Tera Hardware-Software Cooperation*

Gail Alverson Preston Briggs Susan Coatney Simon Kahan
Richard Korry

Tera Computer Company

Abstract

The development of Tera's MTA system was unusual. It respected the need for fast hardware and large shared memory, facilitating execution of the most demanding parallel application programs. But at the same time, it met the need for a clean machine model enabling calculated compiler optimizations and easy programming; and the need for novel architectural features necessary to support fast parallel system software. From its inception, system and application needs have molded the MTA architecture. The result is a system that offers high performance and ease of programming by virtue not only of fast physical hardware and flat shared memory, but also of the streamlined software systems that well utilize the features of the architecture intended to support them.

1 Introduction

The architectural features of Tera's MTA system have been selected to facilitate development and execution of parallel software. That application software benefits from these features is evident from the simple programming model and the high performance attainable without rewriting code or attending to data layout. But the features were chosen not only to ensure that the end product would have broad applicability; they were also chosen to enable efficient compilation and fast, parallel system software. This paper focuses on ways that specific features of the MTA hardware architecture [2] cooperate with Tera's system software: compiler, runtime, operating system, trap handlers, and debugger. These features include:

Shared memory Flat, uniform-access shared memory frees the programmer from all considerations of data placement. Every location is uniformly accessible by every processor. There are no data caches and no stride considerations.

*This research was supported in part by the United States Defense Advanced Research Projects Agency Information Science and Technology Office ARPA Order No. 6512/2-4; Program Code No. OT10 issued by DARPA/CMO under Contract MDA972-90-C-0075. The views and conclusions contained in this document are those of Tera Computer Company and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U. S. Government.

High bandwidth The high-bandwidth network precludes unexpected memory-traffic bottlenecks. Every processor can issue a memory reference at every clock cycle without risk of network or memory congestion.

Multithreading Multithreading and lookahead provide latency tolerance, relieving the pressure to exploit locality. At every clock cycle, a processor issues one instruction from any one of many ready threads. Thus, when one thread requests a value from memory and is thereby delayed, instructions from other threads may execute, utilizing the processor during the interim. Moreover, each memory operation has an associated lookahead number that tells the processor how many more instructions it may execute from the same thread prior to the memory operation completing, thus allowing for even greater latency tolerance.

Stream management A single user-level operation issued by one stream can reserve a number of additional streams. Once reserved, each of these streams can be activated by execution of a second user-level instruction. When a stream completes its work, it can quit by issuing a third user-level instruction. This lightweight stream-management mechanism allows for fast adaptivity to fluctuations in parallelism and the ability to parallelize even very small quantities of work without being swamped by overheads.

Memory state bits The MTA provides four state bits associated with each memory word: a forwarding bit, a full-empty bit, and two data-trap bits. Two of these, the full-empty bit and one of the data-trap bits, are used for lightweight synchronization. Access to a word is controlled by the pointer or instruction used to access the memory location and the values of the tag bits. Regardless of the state of the full-empty bit, if either of the data-trap bits is set and not disabled by the pointer, the memory access fails and the issuing stream traps.

The MTA supports three modes of interaction with the full-empty bit:

Normal Read and write the memory word regardless of the state of the full-empty bit. Writes set the bit to full.

Future Read and write the word only when the bit is full. Leave the bit full.

Synchronized Read only when the bit is full and set the bit to empty. Write only when the bit is empty and set the bit to full.

When a memory access must wait, it is retried several times in the hardware before the stream that issued the operation traps. The retries are done in the memory functional unit and do not interfere with other streams issuing instructions.

Wide instructions Every instruction consists of up to three operations that can include one memory operation and a pair of arithmetic operations. This feature, along with memory lookaheads, provide a mechanism for the compiler to express instruction-level parallelism.

Protection domains Every processor supports up to 16 protection domains, each potentially providing a separate address space to a distinct program. Thus, many programs may execute concurrently on the same processor, their instructions intermixed in the pipelines, allowing multiprogramming to help sustain processor utilization.

2 Compiler

Peak performance of a computer system is irrelevant; what is important is the sustained performance achievable with reasonable effort. If it is too difficult to program a particular machine, then the cost of sustaining high performance will overwhelm the hardware cost of that machine. The best way to reduce the cost of achieving high performance is to provide an effective compilation system. Tera provides three compilers, for C, C++, and Fortran 77. While the front ends differ, they all share the same optimization and code-generation phases. We refer to these parts collectively as "the compiler."

2.1 What is Required for Fast Code

The architecture of the Tera MTA and the design of the compiler drive each other. Naturally, the compiler must attempt to take full advantage of the machine's architecture; but it is important to remember that the machine was designed with **current** compiler technology in mind. In other words, the architecture was intended to be a fairly straightforward compiler target, not to serve as a justification for compiler research. The compiler must do two things to generate fast code for the system:

Exploit parallelism The compiler's primary task is to exploit loop-level parallelism by running the iterations of each parallel loop with multiple hardware threads. As a secondary task, the compiler uses software pipelining [9] to exploit instruction-level parallelism.

Reduce work Given parallelism, the other important task is to reduce the total work (the number of instructions executed) via a combination of traditional optimizations and a few, more recent optimizations based on dependence analysis and loop transformations (e.g., loop fusion, loop interchange, and outer loop unrolling [5]).

From the perspective of the compiler writer, it is equally interesting to consider what is **not** required for fast code.

Data distribution Since the MTA provides a shared, uniformly accessible memory, data distribution or placement is not a performance issue.

Cache management Because there are no data caches, it is unnecessary to block, prefetch, worry about access stride, or lack of set associativity.

Branch prediction The MTA's general latency tolerance extends to cover branch latency, obviating the need for various forms of branch prediction.

Structural hazards The pipeline is completely hidden; each instruction requires exactly 1 issue slot and the results are always available for the next instruction. Thus, there are no structural hazards to complicate software pipelining or instruction scheduling in general.

Comparing the two lists, we see that the MTA requires compilers with well-understood optimization and parallelization technology (of the sort discussed in Wolfe's textbook [10]) versus the relatively experimental techniques currently being explored for data distribution and cache management.

2.2 An Example

The NAS Integer Sort benchmark [3] fills a vector with random 19-bit integers, then computes a **rank** for each number – the position each entry would have if the vector were sorted. Since the range of the numbers is small, it is feasible to use a simple counting sort, similar to a radix sort, to determine the ranks. The algorithm can be expressed in C as follows:

```
for (i = 0; i < key_value_bound; i++)
  count[i] = 0;

for (i = 0; i < nkeys; i++)
  count[key[i]]++;

start[0] = 0;
for (i = 1; i < key_value_bound; i++)
  start[i] = start[i - 1] + count[i - 1];

for (i = 0; i < nkeys; i++)
  rank[i] = start[key[i]]++;
```

The idea is to first accumulate, in `count[n]`, the number of keys with value `n` (accomplished by the first and second loops). Next, we compute a starting location, `start[n]`, for each possible key value `n`. So all the 0's will come first (starting at location 0), then all the 1's, then the 2's, and so forth. Finally, we compute the rank of each key value, so that any key equal to 0 will end up with a rank between 0 and `start[1] - 1` or, more generally, any key equal to `n` will receive a rank between `start[n]` and `start[n + 1] - 1` inclusive.

For example, assume we need to compute the rank of 10 elements in the range of 0 to 7. The initial value of the vector `key` and the values for `count`, `start`, and `rank` computed in the second, third, and fourth loops, respectively, would be:

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
key[0..9]	1	2	4	5	3	3	6	7	4	5
count[0..7]	0	1	1	2	2	2	1	1		
start[0..7]	0	0	1	2	4	6	8	9		
rank[0..9]	0	1	4	7	2	3	8	9	5	6

A parallel version of the code for the MTA looks like:

```
for (i = 0; i < key_value_bound; i++)
  count[i] = 0;

for (i = 0; i < nkeys; i++)
  count[key[i]]++;

start[0] = 0;
for (i = 1; i < key_value_bound; i++)
  start[i] = start[i - 1] + count[i - 1];

start$ = (sync int *) start;
```

```
#pragma tera assert parallel
for (i = 0; i < nkeys; i++)
  rank[i] = start$[key[i]]++;
```

Parallel code is produced for each loop by the compiler. The compiler generates code to acquire multiple threads and distribute iterations of the loops to those threads. Barriers are inserted by the compiler to separate the four main loops. The barriers are implemented as binary trees, where pairs of threads use full-empty bit locking.

The first loop is trivially parallel, and the compiler will recognize this and generate code requiring, asymptotically, 1 instruction per iteration. On a single processor, the loop iterations will be distributed across many threads, with each thread executing a single instruction per iteration. On P processors the iterations will be distributed to threads on all the processors, and the computation will progress at a rate of P iterations per clock tick.

The second loop is also parallelized automatically, with the compiler producing code requiring 2 instructions per iteration. This loop is interesting in that it is not trivially parallel. There is a dependence carried by the increment of `count[]` that would prevent parallelization on many machines. For the MTA, we are able to use a special fetch-and-add instruction that can **atomically** update locations in memory. Without the atomic update, we might see a race between two processors loading, say, `count[5]`, adding 1 to the value, and storing it back to memory.

The third loop is a linear recurrence. The compiler automatically recognizes the recurrence and generates parallel code to solve the recurrence using a technique called cyclic reduction [4]. The generated code requires less than 4 instructions per iteration (actually 3 plus a small logarithmic term).

Notice that we have had to assert that the last loop is parallel. The compiler will not automatically parallelize this loop without the assertion because it always preserves the semantics of the original code. In this case, the original code (without the assertion) guarantees that the first key of a particular value (say 0) will be ranked before some later key of the same value. But we, the users, don't really care about that; it's not part of the original problem specification. So the assertion allows the compiler to generate code that is more flexible – all of the 0's have to be ranked before all of the 1's, but we don't care about the order of the 0's or the order of the 1's.

Note also the cast of `start` to the the special pointer `start$`. The variable `start$` is declared as a pointer to `sync int`, meaning that each integer in the vector is treated by the compiler as a **sync** variable. When the compiler sees an update to a sync variable, namely

```
start$[key[i]]++;
```

it makes sure the update is carried out **atomically**, so that no other iteration can possibly cause a race condition. As in the second loop, this atomic update is performed using a fetch-and-add operation. In the second loop, the compiler detected the need for atomicity automatically, without use of the sync variable. In the fourth loop however, the sync variable is required since the assertion would otherwise instruct the compiler to parallelize the loop without any special consideration for atomic updates.

The generated code for this final loop requires 3 instructions per iteration. Therefore, we expect to spend about 5 instructions per key and 5 instructions per key-value for the entire kernel. We were able to run the benchmark code (ranking 2^{23} integers in the range 0–2^{19}, repeated 10 times) on our prototype in 1.53 seconds. This is the best result ever achieved by a single processor for the Class A problem (the previous best was 2.02 seconds on one processor of a Cray T916).

These work counts (e.g., 1 instruction per iteration on the first loop) show that the compiler is generating perfect code for these loops. For example, consider the second loop

```
for (i = 0; i < nkeys; i++)
  count[key[i]]++;
```

The compiler generates code requiring 2 instructions per iteration. Since an MTA processor can only issue 1 memory reference per instruction, and the loop requires at least a load and a fetch-and-add, 2 instructions is clearly minimal. A disassembled version of the loop core is shown here:

```
(inst 1 (INT_MEM_ADD_INDEX r19 r8 r14) (INT_ADD_IMM r21 r21 8))
(inst 7 (LOAD r14 r21) (NOP) (NOP))
(inst 1 (INT_MEM_ADD_INDEX r19 r8 r20) (INT_ADD_IMM r21 r21 8))
(inst 7 (LOAD r20 r21) (NOP) (NOP))
(inst 1 (INT_MEM_ADD_INDEX r19 r8 r11) (INT_ADD_IMM r21 r21 8))
(inst 7 (LOAD r11 r21) (NOP) (INT_ADD r6 r21 r4))
(inst 1 (INT_MEM_ADD_INDEX r19 r8 r10) (INT_ADD_IMM r21 r21 8))
(inst 7 (LOAD r10 r21) (NOP) (INT_SUB_TEST r0 r6 r7))
(inst 1 (INT_MEM_ADD_INDEX r19 r8 r22) (INT_ADD_IMM r21 r21 8))
(inst 7 (LOAD r22 r21) (NOP) (JUMP_OFTEN IF_ILT cn0 t3))
```

Here, each line represents a single instruction with a lookahead field and 2 or 3 operations. The jump in the last instruction is back to the first instruction, with the destination specified via a target register `t3`. The loop has been unrolled several times during software pipelining, so each iteration of this code (10 instructions) will accomplish 5 iterations of the original loop.

The lookahead fields provide a way for the compiler to explicitly specify dependences, allowing the processor to have multiple outstanding memory references for each thread. A single iteration has a `LOAD` followed by an `INT_MEM_ADD`. Examining the load in the second instruction, we see

```
(inst 7 (LOAD r14 r21) ...)
```

In this case, the source is the location pointed to by `r21` and the destination is `r14`. The lookahead field indicates that the result will not be needed for at least 7 instructions (where only 3 bits are available for the lookahead field). Indeed, if we examine the loop, we see that `r14` is not used until the first instruction of the next iteration. The lookahead field informs the processor that there is no possible interaction between this instruction and the next 7 instructions, allowing significant instruction-level parallelism.

The `INT_MEM_ADD` operation atomically adds a value to a memory location. Specifically, the operation

```
(inst 1 (INT_MEM_ADD_INDEX r19 r8 r14) ...)
```

atomically adds the value in `r19` (in this case, 1) to the location pointed at by `r8 + 8*r14`. The lookahead field is 1, indicating that there is no possible interaction between this memory reference and the next instruction (since they reference different vectors), but that this `INT_MEM_ADD` must complete before the next begins.

Notice that a weakness of the compiler is exposed here. During parallelization, the compiler is able to recognize that the loop-carried dependence can be ignored since the updates are commutative

and can be executed atomically. Unfortunately, the software pipeliner is not as sophisticated and ensures that the updates are executed in order.

It is particularly interesting to note the interaction between the lightweight synchronization provided by the MTA and the compiler's ability to parallelize the loop, and then to software pipeline the iterations executed by each stream. If some sort of operating system interaction were required for each synchronization point (e.g., for each increment of `count[k]`), software pipelining would be impossible and parallelization might not be practical.

3 User Runtime

The software support for **future**[1] and **sync** language extensions [6], as well as for event logging, trap handling, compiler-generated parallelism, and debugging, is provided by a user-level library known as the Runtime. The Runtime interacts directly with some novel aspects of the MTA architecture, including per-stream exceptions, tag bits on memory, and dynamic resource allocation.

3.1 Exceptions and Trap Handling

Unlike most other machines, the trap handler on the MTA is at user level. With multiple streams running, and the possibility of each taking a trap concurrently, requiring the operating system (OS) to field all traps would be inefficient. The trap handler at user level allows each stream to service its traps independently of other streams and of the OS.

The floating-point trap handlers are good examples of the efficiency afforded by avoiding unnecessary detours into the OS. If a stream takes a floating-point exception that is not masked, the hardware causes the stream to jump to a user-level primary trap handler. The primary trap handler detects the cause of the trap (from the stream's private exception register) and calls the appropriate handler. The floating-point handler has the ability to fix the problem, log statistics, and return to the point where the trap occurred, all without interrupting any other stream.

3.2 Lightweight Synchronization

Very lightweight two-phase synchronization is supported by the hardware together with the user-level trap handlers. We use one of the data trap bits and the full-empty bit to implement producer-consumer semantics. The two bits are also used to implement future semantics, where all streams wait on empty for full.

When a synchronized or future access is made on a word whose full-empty bit is in the wrong state, the request returns to the processor with a failed status. The hardware places the request in a retry queue and automatically retries it (phase one). By retrying in hardware, no additional instruction issues are required; the processor can continue issuing instructions at the normal rate of one per clock tick. Thus, optimistic synchronization is cheaply supported without software interaction.

A retry limit register is associated with each protection domain. Each time a synchronization attempt fails, its retry count is incremented. When the retry count exceeds the protection domain's

[1] A future variable describes a location that will eventually receive the result of a computation. A future statement describes that computation and creates the opportunity for it to be executed.

limit, the stream traps, with control transferred to the primary trap handler (phase two). The primary trap handler dispatches to the data-blocked handler, which decides to block the operation (and let the stream do other work), retry the operation, or wake up a blocked thread that is waiting for this access to occur. If the stream is to block, it sets the trap bit on memory (a user-level operation) so that a subsequent satisfying access will trap and unblock it.

Locks, monitors, and barriers, all handled at user level, are easily and efficiently built on the lightweight synchronization that the hardware and the trap handler jointly implement.

3.3 Stream Management

Streams are reserved, created, and returned with user-level instructions. Their low cost and abundance allows them to be used in unique ways.

Daemons The architecture allows auxiliary daemons, streams dedicated to particular tasks, to be used without any significant penalty to the performance of the program.

A growth daemon is created when a program creates a future. The job of the growth daemon is to balance the hardware CPU resources used by the program with the current work available to the program. If there is a lot of work to be done, the daemon creates streams to do the work (note that it does this independently of the OS). As the work wanes, the streams quit. We believe the most effective use of the machine is to acquire those resources only when they are needed. This allows the system to be used in a shared setting, providing reduced turnaround times and increased throughput.

A debugger daemon is created when the debugger, tdb (an adaptation of gdb from the Free Software Foundation [8]), connects to the running program. The debugger daemon is responsible for controlling the program and communicating information about it to tdb, with which the user interacts. As the debugger daemon is running with the program, it can read all the memory of the program and selectively schedule its streams.

A profile daemon is created when the program is compiled and run with profiling/event logging. The user-level event-logging libraries export calls enabling the program to log events to a file. The events are first logged very quickly to internal memory buffers by the stream registering the event. This stream completes its log quickly, to minimize changes to the timing behavior of the program. The profile daemon empties the buffers to files, and creates helper daemons when necessary to keep up with demand.

Swapping Streams executing at user level are the responsibility of the Runtime. When the OS requires the program to unload from the protection domains, it sends a domain signal to the program. This action causes all the streams of the program to trap with a domain-signal exception. In the domain-signal handler, each stream saves its state and quits, with the last stream making a system call informing the OS saying the Runtime is ready for the swap. On reload, that last stream is restarted by the OS. It recreates all the other other streams directly and at user level. The newly created streams restore their own state and continue. This protocol demonstrates a nice separation of concerns between the user level and the OS enabled by the architecture. The user level handles all the streams of the program and, with the user-level trap handlers, can do this efficiently. The OS manages the larger resources of the task, such as the processor protection domains.

Dynamic Allocation As explained previously, a growth daemon is created when a program creates a future and is responsible for acquiring additional hardware streams and requesting processors as the parallel work available increases. Both the creation of the daemon and the acquisition process use features special to the MTA.

So that programs spawning no futures may avoid its overhead, albeit small, the growth daemon is created only once the first future is created. The mechanism for its creation uses the second data-trap bit of two associated with each memory location. Program initialization sets this trap bit on the variable acting as the index into the (initially empty) queue of future computations. The first thread to enqueue a future hits the trap bit, and is subsequently diverted by the trap handler to execute a function that creates the growth daemon; it then returns to the work of enqueing the future.

The growth daemon itself consumes a stream resource, and competes equally with other streams for the opportunity to issue instructions into the processor pipelines. To prevent the growth daemon from having a significant impact on performance, it is necessary to somehow reduce the average rate at which it issues instructions relative to other streams. One way to do this is to have it assess the need for growth only periodically instead of continually. This requires it to sleep in the interim, between assessments, thus executing instructions only while performing the assessments themselves.

Sleeping nominally requires OS intervention: the hardware provides no direct way to prevent a stream from executing for a specified duration. Fortunately, there is a cheaper and more fine-grained way to achieve the goal without interaction with the OS that makes unintended use of the hardware support features for synchronization. First, the daemon masks trap-bits in the exception register so that it will not trap. Then it performs a synchronized zero-lookahead load of an arbitrary empty memory location. The load will trip over the retry limit only after about ten-thousand cycles, during which the daemon will be prevented from executing any additional instructions. This reduces overall utilization of the daemon while sleeping to less than 0.05% of the available issue slots (although it does consume memory bandwidth). When the instruction does time-out, because traps are masked, the trap handler is not invoked. Instead the daemon checks the time, and insufficient has elapsed, it repeats the procedure.

The assessment of whether or not to add resources depends upon per-processor hardware counters: one of these counters accumulates the total number of unused instruction issue slots, or **pipeline bubbles**; another records the cumulative number of instructions ready to begin execution, and can be used to estimate average surplus parallelism. The assessment also depends upon the history of the work queue. From all of this information, the daemon estimates processor utilization or over-saturation, and determines whether or not and how to grow.

The growth daemon can create additional streams within a few instructions on an existing processor, just as does the compiler. However, requesting additional processors requires a call to the operating system. There is no need for the daemon or the user program to wait for these new processors to join in, however: whenever they become available, streams on the newly added processors can begin work, without explicit interaction with the already existing processors. This capability is possible given shared memory in which to hold the work queue.

Unlike growth, decay requires few special features: when streams cannot find work on the queue for some time, they quit.

4 Operating System

The Tera OS is a parallel, high-performance, multiprocessor operating system. It is fully symmetric, and distributed across all processors. Operating system services can be accessed from any processor and multiple users.

The OS has a microkernel-based architecture. The microkernel encapsulates the hardware interface and manages the hardware resources, providing memory management, process management, and scheduling services. All other services are provided by the supervisor layer, including networking, file systems, I/O, and Unix. The user-level Runtime provides management of user-level parallel activities.

The MTA provides four hardware privilege levels: IPL, KERNEL, SUPER, and USER. The microkernel and supervisor execute at KERNEL and SUPER levels, respectively. User code executes at USER level. IPL level is used only in rare instances for specialized operations. The hardware privilege levels provide a firewall between the software layers, insulating lower layers from malicious or errant code at higher layers. In particular, it protects the microkernel from the large body of imported code in the supervisor layer.

The microkernel is highly concurrent and distributed across all processors. One protection domain on each processor is reserved for exclusive use by the OS. This domain is populated by daemons, kernel-privileged threads that act as resource servers and/or execute remote requests to modify the processor hardware state. Daemons communicate through a lightweight message-passing facility (IPC), easily implemented via the MTA's shared memory. The IPC facility is built on the same underlying synchronization primitives used elsewhere in the OS. The concept of daemons communicating via IPC and executing in an event-driven manner has proved to be a flexible and useful paradigm for constructing system-resource servers.

4.1 No Hardware Interrupts

The MTA does not have hardware interrupts. In general, OS thread execution is non-preemptible; the only exception is domain signal, used on OS threads by the debugger. The lack of interrupts has simplified the code in some respects, and required some innovative solutions. Because there is no mechanism for interrupting a processor to gain its attention, the Tera OS dedicates one thread per processor to poll for external requests; we refer to this thread as the **listener**. The listener waits for requests to arrive, and creates threads to execute the requests. The requests are for operations that must execute on the processor; for instance, loading a new user program into a protection domain. The listener also replaces the conventional clock interrupt, and periodically services time-dependent activities, such as the alarm subsystem.

The lack of interrupts also affects the design of the I/O subsystem. There is no way to interrupt the processor to service the arrival of external data in a timely manner. The networking subsystem handles this by creating a work order for each arriving packet that is inserted into a global queue of work orders. A number of dedicated supervisor daemons service the queue in parallel. By varying the number of daemons, the system can ensure that even a large volume of incoming network packets are processed with reasonable efficiency, throughput, and scalability.

4.2 Daemons

The OS views concurrency as inexpensive and desirable, and uses streams to perform required services on a demand-driven basis. For instance, the scheduling subsystem generates events that are sent as IPC messages to the scheduler threads. The processor-scheduler thread blocks on an IPC channel, waiting for events to arrive. When an event arrives, the thread evaluates the request and creates a new thread to handle it.

IPC is often used to communicate between the microkernel and supervisor layers. Threads executing in the supervisor layer can make calls directly into the microkernel through a protected kernel library interface that changes the privilege level of the stream. However, there is no upcall interface to allow microkernel threads to execute supervisor code. Instead, IPC is used as a notification mechanism from the microkernel to the supervisor layer.

For example, the Unix timeout mechanism is implemented using a supervisor level timeout daemon and the microkernel listener. When a Unix timeout is posted, a corresponding alarm request is enqueued on a per-processor queue that is monitored by the listener. The timeout daemon is normally blocked on an IPC channel waiting for a message to arrive. When the alarm expires, the listener sends an IPC message to the timeout daemon. The timeout daemon examines the message and creates a daemon to execute the function associated with the timeout. This implementation approach preserves a clean interface between the Unix and microkernel functionality.

At supervisor level there is actually a thread system in place, such that tasks (threads) are put into a work pool and a set of daemons service that pool. During normal execution, the OS generates a large volume of work that must be processed in parallel in order to provide reasonable efficiency, throughput, and scalability. Much of this parallelism is achieved by having the originating stream do most of the work. There are occasions in which there is no stream associated with the generated work; for instance, the networking subsystem needs to process outgoing and incoming network packets in parallel. Outgoing packets are processed in parallel by the streams that are sending the packets. When an incoming packet arrives, however, the ultimate thread destined to receive the packet cannot be determined until the TCP header of the packet is analyzed. These operations are typically handled via software interrupts in Unix. With our OS, the task is described by a thread and put into a prioritized work pool. Daemons service the pool. If a high priority item arrives and no daemon is available, a daemon is created (subject to a maximum).

4.3 Symmetric Design

The OS data resides in globally shared memory. Shared memory allows system operations to be executed from any processor, and allows a fully symmetric OS design. The design of the IPC mechanism used for communication between OS daemons is also based on shared memory, which minimizes overhead and results in an efficient implementation.

4.4 Privilege-Level Change

The MTA hardware provides a special instruction that changes the privilege level of the executing thread. This allows an efficient implementation of system calls through transfer of privilege level. The system provides a fixed set of entry points in the supervisor-program address space; the entries serve as **gateways** into the privileged code. When the user thread executes a system call, it simply

branches to a gateway, where it is granted supervisor privileges while executing the routine. The microkernel uses an identical approach for providing entry from the supervisor layer.

4.5 Multiple Protection Domains

The MTA has sixteen protection domains per processor. This has allowed an innovative type of scheduling [1]. The OS dedicates one protection domain to itself while having fifteen domains to schedule amongst (up to fifteen) user programs.

Two types of schedulers are used to coordinate the scheduling of the user protection domains: a multiprocessor-scheduler schedules jobs spanning more than one processor (batch jobs); and a single-processor-scheduler schedules jobs on one processor (interactive jobs).

The decision to have two types of processor schedulers arose from the realization that batch jobs and interactive jobs have completely different requirements. Interactive jobs require quick response time but generally do not execute for large amounts of time. They are also, on the whole, smaller and execute in a single team on one processor. Batch jobs have the inverse requirements: they require large execution times and multiple processors but can handle not executing for longer periods of time.

While there is only one global multi-processor scheduler per machine, there are also single-processor schedulers for every processor. This organization helps minimize contention for data structures. Each small single-processor scheduler receives the majority of protection domains per processor. The multi-processor scheduler receives the minority. The distribution works well as typically we only want to run several batch jobs on a processor at once, in order to let them each acquire a larger number of stream resources. Interactive jobs take up the slack, and may be numerous.

4.6 No ROM Monitor

The MTA does not provide a ROM monitor to boot the OS, or to help diagnose problems with the OS. The Tera standalone subsystem provides much of the functionality normally provided by a ROM monitor. Standalone is loaded into memory early in the bootstrap sequence. It communicates with the maintenance workstation via a HIPPI channel that is multiplexed to provide communication for a variety of services. Standalone loads the OS image via the HIPPI channel, and starts a thread in the system initialization routine. Shared memory is used for communication between standalone and the OS.

After the system is loaded, the standalone module remains resident, and one thread sits in a polling loop, waiting for requests from the maintenance workstation.

An advantage of the standalone module is that as a program is loaded in by the primary boot, it can share sources with the main OS and provide a wide range of capabilities to the user. Functionality of standalone far surpasses that of a ROM monitor. It is also easier to update.

The disadvantage with this approach is that loading standalone requires a stable HIPPI connection and a bug-free rudimentary HIPPI driver in primary boot. This stability is something we have not always had during early system boot attempts on prototype hardware. Also, since standalone is fairly complex software, debugging is more difficult. Interactive debugging relies on the HIPPI connection and HIPPI drivers. For debugging the HIPPI drivers, we resort to core dumps.

5 Debugger

The Tera debugger, tdb, is both motivated and assisted by aspects of the architecture. While we kept the gdb-style command-line interface, most of the internals were rewritten to deal with multiple stream contexts, our symbol tables, program libraries, and compiler optimizations such as parallelization and inlining. We also use a different approach for the tdb process to communicate with the running program.

Most conventional debuggers [7, 8] communicate with the running program using a ptrace or proc type mechanism. Here, the operating system intervenes to help obtain information about the program. On the MTA, where stream state is the responsibility of the user-level Runtime, having the OS intervene is not necessary or appropriate. Our debugger communicates through a socket connection to a user-level stream that runs in the program and is dedicated to debugging. There is an established protocol between the tdb process and the debugger daemon (debugger nub) in the program. Tdb can ask the nub for information about the threads, to evaluate expressions, to set breakpoints, to continue a certain thread, etc. The daemon either does the operations itself (like setting breakpoints or evaluating expressions) or directs the streams appropriately (continue this thread but not another). As the daemon is running with the other streams of the program, it can read and write memory directly, as well as gather information about the streams.

A limitation of the coexistence of the debugger nub and the program is that an errant program could corrupt the debugger specific memory. To help limit this effect we have the debugger use a separate malloc pool than the rest of the program.

5.1 Expression Evaluation

In most cases, tdb does not evaluate expressions itself but rather sends a message containing the expression to the nub for evaluation within the program. We moved expression evaluation to the nub in order to implement conditional breakpoints efficiently – we did not want to require all streams to stop in order to evaluate a single stream's condition (especially as thousands of streams could be running). Every time the program stops to communicate with tdb, its timing is changed. Instead, the stream taking the breakpoint evaluates the associated condition itself. Only when the condition is true does control transfer to tdb. This procedure scales perfectly; as many streams as necessary can evaluate their own conditions concurrently.

The OS incorporates a separate but equivalent debugger nub of its own, allowing it to have a full set of debugging capabilities without the need for a special kernel debugger. Furthermore, we have extended this nub concept to support the debugging of both user-level and OS core files. We created nubs that interact with core files, providing the same high-level tdb debugging interface whether you are running on the system or debugging off-line.

5.2 Breakpoints

Multiple streams sharing instruction space pose a problem with breakpoint handling. Conventional debuggers implement breakpoints with a break instruction. To step past the breakpoint, the original instruction is rewritten into text memory, the program steps over the instruction, and the break instruction is written back. Were this technique used in the parallel context of the MTA, one stream could miss the breakpoint.

We have developed an out-of-line instruction execution scheme to address this problem. We have built an interpreter that, given an instruction and a program counter/text address, computes a block of instructions, an out-of-line buffer, that mimic the original instruction executed at a different address. The break instruction is left in place; a special breakpoint trap handler reroutes the stream through the out-of-line buffer, which, as its final act, jumps to where the original instruction would have ended up. Typically execution then continues at the address following the breakpoint. Many instructions can simply be duplicated in the out-of-line buffer. Branching and potential trapping behavior, however, introduce complexity.

The out-of-line instruction execution has proven very useful. When the debugger daemon needs to sidestep a breakpoint set in the nub, it simply executes the out-of-line instruction buffer. Continuing after a conditional breakpoint (that has proven false) is also easy and requires no tdb interaction.

5.3 Watchpoints

While one of the data trap bits is used for synchronization, the second trap bit is used to implement watchpoints. To set a watchpoint, the debugger turns on the variable's associated trap bit. If a stream reads or writes the variable, the hardware jumps it to the primary trap handler. The handler dispatches to a routine registered with that variable, which, in the case of the debugger, is a watchpoint handler. Very fast and practical watchpoints are the result of the MTA's tagged memory. In practice, with the large amount of parallelism running in programs, it has proven very useful to watch memory instead of watching streams.

6 Conclusions

The Tera MTA has a number of interesting hardware features whose virtues are not immediately apparent. In this paper, we have shown how these features, in cooperation with various software techniques, enable superior solutions to well-known problems. This synergy is not the result of luck or an omniscient architect: it is the result of designing hardware and software together.

Because realizing each of these features in hardware has a corresponding cost, it is worth asking, in hindsight, which of them are really necessary for effective parallel computation. Most other architectures support few of the major features highlighted in the introduction. Can parallel computers be designed perhaps at lower cost when absent the features, yet with no loss of utility?

True shared memory is the cornerstone of the Tera MTA design. Without it, programmers or compilers are obliged to ensure most data is physically close to the processor at the time it is operated upon, or suffer from severe performance degradation. This data layout requirement restricts choice of algorithm and implementation, and inhibits the scaling of some highly parallel applications. So, even neglecting benefits of shared memory to compiler development, we know that it is a mandatory feature of any general purpose parallel computer that is easy to program.

Fast stream creation is not necessary, but it allows for exploitation of extremely fine grained parallelism, even when that parallelism varies rapidly during the course of execution. With slower stream creation, we would lose efficiency on problems with erratic parallelism, and some of our compiler design decisions would change. Because MTA stream creation is at user-level, the operating system is simpler, and the compiler can reason about the costs of parallelization.

The full-empty and memory-trap bits are required to implement synchronization without busy waiting. There may be other implementations of synchronization without busy waiting, but this one entails only moderate expense and has proven extremely versatile. For example, the same feature is used to implement fast watchpoints in the debugger.

Wide instructions and lookahead provide a way to utilize instruction-level parallelism. Both are mandated by their cost-benefit for single instruction stream performance: without wide instructions, we would require a faster clock to get the same amount of computation done; without lookahead, we would require a larger number of streams to tolerate the same amount of memory or synchronization latency.

Without multiple protection domains per processor, only one job could be resident per processor at any time. Most programs are not parallel all the time; consequently processors would be under-utilized. Furthermore, multiple protection domains allow full processor utilization even while some jobs are being swapped out; again, without them, the processors would be very poorly utilized during swap periods.

Developing both MTA software and hardware together has ensured that hardware features support the needs of software systems. Solutions to challenges associated with parallel computer systems have been addressed economically: where both a hardware and a software solution is possible, realization into one or the other has been intentional. Overall, the features of the MTA provide an elegant basis for implementing both system and application software.

References

[1] Gail Alverson, Simon Kahan, Richard Korry, Cathy McCann, and Burton Smith. Scheduling on the Tera MTA. In *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

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

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Fredrickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks – Summary and preliminary results. In *Proceedings of Supercomputing '91*, pages 158–165, November 1991.

[4] David Callahan. Recognizing and parallelizing bounded recurrences. In U. Banerjee, D. Gellernter, A. Nicoalu, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 589 of *Lecture Notes in Computer Science*, pages 169–185. Springer-Verlag, 1992.

[5] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. *SIGPLAN Notices*, 25(6):53–65, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[6] David Callahan and Burton Smith. A future-based parallel language for a general-purpose highly-parallel computer. In David Gelernter, Alexandru Nicolau, and David Padua, editors,

Languages and Compilers for Parallel Computing, pages 95–113. The MIT Press, Cambridge, Massachusetts, 1990.

[7] Mark Linton. The evolution of DBX. In *USENIX Summer Conference*, 1990.

[8] Richard Stallman and Cygnus Support. Debugging with GDB, January 1994.

[9] Roy F. Touzeau. A Fortran compiler for the FPS-164 Scientific Computer. *SIGPLAN Notices*, 19(6):48–57, June 1984. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.*

[10] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.