The Tera Computer System*

Robert Alverson David Callahan Daniel Cummings Brian Koblenz
Allan Porterfield Burton Smith

Tera Computer Company
Seattle, Washington USA

1 Introduction

The Tera architecture was designed with several major goals in mind. First, it needed to be suitable for very high speed implementations, *i. e.*, admit a short clock period and be scalable to many processors. This goal will be achieved; a maximum configuration of the first implementation of the architecture will have 256 processors, 512 memory units, 256 I/O cache units, 256 I/O processors, and 4096 interconnection network nodes and a clock period less than 3 nanoseconds. The abstract architecture is scalable essentially without limit (although a particular implementation is not, of course). The only requirement is that the number of instruction streams increase more rapidly than the number of physical processors. Although this means that speedup is sublinear in the number of instruction streams, it can still increase linearly with the number of physical processors. The price/performance ratio of the system is unmatched, and puts Tera's high performance within economic reach.

Second, it was important that the architecture be applicable to a wide spectrum of problems. Programs that do not vectorize well, perhaps because of a preponderance of scalar operations or too-frequent conditional branches, will execute efficiently as long as there is sufficient parallelism to keep the processors busy. Virtually any parallelism available in the total computational workload can be turned into speed, from operation level parallelism within program basic blocks to multiuser time- and space-sharing. The architecture even has strong support for implementing non-numeric languages like Lisp and Prolog and highly applicative languages like Sisal and Id.

A third goal was ease of compiler implementation. Although the instruction set does have a few unusual features, these do not seem to pose unduly hard problems for the code generator. There are no register or memory addressing constraints and only three addressing modes. Condition code setting is consistent and orthogonal. Although the richness of the instruction set often allows several ways to do something, the variation in their relative costs as the execution environment changes tends to be small. Because the architecture permits the free exchange of spatial and temporal locality for parallelism, a highly optimizing compiler may work hard improving locality and trade the parallelism thereby saved for more speed. On the other hand, if there is sufficient parallelism the compiler has a relatively easy job.

The Tera architecture is derived from that of Horizon [6, 9, 10]; although they are highly similar multistream MIMD systems, there are many significant differences between the two designs.

2 Interconnection Network

The interconnection network is a three-dimensional mesh of pipelined packet-switching nodes, each of which is linked to some of its neighbors. Each link can transport a packet containing source and destination addresses, an operation, and 64 data bits in both directions simultaneously on every clock tick. Some of the nodes are also linked to *resources*, *i. e.*, processors, data memory units, I/O processors, and I/O cache units. Instead of locating the processors on one side of the network and memories on the other (in what Robert Keller has called a "dancehall" configuration[5]), the resources are distributed more-or-less uniformly throughout the network. This permits data to be placed in memory units near the appropriate processor when that is possible and otherwise generally maximizes the distance between possibly interfering resources.

*This research was supported by the United States Defense Advanced Research Projects Agency under Contract MDA972-89-C-0002. 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.

The interconnection network of a 256 processor Tera system contains 4096 nodes arranged in a 16x16x16 toroidal mesh; that is, the mesh "wraps around" in all three dimensions. Of the 4096 nodes, 1280 are attached to the resources comprising 256 processors, 512 data memory units, 256 I/O cache units and 256 I/O processors. The 2816 remaining nodes do not have resources attached but still provide message bandwidth. To increase node performance, some of the links are missing. If the three directions are named X, Y, and Z, then X-links and Y-links are missing on alternate Z layers. This reduces the node degree from 6 to 4, or from 7 to 5 counting the resource link.

In spite of its missing links, the bandwidth of the network is very high. Any plane bisecting the network crosses at least 256 links, giving the network a data bisection bandwidth of one 64-bit data word per processor per tick in each direction. This bandwidth is needed to support shared memory addressing in the event all 256 processors are addressing memory on the other side of some bisecting plane simultaneously.

As the Tera architecture scales to larger numbers of processors p, the number of network nodes grows as $p^{3/2}$ rather than the $p\log(p)$ associated with the more commonly used multistage networks. For example, a 1024-processor system would have 32,768 nodes. The reason for the overhead per processor of $p^{1/2}$ instead of $\log(p)$ stems from the fact that the system is speed-of-light limited. One can argue that memory latency is fully masked by parallelism only when the number of messages being routed by the network is at least $p \times l$, where l is the (round-trip) latency. Since messages occupy volume, the network must have volume proportional to $p \times l$; since the speed of light is finite, the volume is also proportional to l^3 and therefore l is proportional to $p^{1/2}$ rather than $\log(p)$.

3 Memory

A full-sized system contains 512 data memory units of 128 megabytes each. Memory is byte-addressable, and is organized in 64-bit words. Four additional *access state* bits, more fully described in section 5, are associated with each word. Data and access state are each equipped with a separate set of single error correcting, double error detecting code bits. Data addresses are randomized in the processors using a scheme similar to that developed for the RP3[8]. The randomization is excellent for avoiding memory bank hotspots and network congestion, but makes it difficult to exploit memory locality using nearby memory units. In the Tera system, the randomization is combined with another notion called *distribution*. The processor data segment map has a distribution factor associated with each segment entry. Consecutive virtual addresses in a segment can be distributed among all 512 data memory units, a single unit, or any power of two in between.

Disk speeds have not kept pace with advances in processor and memory performance in recent years. The only currently reasonable solution to this problem is to lower the level of disks in the memory hierarchy by placing a large semiconductor memory between the disks and data memory. In a fully configured Tera system, the 70 gigabyte per second sustained bandwidth needed between secondary storage and data memory is supplied by 256 I/O cache units comprising a directly addressable memory of 256 gigabytes.

The I/O cache units are functionally identical to data memory. The only difference is that their latency is higher because their memory chips are slower (but denser). The fact that I/O cache has all of the attributes of main memory makes it possible to map I/O buffers directly into the address spaces of the application programs that access them. This is used to avoid copying by remapping segments.

A processor fetches instructions through a special path to a neighboring I/O cache unit. This avoids network traffic and network latency, but requires one copy of a program be made for every processor it is to run on.

4 Processors

Each processor in a Tera computer can execute multiple instruction streams simultaneously. In the current implementation, as few as one or as many as 128 program counters may be active at once. On every tick of the clock, the processor logic selects a stream that is ready to execute and allows it to issue its next instruction. Since instruction interpretation is completely pipelined by the processor and by the network and memories as well, a new instruction from a different stream may be issued in each tick without interfering with its predecessors. When an instruction finishes, the stream to which it belongs thereby becomes ready to execute the next instruction. As long as there are enough instruction streams in the processor so that the average instruction latency is filled with instructions from other streams, the processor is being fully utilized. Thus, it is only necessary to have enough streams to hide the expected latency (perhaps 70 ticks on average); once latency is hidden the processor is running at peak performance and additional streams do not speed the result.

If a stream were not allowed to issue its next instruction until the previous instruction completed then approximately 70 different streams would be required on each processor to hide the expected latency. The lookahead described in section 4.3 allows streams to issue multiple instructions in parallel, thereby reducing the number of streams needed to achieve peak performance.

4.1 Stream State

Each stream has the following state associated with it:

- 1 64-bit Stream Status Word (SSW)
- 32 64-bit General Registers (R0-R31)
- 8 64-bit Target Registers (T0-T7)

Context switching is so rapid that the processor has no time to swap the processor-resident stream state. Instead, it has 128 of everything, *i. e.*, 128 SSW's, 4096 general registers, and 1024 target registers. It is appropriate to compare these registers in both quantity and function to vector registers or words of cache in other architectures. In all three cases, the objective is to improve locality and avoid reloading data.

Program addresses are 32 bits long. Each stream's current program counter is located in the low half of its SSW. The upper half describes various modes (*e. g.* floating point rounding, lookahead disable), the trap disable mask (*e. g.* data alignment, floating overflow), and the four most recently generated condition codes. Most operations have a `_TEST` variant which emits a condition code, and branch operations can examine any subset of the last four condition codes emitted and branch appropriately.

Also associated with each stream are 32 64-bit general registers. Register `R0` is special in that it reads as 0, and output to it is discarded. Otherwise, all general registers are identical.

The target registers are used as branch targets. The format of the target registers is identical to that of the SSW, though most control transfer operations only use the low 32 bits to determine a new PC. Separating the determination of the branch target address from the decision to branch allows the hardware to prefetch instructions at the branch targets, thus avoiding delay when the branch decision is made. Using target registers also makes branch operations smaller, resulting in tighter loops. There are also skip operations, which obviate the need to set targets for short forward branches.

One target register (`T0`) points to the trap handler, which is nominally an unprivileged program. When a trap occurs, the effect is as if a coroutine call to `T0` had been executed. This makes trap handling extremely lightweight and independent of the operating system. Trap handlers can be changed by the user to achieve specific trap capabilities and priorities without loss of efficiency.

4.2 Horizontal Instructions

Processor effectiveness, the utilization of the instruction interpretation resources, has always been constrained by the difficulty of issuing more than one instruction per tick. This difficulty has become known as the Flynn Bottleneck[2]. Vector instructions sidestep this difficulty in part, but are not able to handle frequent conditional branches or heterogeneous scalar operations well. Processors with horizontal instructions, extreme examples of which are sometimes called Very Long Instruction Word (VLIW) architectures, offer a good alternative to vector instructions. In a horizontal instruction, several operations are specified together. Memory operations are usually simple loads and stores, and the others are two- or three-address register-to-register operations. If the overall architecture and organization are capable of achieving one instruction per tick, then every functional unit mentioned in the instruction is well-used. If the instructions are only moderately long, branches can be sufficiently frequent.

Tera instructions are mildly horizontal. They typically specify three operations: a memory reference operation like `UNS_LOADB`(yte), an arithmetic operation like `FLOAT_ADD_MUL`(tiply), and a control operation like `JUMP`. The control operation can also be a second arithmetic operation, `FLOAT_ADD`, or perhaps an `INTEGER_ADD` used in an address computation. Vectorizable loops can be processed at nominal vector rates (one flop per tick) using only horizontal instructions with these three kinds of operations. Matrix-vector multiplication attains nearly two flops per tick via the same technique used for its efficient vectorization.

4.3 Explicit-Dependence Lookahead

If there are enough streams executing on each processor to hide the average latency (about 70 ticks) then the machine is running at peak performance. However, if each stream can execute some of its instructions in parallel (*e. g.* 2 successive loads) then fewer streams and parallel activities are required to achieve peak performance.

The obvious solution to this problem is to introduce instruction lookahead; the only difficulty is controlling it. The traditional register reservation approach requires far too much scoreboard bandwidth in this kind of architecture. Either multi-streaming or horizontal instructions alone would preclude scoreboarding. The traditional alternative, exposing the pipeline, is also impractical because multi-streaming and unpredictable memory operation latency make it impossible to generate code that is both efficient and safe.

The Tera architecture uses a new technique called explicit-dependence lookahead. The idea is quite simple: each instruction contains a three bit lookahead field that explicitly specifies how many instructions from this stream will issue before encountering an instruction that depends on the current one. Since seven is the maximum possible lookahead value, at most eight instructions and twenty-four operations can be concurrently executing from each stream. A stream is ready to issue a new instruction when all instructions with lookahead values referring to the new instruction have completed. Thus, if each stream maintains a lookahead of seven then nine streams are needed to hide 72 ticks of latency.

Lookahead across one or more branch operations is handled by specifying the minimum of all distances involved. The variant branch operations `JUMP_OFTEN` and `JUMP_SELDOM`, for high- and low-probability branches respectively, facilitate optimization by providing a barrier to lookahead along the less likely path. There are also `SKIP_OFTEN` and `SKIP_SELDOM` operations. The overall approach is philosophically similar to exposed-pipeline lookahead except that the quanta are instructions, not ticks.

4.4 Protection Domains

Each processor supports as many as 16 active protection domains that define the program memory, data memory, and number of streams allocated to the computations using that processor. Each executing stream is assigned to a protection domain, but which domain (or which processor, for that matter) is not known to the user program. In this sense, a protection domain is a virtual processor and may be moved from one physical processor to another.

The protection domains share a single 64K data segment map and a 16K program page map. Each protection domain has two pairs of map base and limit registers that describe the region of each map available to it. The upper 2048 data segments and 1024 program pages are not relocated by the map bases, and are used by the operating system. Any active protection domain can use all of either or both maps. The map entries contain the physical address; the levels of privilege needed to read, write, or execute the segment or page; whether the segment or page was read, written, or executed, as appropriate; and the distribution (for the data map).

The number of streams available to a program is regulated by three quantities `slim`, `scur`, and `sres` associated with each protection domain. The current number of streams executing in the protection domain is recorded by `scur`; it is incremented when a stream is created and decremented when a stream quits. A create can only succeed when the incremented `scur` does not exceed `sres`, the number of streams reserved in the protection domain. The operations for reserving streams are unprivileged, and allow several streams to be reserved or released simultaneously. The stream limit `slim` is the operating system limit on the number of streams the protection domain can reserve.

When a stream executes a `CREATE` operation to create a new stream it increments `scur`, generates the initial SSW for the stream using one of its own target registers, copies the trap target `T0` from its own `T0` register, and loads three registers in the new stream from its own general purpose registers. The newly created stream can quickly begin executing useful work in cooperation with its creator as long as significant storage allocation is unnecessary. The `QUIT` operation terminates the stream that executes it, and decrements both `sres` and `scur`. The `QUIT_PRESERVE` operation only decrements `scur`, thereby giving up a stream without surrendering its reservation.

Each protection domain has a *retry limit* that determines how many times a memory reference can fail in testing a location's full/empty bit (see section 5) before it will trap. If a synchronization is not satisfied for a long time, then possibly a heavier weight mechanism that avoids busy waiting should be used to wait for the synchronization. The retry limit should be based on the amount of trap processing overhead, which varies depending on the run-time environment. The trap handler thus can invoke the heavier weight mechanism when appropriate.

4.5 Privilege Levels

The privilege levels apply to each stream independently. There are four levels of privilege: user, supervisor, kernel, and IPL. IPL level operates in absolute addressing mode and is the highest privilege level. User, supervisor, and kernel levels use the program and data maps for address translation, and represent increasing levels of privilege. The data map entries define the minimum levels needed to read and write each segment, and the program map entries define the *exact* level needed to execute from each page. The current privilege level of a stream is stored as part of the privileged stream state and is not available to a user-level stream.

Two hardware operations are provided to allow an executing stream to change its privilege level. The (`LEVEL_ENTER` *lev*) operation sets the current privilege level to the instruction map level if the current level is equal to *lev*. The `LEVEL_ENTER` operation is located at every entry point that can accept a call from a different privilege level. A trap occurs if the current level is not equal to *lev*. The (`LEVEL_RETURN` *lev*) operation is used to return to the original privilege level. A trap occurs if *lev* is greater than the current privilege level.

4.6 Exceptions

Exceptional conditions can occur in two ways. First, an instruction may not be executed due to insufficient privilege, as with a `LEVEL_RETURN` which attempts to raise the privilege level. This type of exception is quite easy to handle. More commonly, exceptions occur *while* executing instructions. With lookahead, further instructions may already be executing and overwriting registers which would be needed to restart instructions.

Rather than keep shadow copies of registers to support rollback, the Tera architecture defines certain exceptions as a side effect of instruction *completion*. In this model, exceptions are guaranteed to be signaled before they are needed, as indicated by the lookahead field. Thus, if instruction j depends on instruction i, all possible exceptions during the execution of instruction i will be signaled before instruction j begins execution.

To support diagnosis and recovery, certain state must be available to the trap handler. A trap can be caused by any of the three operations in an instruction. For each of the (at most eight) memory operations that trapped, the processor provides the trap handler with the trap reason and enough state to allow the operation to be retried (*e. g.* for demand paged virtual memory).

For arithmetic traps caused by the arithmetic operations no state is automatically provided to the trap handler. The decision to preserve operand values for possible use by the trap handler is made by the compiler. While the lookahead field normally only guards true dependence for registers, operand values may be preserved by limiting lookahead to guard antidependence as well.

5 Tagged Memory

Each memory location in the Tera computer system is equipped with four access state bits in addition to a 64-bit value. These access state bits allow the hardware to implement several useful modifications to the usual semantics of memory reference. The two data trap bits generate application-specific lightweight traps, the forward bit implements invisible indirect addressing, and the full/empty bit is used for lightweight synchronization. The influence of these access state bits can be suppressed by a corresponding set of bits in the pointer value used to access the memory.

The two trap bits in the access state are independent of each other and are available for use by the language implementer. If a trap bit is set in a location and the corresponding trap disable bit in the pointer is clear, a trap will occur. Uses for the trap bits include data breakpoints, demand-driven evaluation, run-time type exception signaling, implementation of "active" memory objects, and even stack limit checking.

The forward bit implements a kind of "invisible indirection". Unlike normal indirection, forwarding is controlled by both the pointer and the location pointed to. If the forward bit is set in the memory location and forwarding is not disabled in the pointer, the value found in the location is to be interpreted as a pointer to the target of the memory reference rather than the target itself. Dereferencing will continue until the pointer either disables forwarding or discovers that the addressed location has its forward bit reset. The primary use of forwarding is for on-the-fly modification of address-location bindings, for example in concurrent storage reclamation involving the copying of live structures from one space to another.

The full/empty bit controls the synchronizing behavior of memory references. Load and store operations can optionally use the full/empty bit in the addressed memory word by setting bits in the access control field. The four values for access control are shown below.

value	`LOAD`	`STORE`
0	read regardless	write regardless and set full
1	reserved	reserved
2	wait for full and leave full	wait for full and leave full
3	wait for full and set empty	wait for empty and set full

When access control is 2, loads and stores wait for the memory cell to be full before proceeding. In this context, it is sometimes useful to think of the full state as meaning "available" and the empty state as meaning "unavailable". The reading or writing of any part of an object is conveniently prevented by marking that part of it "unavailable". The access control value of 3 causes loads to be treated as "consume" operations and stores as "produce" operations. A load waits for full and then sets empty as it reads, and a store waits for empty and then sets full as it writes. A forwarded location that is not disabled and that has its full/empty bit set to empty is treated as "unavailable" until it fills again, irrespective of access control.

Additional operations exist to fetch the access state of a given memory location or to set the access state for a given location.

Although the full/empty bit provides a fast way of implementing arbitrary indivisible memory operations, the need for extremely brief mutual exclusion during "integer add to memory" is so important for scheduling applications that this function is done entirely within each memory unit by a single operation, `FETCH_ADD`. This is the Ultracomputer fetch-and-add operation[3], and differs from it only in that the network hardware does not combine fetch-and-add operations to the same memory location.

6 Arithmetic

The numeric data types directly supported by the Tera architecture include:

- 64 bit twos complement integers
- 64 bit unsigned integers
- 64 bit floating point numbers
- 64 bit complex numbers

Operations on these types include addition, subtraction, multiplication, conversion, and comparison. Reciprocation of unsigned and floating point quantities is provided for using Newton's method.

Other types are supported indirectly, including:

- 8, 16, and 32 bit twos complement integers
- 8, 16, and 32 bit unsigned integers
- arbitrary length unsigned integers
- 32 bit floating point numbers
- 128 bit "doubled precision" numbers

The shorter integers are sign- or zero-extended to 64 bit quantities as they are loaded from memory, and truncated to the appropriate length as they are stored. The fundamental support for arbitrary length integer arithmetic is provided by the operations `INTEGER_ADD_MUL`, `UPPER_ADD_MUL`, and `CARRY_ADD_TEST` that together implement $64 \times n$ bit unsigned multiply-add in approximately $2 \times n^2$ instructions.

The 32 bit floating point numbers are simply the real parts of the 64 bit complex type with imaginary parts set to zero. The 128 bit "doubled precision" type was pointed out to us by Kahan [1, 7, 4]; it represents a real number R as the unevaluated sum of two 64 bit floating point numbers r and ρ, where ρ is insignificant with respect to r and as near as possible to $R-r$. Support for this type is provided by `FLOAT_ADD_LOWER` which (with `FLOAT_ADD`) implements "doubled precision" addition in six instructions, and by `FLOAT_MUL_ADD` which rounds only once and is used to implement "doubled precision" multiplication in five instructions.

References

[1] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Math.*, 18:224–242, 1971.

[2] M. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[3] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, 1984.

[4] W. Kahan. Doubled-precision IEEE standard 754 floating-point arithmetic. Unpublished manuscript, February 1987.

[5] R. M. Keller. Rediflow: A proposed architecture for combining reduction and dataflow. In *PAW83: Visuals Used at the 1983 Parallel Architecture Workshop*, University of Colorado, Boulder, 1983.

[6] J. T. Kuehn and B. J. Smith. The Horizon supercomputer system: Architecture and software. In *Proceedings of Supercomputing '88*, Orlando, Florida, November 1988.

[7] S. Linnainmaa. Software for doubled-precision floating-point computations. *ACM Transactions on Mathematical Software*, 7:272–283, 1981.

[8] A. Norton and E. Melton. A class of boolean linear transformations for conflict-free power-of-two stride access. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 247–254, August 1987.

[9] Frank Pittelli and David Smitley. Analysis of a 3d toriodal network for a sihared memory architecture. In *Proceedings of Supercomputing '88*, Orlando, Florida, November 1988.

[10] M. R. Thistle and B. J. Smith. A processor architecture for Horizon. In *Proceedings of Supercomputing '88*, pages 35–41, Orlando, Florida, November 1988.