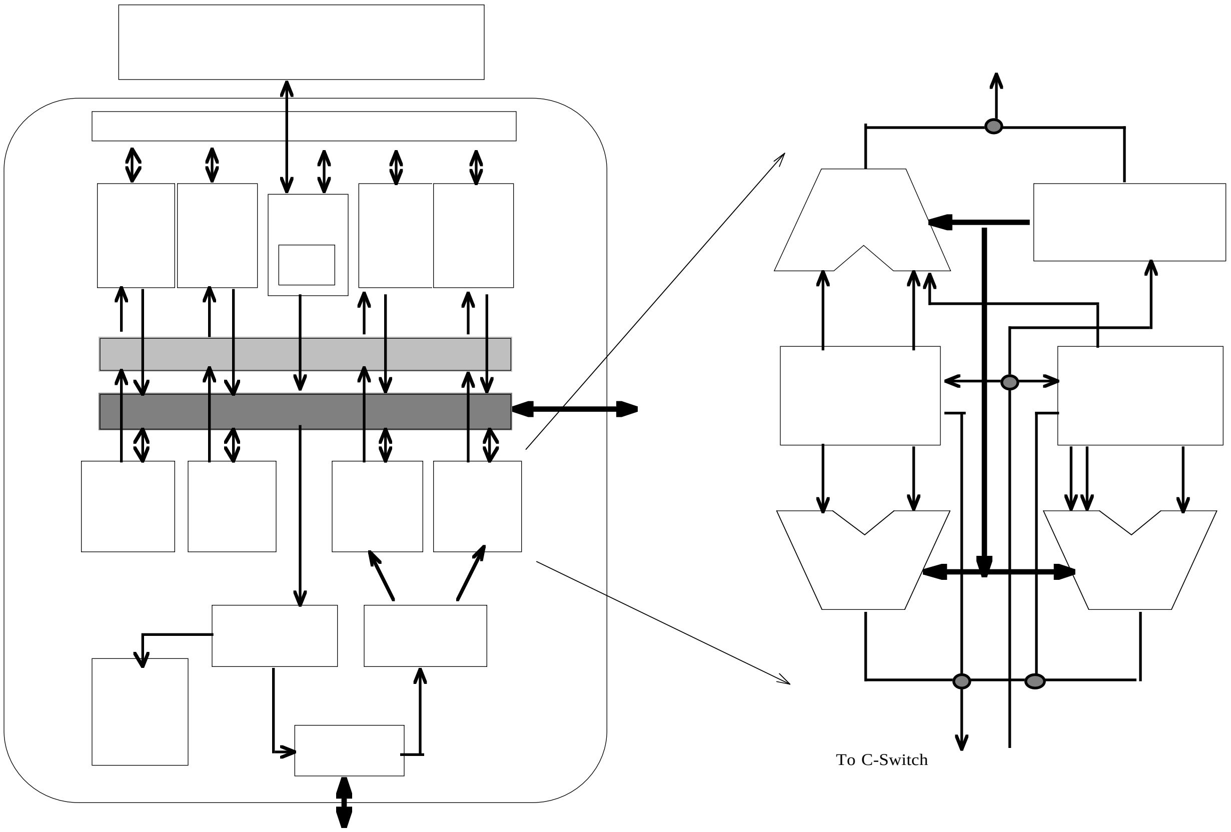


Figure 10: The MAP architecture and its four clusters.

5.5. M-Machine

M-machine is an experimental multicomputer being developed by MIT. The M-Machine efficiently exploits increased circuit density by devoting more chip area to the processor. It is claimed that a 32-node M-Machine system with 256 MBytes of memory has 128 times the peak performance of uniprocessor with the same memory capacity at 1.5 time the area, 85 times improvement in peak performance/area [Filo95]. The M-Machine consists of a collection of computing nodes interconnected by a bidirectional 3-D mesh network. Each node consists of a multi-ALU processor (MAP) and 8 MBytes of synchronous DRAM. A MAP contains four execution clusters, four cache banks, a network interface, and a router. Each of the four MAP clusters is a 64-bit, three-way issue, pipelined processor consisting of a Memory Unit, an Integer Unit, and a Floating-Point Unit as shown in Figure 10. The Memory Unit is used for interfacing to the memory and the



cluster switch (C-Switch). The cache is organized as four word-interleaved 32-KByte banks to permit four consecutive accesses. Each word has a synchronization bit which is manipulated by special load and store operations for atomic read-modify-write operations.

M-Machine supports a single global virtual address space through a global translation lookaside buffer (GTLB). GTLB is used to translate a virtual address into physical node identifier in the message. Messages are composed in the general registers of a cluster and launched automatically using user-level `send` instructions. Arriving messages are queued in a register-mapped FIFO, and a system-level message handler performs the requested operations specified in the message.

Each MAP instruction contains one to three operations and may execute out-of-order. The M-Machine exploits instruction-level parallelism by running up to 12 parallel instruction sequences (called H-Thread) concurrently. In addition, MAP interleaves the 12-wide instruction streams (called V-Thread) from different threads of computation to exploit thread-level parallelism and to mask various latencies that occur in the pipeline, (i.e., during memory accesses and communication). Six V-Threads are resident in a cluster, and each V-Thread consists of four H-Threads. Each V-Thread consists of a sequence of 3-wide instructions containing an integer, a memory, and a floating-point operation. Within an H-Thread, instructions are issued in order, but may complete out of order. Synchronization and communication among H-Threads in the same V-Thread is done using a scoreboard bit associated with each register. However, H-Threads in different V-Threads may only communicate and synchronize through memory and massages.

5.6. Simultaneous Multithreading

Simultaneous multithreading (SMT) is a technique that allows multiple independent threads from different programs to issue multiple instructions to a superscalar processor’s functional units. Therefore, SMT combines the multiple instruction-issue features of modern superscalar processors with the latency-hiding ability of multithreaded architectures, alleviating the problems of long latencies and limited per-thread parallelism. This means that the SMT model can be realized without extensive changes to a conventional superscalar processor architecture.

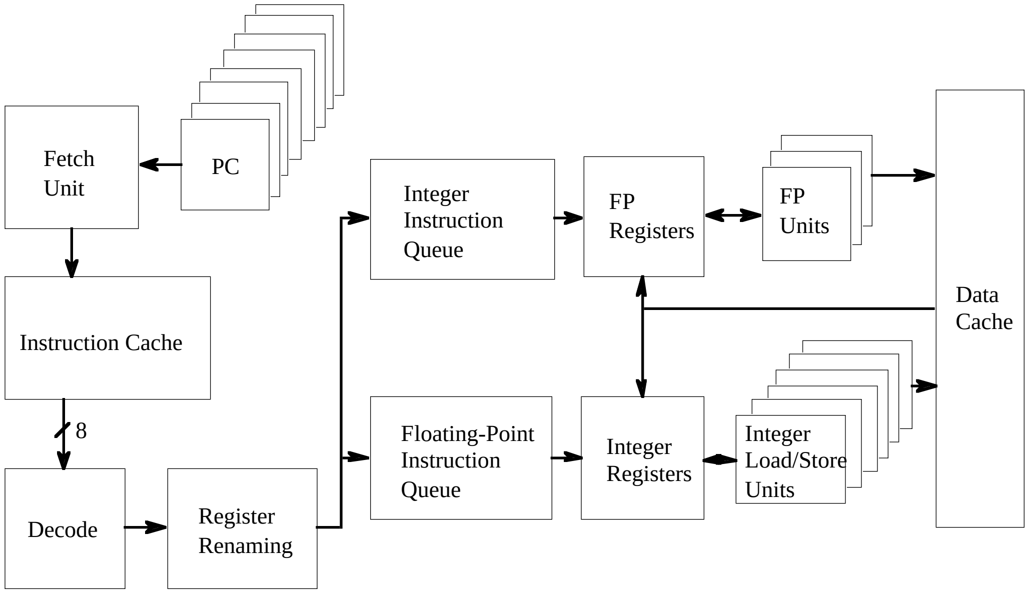


Figure 11: A Basic Simultaneous Multithreading Hardware Architecture

Figure 11 shows a hardware organization of an 8-thread simultaneous multithreading machine proposed in [Tullsen95, 96]. The processor execution stage is composed of three Floaing-Point Units and six Integer Units. Therefore, the peak instruction bandwidth is nine. However, throughput of the machine is bounded to eight instructions per cycle due to the bandwidth of Fetch and Decode Units. Each Integer and Floating-Point Instruction Queue (IQ) holds 32 entries, and the caches are multi-ported and interleaved. In addition, an 8-thread SMT machine has 256 physical registers (i.e., 32-registers per each thread) and 100 additional registers for register renaming.

The throughput of the basic SMT system is 2% less than a superscalar with similar hardware resources when running on a single thread due to the need for longer pipelines to accommodate a large register file. However, its estimated peak throughput with multiple threads is 84% higher than that of a superscalar processor. Also, the system throughput peaks at 4 instructions per cycle, even with eight threads. This early saturation is caused by the three factors: (1) small IQ size. (2) limited fetch throughput (only 4.2 useful instructions are fetched per cycle), and (3) lack of instruction-level parallelism. However, the performance of simultaneous multithreading hardware can be improved by modifying the Fetch Unit and Instruction Queues. The fetch throughput can be improved by optimizing fetch efficiency (i.e., partitioning fetch unit among threads), fetch effectiveness (i.e., selective instruction fetch or fetch policies), and fetch availability (i.e., eliminating conditions that block the fetch unit).

It has been shown that the best performance is obtained when the Fetch Unit is partitioned in such a way that eight instructions are fetched from two threads, and the priority is given to the threads with the smaller number of instructions in the decode stage [Tullsen 96]. Fetch misses can be reduced by examining the I-cache tag one cycle earlier, and then selecting only threads that cause no cache miss. However, this scheme requires extra ports on the I-cache tags and increases misfetch penalties due to an additional pipeline stage needed for early tag lookup. The resulting performance shows a factor of 2.5 throughput gain over a conventional superscalar architecture when running at 8 threads, yielding a 5.4 instructions per cycle. These experiments lead to following observations.

- Techniques such as dynamic scheduling and speculative execution in a superscalar processor are not sufficient to take full advantage of a wide-issue processor without simultaneous multithreading.
- Instruction scheduling in SMT is no more complex than that of a dynamically scheduled superscalar processor.
- Register file data paths in SMT are no more complex than those in a superscalar, and the performance implication on the register file and its longer pipeline is small.
- The required instruction fetch throughput is attainable without increasing the fetch bandwidth by partitioning the Fetch Unit and intelligent instruction selection to fetch.

6. Performance Models

Whether we deal with finely multithreaded or coarsely multithreaded architecture, there are limitations to the improvements in processor utilization that can be achieved. The most important limitation is applications running on a multithreaded system may not exhibit sufficiently large degrees of parallelism to permit the identification and scheduling of multiple threads on each processor. Even if sufficient parallelism exists, the cost of multithreading should be traded off against any loss of performance due to active threads sharing the cache and processor cycles wasted during context switches. In this section we will outline analytical models that can be used to describe these competing aspects of multithreaded systems.

In the simplest case, we assume that the processor switches between threads only on long latency operations, such as remote memory accesses. Let L denote a fixed latency for such operations. Let R be the average amount of time that each thread executes before encountering a long latency operation. Let C be the (fixed) overhead in switching between threads. Consider the case when there is only one thread. The processor utilization can be described by

$$U_1 = \frac{R}{R+L}. \qquad \text{(Eq. 1)}$$

The utilization is limited by the frequency of long latency operations, $\rho = 1/R$, and the average time required to service the long latency operation L.

If L is much larger than C, the time to switch between threads, then useful work can be performed during the latency operations. In addition, if the number of threads is sufficiently large, long latency operations can be completely hidden. In such a case, the processor utilization can be described as

$$U_{N_{SAT}} = \frac{R}{R+C}, \qquad \text{(Eq. 2)}$$

where N_{SAT} is the number of threads required to totally mask L. Note that increasing the number of threads beyond N_{SAT} will not increase the processor utilization. We will denote this as saturation number of threads which satisfies:

$$N_{SAT} \geq \frac{R+L}{R+C} \qquad \text{(Eq. 3)}$$

If there are insufficient number of threads to totally mask the latency L, the processor utilization can be described by

$$U_N = \frac{NR}{R+L} \qquad \text{(Eq. 4)}$$

Note that the overhead of switching among thread does not appear in the above equation since this time would have been idle (or wasted) in a single threaded system.

Using the above equations, the speedup that can be achieved is given by

$$S_N = \frac{U_N}{U_1} = \begin{Bmatrix} N & \text{if } N < N_{SAT} \\ \dfrac{R+L}{R+C} & \text{otherwise} \end{Bmatrix} \qquad \text{(Eq. 5)}$$

As shown in Eq. 5, the minimum number of threads needed to achieve maximum utilization, N_{SAT} $\geq (R+L)/(R+C)$, depends on time between thread switches (*R*), the time to service long latency operation (*L*), and the thread switching overhead (*C*). For example, a fine grained multithreaded system, *R*=1, with negligible thread switching overhead (e.g., using multiple hardware contexts) requires at least (1+*L*) threads to achieve optimum utilization. When *C* is not negligible, *R* should be much larger (i.e., coarser-grain multithreading) to achieve useful performance gains using multithreaded systems.

The above model ignored the performance impact due to higher cache miss rates in a multithreaded system and higher demands on the network placed by higher processor utilization. In addition, the above model assumed fixed latencies, and fixed frequency of long latency operations in threads. If we assume that a thread switch occurs on every cache miss, then we can equate cache miss rate *m* with the frequency of long latency operations, $\rho = 1/R$. Realistically, a thread switch occurs only on nonlocal cache misses. The speedup of a multithreaded system can be rewritten as

$$S_N = \frac{U_N}{U_1} = \begin{Bmatrix} N & \text{if } N < N_{SAT} \\ \dfrac{1+mL}{1+mC} & \text{otherwise} \end{Bmatrix} \qquad \text{(Eq. 6)}$$

The cache miss penalty is the primary contributor to *L*. Note that we assume constant cache miss rate and miss penalty in the above equation. The effect of thread switch on other long latency operations such as synchronization delays can also be added to above equation.

In deriving Eq. 6, we have assumed that cache miss rate and miss penalties are not effected by multithreading. However, cache miss rate is negatively effected by increasing the degree of multithreading. Likewise, the miss penalty increases with the number of threads due to higher network utilization (leading to longer delays in accessing remote memory modules).

Let us consider the impact of multiple threads on network delays (or miss penalties). The average rate of networks requests by a single thread is equal to the miss rate *m* = 1/*R*. As the number of threads is increased, the rate of requests is increased proportionately to *mN*, until *N* becomes equal from the network as $T = (\mu - \lambda_N)^{-1}$, where μ is the service time and λ_N is the rate of arrivals. Assuming Poisson distribution for the cache misses, we obtain $T = (\mu - mN)^{-1}$.

Note that the above derivation must be modified to account for the non-Poisson process that underlies cache misses. Network service time must reflect the topology, bandwidth, and routing algorithms of specific networks. For example, in [Agrawal 92], the miss penalties due to multiple threads assuming a *k*-ary *n*-dimensional cube network was computed. This analysis shows that network delays increase almost linearly with the number of threads, which is given as

$$T = \frac{T_0}{2} + \frac{BNk}{6} - \frac{1}{2m} + \frac{1}{2}\sqrt{\left(T_0 - \frac{BNk}{3} + \frac{1}{m}\right)^2 + 8NB^2 n \frac{k}{3}\left(1 - \frac{3}{k}\right)}, \qquad \text{(Eq. 7)}$$

where *M* represents the memory access time, *B* is the message size, *n* is the network dimension, *k* is the radix of the network radix, and T_0 represents the network delay without contention, i.e., $T_0 = 2nk_d + M + B - 1$, where k_d represents the average number of hops a message travels in each dimension.

In order to compute the impact of multiple threads on cache miss rates, let us review the behavior cache memories. It has been shown that the components of cache misses can be classified as *nonstationary, intrinsic-interference, multiprogramming-related,* and *coherency-related invalidations*. Nonstationary misses, m_{ns}, are due to “cold start” misses that bring blocks into the cache for the first time. Intrinsic-interference misses, m_{intr}, result from misses caused by conflicts among cache blocks of a working set that compete for the same cache set. Multiprogramming related misses account for the cases when one thread displaces the cache blocks of another thread. Coherency related invalidation, m_{inv}, occur in multiprocessor systems where the changes made in one processor may require invalidation of other processor cache entries.

Increasing the degree of multithreading will effect both the intrinsic-interference and multiprogramming components of cache misses. When more threads occupy the cache, we can assume that each thread is allocated a smaller working set, and this in turn leads to higher intrinsic conflicts. Likewise, as the number of threads is increased, the multiprogramming-related component also increases since there is a higher probability that cache blocks of active threads displace those of inactive threads. The miss rates in the presence of N threads is derived by Agrawal [Agrawal 92], which is given as.

$$m(N) \approx m_{fixed} + m_{intr} + m_{intr}(N-1)\left(1+\frac{1}{c}\right), \qquad \text{(Eq. 8)}$$

where c represents the collision rate and m_{intr} is a function of c, working set u, the time interval used to measure the working set τ, and the number of cache sets S, i.e.,

$$m_{intr} \approx \frac{c}{\tau}\frac{u^2}{S}$$

It is interesting to note that with sufficiently large cache memories, the multiprogramming related component of the cache miss rate is not effected by the number of threads. This is because, the cache memory is large enough to hold the working sets of all resident threads. The number of threads proportionately increases the intrinsic-interference component of cache misses. Set associativity is another issue that significant affects the performance of cache memories for multithreaded systems; higher associativities can compensate for the increased intrinsic interference in a multithreaded system. The collision rate parameter used in deriving cache miss rates by Agrawal [Agrawal92] must be described as a function of set associativity. Alternatively, set associativity can be modeled by treating the cache memory as several smaller direct mapped caches, each allocated to a different thread. This is the case when instructions from different threads are interleaved to achieve higher pipeline utilizations.

7. Conclusions and Prognostication

The past couple of decades have seen tremendous progress in the technology of computing devices, both in terms of functionality and performance. It is predicted that over the next five years, it will be possible to fabricate processors containing billions of transistor circuits operating at GigaHertz speeds [DARPA97]. While there has been a continuing growth in the density of DRAM memory chips, improvements in the access times and I/O bandwidth of memory parts have not kept pace with processor clock rates. This has widened the relative performance of processors and memory. The memory latency problem is further compounded by complex memory hierarchies which need to be traversed between processors and main memory. Multithreading is becoming increasingly popular as a technique for tolerating memory latency. It requires concurrency and complicated processors. However it offers the advantage of being able to exploit MIMD concurrency as well as interleave multiple users so as to maximize system throughput.

In this paper we have introduced the multithreaded paradigm as supported in programming languages, rutime systems, OS-kernels, and in processor architectures. We have also presented simple analytical models that can be used to investigate the limits of multithreaded systems. Without adequate hardware support, such as multiple hardware contexts, fast context-switch, non-blocking caches, out-of-order instruction issue and completion, register renaming, we will not be able to take full advantage of the multithreading model of computation. As the feature size of logic devices reduces, we feel that the silicon area can be put to better use by providing support for multithreading. The addition of more cache memory (or more levels of cache) will result in only insignificant and diminishing performance improvements. The addition of more pipelines (as in superscalar) will only prove effective with multithreading model of execution.

Hardware support alone is not sufficient to exploit the benefits of multithreading. We believe that the performance benefits of multithreading can only be realized when the paradigm is applied across all levels: from applications programming to hardware implementations. Fortunately, a number of research projects are underway for designing multithreaded systems that include new architectures, new programming languages, new compiling techniques, more efficient interprocessor communication, and customized microkernels.

New programming languages supporting both fine-grained and coarse-grained multithreaded concurrency are becoming available. Unless applications are programmed using these languages, the exploitable parallelism (in single threaded applications) will be very limited. New compile-time analysis and optimization approaches must be discovered to map user-level concurrency onto processor level threads. For example, it may be necessary to rethink register usages: it may be worthwhile loading multiple registers (belonging to different threads) with the same value, thus eliminating unnecessary data dependencies. It may be necessary to aggressively use speculative execution, and mixing instructions from unrelated threads to increase thread run-lengths.

While some of the research projects described in this paper have produced improvements over single threaded abstractions, in a majority of cases, they have shown only small or incremental improvements in performance. One of the issues often ignored by multithreaded systems is the performance degradation of single-threaded applications, due increased hardware data paths. Recently, numerous alternate approaches to tolerating memory latencies have been proposed, including DataScalar [Berger97], Multscalar [Sohi95], preload/prefetch techniques ([Baer91], [Dahlgren93], [Farkas95]. There has been a proposal for moving the processor onto DRAM chips, to reduce the latency [Saulsbury96]. It is our belief that multithreaded model of execution should be combined with some of these approaches proposed for sequential (single-threaded) execution systems. For example, the preloading can be adapted to multithreaded systems. The success of multithreading as a viable computational model depends on the integration of these efforts.

8. References

[Agrawal 93] Argawal, A. *et al*., "Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors," *IEEE Micro,* June 1993, pp 48-61.

[Agrawal 96] Argawal, A., *et al*., "The MIT Alewife Machine: Architecture and Performance," *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995, pp 2-13.

[Alverson 90] Alverson, R. *et al*., "The Tera Computer System," *International Conference on Supercomputing*, 1990, pp 1-6.

[Anderson 91] Anderson, T. E. *et. al.* "Scheduler activations: Effective kernel support for user level management of parallelism," *Proc of 13th Symposium on OS Principles*, pp 95-109, 1991.

[Ang 92] Ang, B. S., Arvind, and Chiou D., "StartT the Next Generation: Integrating Global Caches and Dataflow Architecture," *Proceedings of the 19th international Symposium on Computer Architecture*, 1992, Dataflow workshop.

[Ang96] Ang, B. S., Chiou, D, Rudolph, L., and Arvind, "Message Passing Support in StarT-Voyager," MIT Laboratory for Computer Science, CSG Memo 387, July 1996.

[Baer91] Baer, J-L. and Chen, T. "An effective on-chip preloading scheme to reduce data access penalty," *Proceedings of Supercomputing'91*, Nov. 1991, pp 178-186

[Berg 95] Berg, D.J. "Java Threads", A white paper from Sun Microsystems, 1995

[Berger97] Berger, D., Kaxiras, S., and Goodman, J., "Datascalar architecture," *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

[Bershad90] Bershad, B. *et. al.*, "Lightweight remote procedure call," *ACM Transactions on Computer Systems*, Feb. 1990, pp 37-55.

[Bershad94] Bershad, B. *et. al.*, “SPIN - An extensible microkernel for application-specific operating system services,” Tech Rept. 94-03-03, Dept of CS, University of Washington, Feb. 1994.

[Blumofe95] Blumofe, R. D. *et. al.*, “Cilk: An efficient multithreaded runtime system”, Proc of the *5th ACM Symposium on Principles and Practice of Parallel Programming (PPoP)*, July 1995.

[Boughton94] Boughton, G. A., “Arctic Routing Chip,” *Proceedings of the 1st International Workshop, PCRCW*, Vol. 853 of Lecture Notes in Computer Science, Springer-Verlag, 1994, pp 310 - 317.

[Boykin93] Boykin *et. al.*, “*Programming under Mach*,” Addison-Wesley Publishers, 1993.

[Catanzaro94] Catanzaro, B., “Multiprocessor System Architecture,” Prentice Hall, 1994.

[Chaiken91] Chaiken, D. *et al.*, “LimitLESS Directories: A Scalable Cache Coherence Scheme,” *The 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 224-234.

[Chiou95] Chiou D., *et al.*, “StatT-NG: Delivering Seamless Parallel Computing,” *Proceedings of EURO-PAR*, 1995, Stockholm, Sweden.

[Culler93] Culler, D. E, Goldstein, S. C., Schauser K. E. and von Eicken, T., “TAM- A compiler controlled threaded abstract machine,” *Journal of Parallel and Distributed Computing*, 18, pp 347-370.

[Dahlgren93] Dahlgren, F., Dubois, M., and Stenstrom, P., “Fixed and adaptive sequential prefetching in shared memory uniprocessors,” *Proceedings of the International Conference on Parallel Processing*, 1993.

[DARPA97] DARPA.“Multithreaded and Other Experimental Computer Architectures,” BAA 97-03.

[Engler95] Engler, D. R., Kaashoek, F. M., and O’Toole, J., “Exokernel: An operating system architecture for application-level resource management,” *Proceedings of the 15th Symposium on Operating Systems Principles*, Dec. 1995.

[Eykholt92] Eykholt, J. R. *et. al.*, “Beyond multiprocessing - Multithreading the Sun OS kernel,” *1992 Summer USENIX Conference Proceedings*, San Antonio,TX, June 1992.

[Farkas95] Farkas, K., Jouppi, N., and Chow, P., “How Useful are Non-blocking Loads, Stream Buffers and Speculative Execution in Multiple issue processors,” *Proceedins of the First HPCA*, Jan. 1995.

[Filo95] Fillo, M. *et al.*, “The M-Machine Multicomputer,” *Proceedings of MICRO-28*, 1995 (Also available as MIT AI Lab Memo 1532).

[Freely93] Feeley, M. J. *et. al.*,“User level threads and interprocess communication,” *Tech Rept. 92-02-03*, Dept of CS, University of Washington, 1993.

[Halstead85] Halstead, R., “Multilisp: A language for concurrent symbolic computation,” *ACM Transactions on Prog Lang and Syst,* Oct. 1985, pp 501-538.

[Heinlein94] Heinlein J. *et al.*, “Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor,” *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct., 1994.

[Hoe96] Hoe, J. C. and Ehrlich, M., “StarT-JR: A Parallel System from Commodity Technology,” *Proceedings of the 7th Transputer/Occam International Conference*, Tokyo, Japan, November 1996.

[ISO 94] International Organization for Stands. “Annotated Ada Reference Manual: Version 6.0”, *ISO/IEC 8652:1995* (E), Dec. 1994.

[Kleiman96] Kleiman, S., Shah, D., and Smaalders, B., “Programming with Threads,” Prentice Hall, 1996.

[Kodama95] Kodama, Y., Sakane, H., Sato, M., Yamana, H., Sakai, S., and Yamaguchi, Y., “The EM-X Parallel Computer: Architecture and Basic Performance,” *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995, pp. 14-23

[Kuskin 94] Kuskin J. *et al*., “The Stanford FLASH Multiprocessor,” *Proceedings of the 21st International Symposium on Computer Architecture*, 1994, pp 302-313.

[Lee94] Lee, B. and Hurson, A. R., “Dataflow Architectures and Multithreading,” *IEEE Computer*, August, 1994, pp. 27-39.

[Nikhil91] Nikhil, R., “Id (Version 90.1) Reference Manual,” *Tech Rept., CSG Memo 284-2*, MIT Laboratory for Computer Science.

[Nikhil92] Nikhil, R. S., Papadopoulos, G. M., and Arvind, “*T: A Multithreaded Massively Parallel Architecture,” *Proceedings of the 19th International Symposium on Computer Architecture*, 1992.

[Nikhil94] Nikhil, R. S.,“Cid: A parallel shared-memory C for distributed memory machines”, *Proc of 7th International Workshop on Languages and Compilers for Parallel Computing*, Ithaca, Aug. 1994 Springer-Verlog, pp 377-390

[Pier83] Pier, K. A., “A retrospective on the Dorado. A high performance personal computer,” *Proceedings of the 10th International Symposium on Computer Architecture*, pp 252-269, 1983.

[Sakane95] Sakane, H., Sato, M., Kodama, Y., Yamana, H., Sakai, S., and Yamaguchi, Y., “Dynamic Characteristics of Multithreaded Execution in the EM-X Multiprocessor,” *Proceedings of 1995 International Workshop on Computer Performance Measurement and Analysis(PERMEAN '95)*, Beppu Ohita JAPAN, pp. 14-22

[Sato92] Sato, M. *et al*., “Thread-Based Programming for EM4 Hybrid Dataflow Machine,” *Proceedings of the 19th International Symposium on Computer Architecture*, 1992.

[Saulsbury96] Saulsbury, A., Pong, F., and Nowatzyk, A., “Missing the memory wall: A case for processor/memory integration,” *Proceedings of the 23rd International Symposium on Computer Architecture,* May 1996, pp 90-101.

[Smith85] Smith, B., “The architecture of HEP” in *Parallel MIMD Computation: HEP Supercomputer and applications*, edited by J. S. Kowalik, MIT Press 1985.

[Sohi95] Sohi, G, Breach, S, and Vijaykumar, T., “Multiscalars processors”, *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995, pp 414-424.

[Stein92] Stein, D and Shaw, D. “Implementing lightweight threads”, *Proceedings of the 1992 Summer USENIX Conference*, San Antonio,TX, June 1992.

[Thekkath94a] Thekkath, R., and Eggers, S. J., “The Effectiveness of Multiple Hardware Contexts,” *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994, pp 328-337.

[Thekkath94b] Thekkath, R., and Eggers, S. J., “Impact of Sharing-Based Thread Placement on Multithreaded Architectures,” *Proceedings of the 21st International Symposium on Computer Architecture*, 1994.

[Theobald93] Theobald, K. B., “Panel Session of the 1991 Workshop on Multithreaded Computers,” ACAPS Technical Memo 30, April 1, 1993. McGill University.

[Tullsen95] Tullsen, D. M., *et al.*, “Simultaneous Multithreading: Maximizing On-Chip Parallelism,” *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995, pp 392-403.

[Tullsen96] Tullsen, D. M., *et al.*, “Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor,” *Proceedings of the 23rd International Symposium on Computer Architecture*, 1996.