Exploiting Heterogeneous Parallelism on a Multithreaded Multiprocessor*

Gail Alverson Robert Alverson David Callahan
Brian Koblenz Allan Porterfield Burton Smith

Tera Computer Company
Seattle, Washington USA

Abstract

This paper describes an integrated architecture, compiler, runtime, and operating system solution to exploiting heterogeneous parallelism. The architecture is a pipelined multithreaded multiprocessor, enabling the execution of very fine (multiple operations within an instruction) to very coarse (multiple jobs) parallel activities. The compiler and runtime focus on managing parallelism within a job, while the operating system focuses on managing parallelism across jobs. By considering the entire system in the design, we were able to smoothly interface its four components. While each component is primarily responsible for managing its own level of parallel activity, feedback mechanisms between components enable resource allocation and usage to be dynamically updated. This dynamic adaptation to changing requirements and available resources fosters both high utilization of the machine and the efficient expression and execution of parallelism.

1 Introduction

An application set contains parallelism at many levels. Fine grain parallelism within an application can range from pipelined or co-scheduled operations to tightly parallel loops. Medium grain parallelism, on the order of hundreds of instructions, describes more general, varied, and largely independent activities. Coarse grain parallelism presents itself in the form of applications (or large application sections) that can run simultaneously in a multiprogrammed fashion. We use the phrase *heterogeneous parallelism* to refer to this spectrum of parallelism with widely varying grain-size and resource requirements.

This paper describes an integrated hardware and software system designed to exploit heterogeneous parallelism. Each of the hardware, compiler, runtime, and operating system components are responsible for managing parallelism at different levels. In addition, the components cooperate to adapt to changing parallel and resource requirements.

Briefly, the most important attribute of the hardware is that it is multithreaded. In a multithreaded architecture, each physical processor supports some number of instruction streams, each of which is programmed essentially like a traditional uniprocessor. These multiple streams are then multiplexed by the processor hardware onto a single set of functional units. Our principle motivation for adopting multithreaded processors is to tolerate latency from memory units and synchronization. Because these latencies grow with the size of the multiprocessor, they must be tolerated for an effectively scalable machine.

The most important attribute of the overall system is the virtual machine it presents to the parallel programmer. This, in turn, largely influences how easily the parallel machine can be used. The virtual machine defined by our parallel language has an unbounded number of "processors", each with uniform access to a shared memory. The language is a traditional imperative language augmented with data and control constructs to specify thread creation and synchronization. The fiction of an unbounded number of processors is maintained by the software through time-sharing when the semantics of the program require more logical processors than avail-

*This research was supported 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.

able streams. The fiction of uniform access is maintained by the multithreading, which masks waiting in one stream with computation in another.

We focus on answering two questions that are central to systems that support heterogeneous parallelism.

- What is the role of each component of the system — the hardware, operating system, compiler, and runtime — in detecting and using parallelism within and across applications?
- How can the system, as a whole, adapt to the changing resource requirements of its applications?

The second question stems from research concerning the amount of the available parallelism in programs[8, 11]. Results indicate that the parallelism in a program can vary by orders of magnitude over relatively short time intervals. Optimized numerical programs also show wide and rapid variance in available parallelism. Good machine utilization requires that one find a way to smooth out these resource profiles. A dynamic solution to resource management, in which stream resources can move from one task to another within and across jobs, appears most reasonable and unobtrusive to users. Intuitively, peaks of one computation can be used to fill the valleys of another.

The Tera system, reflecting our work, is intended for commercial use in a large scale multi-user scientific computing environment. Machines will have up to 256 processors, with approximately 32,000 streams available in a full size machine. This paper describes how each component of the system manages its particular level of parallelism. We also identify the information each component must provide to the other, in order to adapt to dynamic changes in application parallelism and resources. Interestingly, and in part due to our concurrent design of the system, the set of communication channels needed to provide this integration is small.

The next section contrasts the Tera system with other multithreaded systems. This is followed in Section 3 by a brief description of our architecture. Section 4 outlines support for very fine-grained parallelism. Section 5 then discusses the special needs of compiler generated parallelism. Section 6 describes the general runtime environment with emphasis on efficient synchronization. Section 7 outlines operating system concerns of allocating resources to different jobs.

2 Related Systems

The general need for multithreading arises whenever there is insufficient parallelism available within a single stream to keep the machine resources busy in the face of resource latency. Resource latency includes memory latency, synchronization latency, and processing latency. Rather than simply allowing the hardware resources to idle, they are shared among multiple streams.

A primary motivation for multithreaded architectures is to provide scalable shared memory. *Scalable* means that as the number of processors increase, the amount of work that can be performed in a given time interval increases proportionately. *Shared memory* refers to a communication paradigm wherein parallel entities communicate through reading and writing locations in a single shared address space. Ideally, shared memory provides the illusion of uniform memory access.

Miller. One of the early multithreaded designs was the multiple-stream registerless shared-resource processor proposed by Miller in 1974 [13]. The main thrust of the design is to use queues to hold outstanding operations at each stage in the instruction processing. At each clock tick, an element of each queue that is "ready" is processed and moved to the next queue in the pipeline. By providing for a large number of instruction streams, the processing elements associated with each queue are kept busy a large fraction of the time.

HEP. The Denelcor HEP system [15], introduced in 1978, was the first commercial multithreaded multiprocessor. A HEP system contains up to 16 processors connected by a high bandwidth switch. Each processor has support for 16 protection domains and 128 instruction streams. A protection domain defines a virtual address space; each job executing on the same processor has its own protection domain. An instruction stream is backed by a program counter and a portion of a per-processor shared register file. The processor's instruction issuing mechanism enables fast instruction level task switching in the hardware. In addition to multithreading, a key feature of the HEP is its pipelining of both the memory and ALUs to increase bandwidth. Because the active instructions in the pipes come from different streams, they are usually independent and interlocks are seldom encountered.

Synchronization in the HEP is achieved with a full/empty bit on each memory location and general purpose register. A write to a location is implemented by an instruction that, when empty, indivisibly writes the location and sets it full. A read, similarly, is implemented by an instruction that, when full, indivisibly reads the location and sets it empty. When the synchronization condition does not hold, the instruction is reattempted on the stream's next turn for execution. Because the register set is shared among all streams of a processor, the HEP includes register reservation bits to ensure atomicity of operations on shared registers.

MASA. MASA [9] is a multithreaded architecture design that has many similarities with the HEP architecture. A novel feature of MASA is its use of parallel streams for lightweight procedure calls and traps. This feature is supported by MASA's bank of register sets, which is shared in the style of register windows. As well as having their own registers, spawned streams share several registers with their parent. When a trap occurs, a new stream is created as a trap handler. This creation automatically protects the context of the trapped stream, yet allows its content to be probed by the handler. Similarly, by creating a new stream for each procedure invocation, parameters of the procedure are passed in the shared registers automatically. New registers are available, at no additional cost.

Horizon. The Horizon [10], a successor of the HEP, adds to the family of shared memory MIMD systems for scientific computation.

Unlike the HEP, each stream of the Horizon has its own private register set. This change eliminates the need for full/empty and reserved bits on the registers, allowing faster clock rates and more parallelism in register access. The Horizon architecture is horizontal, with each 64-bit instruction having the potential to simultaneously initiate a memory, an ALU, and a control operation. Each instruction contains a 3-bit lookahead field to allow multiple outstanding operations without hardware register interlocks. The compiler sets lookahead to indicate the number of subsequent instructions that can be issued without waiting for the completion of this instruction. A full/empty bit on memory locations is used for synchronization of streams. Unlike the HEP, the Horizon proposes a hardware back-off scheme for failed synchronized loads and stores. The purpose of the delayed retry is to minimize the network traffic. The Horizon also provides a user-level trap mechanism, substantially reducing the cost of traps.

Tera. The Tera architecture inherits much of its design and philosophy from the HEP and Horizon. As the next sections will explain, however, the Tera is customized to better facilitate heterogeneous parallelism. For example, the architecture includes a RESERVE operation for assuring success of stream creation and counters for system monitoring. Hardware retry is used in conjunction with data trap bits for two phase synchronization. The CREATE operation has a new interface, and lookahead only applies to memory operations.

The Tera software system, similarly, explicitly addresses the cooperative sharing of the machine's parallel resources. The management of all types of parallelism, particularly at the coarser levels, distinguishes the Tera system from its predecessors.

Alewife. The Alewife architecture [1] of MIT is another recent addition to the MIMD distributed shared-memory multithreaded community. The Alewife design seeks to avoid memory latency, not only to tolerate it.

Alewife includes caches for shared data plus a hardware maintained directory scheme for retaining coherency between them. Effectiveness of the caches depends largely on the locality of program references. Alewife employs coarse grain multithreading, where instructions from a given stream are executed until the stream performs a remote memory request or fails in a synchronization attempt. Stream scheduling is done in software.

As with the previous systems, Alewife provides a full/empty bit on each memory location for synchronization. Loads and stores that (temporarily) fail cause a trap to occur and, potentially, the stream to be switched.

Because Alewife uses coarse grain multithreading, the machine may require fewer program counters than the Tera to keep a processor busy. Because the Tera makes no assumptions about locality, on the other hand, it may be easier to write scalable programs on the Tera. It appears that it is important for the Tera programmer/system to find parallelism in the application for good performance, while it is important for the Alewife programmer/system to find both locality in the application and some parallelism.

TAM. The Threaded Abstract Machine (TAM) [7] is a compiler-based approach to latency tolerance and lightweight synchronization. TAM is an execution model for fine grained parallelism. The model places all synchronization, scheduling, and storage management under compiler control by making these operations explicit in the instruction set. The model is not a hardware design; it instead can be implemented on a variety of conventional (and unconventional) architectures. Although it is difficult to compare models with architectures, we contrast several general features of the TAM with those of the Tera.

The TAM compiler schedules instruction streams. Consequently, the scheduling policies are flexible and can take advantage of information known about the executing program. The Tera runtime system provides the same flexibility with scheduling medium-grained program units. To minimize overhead for fine-grained parallelism, however, streams on the Tera are scheduled by the hardware. At this level, given enough parallelism, simple policies perform adequately.

TAM has another level in its scheduling hierarchy called a "quantum". A quantum represents a set of threads that can be statically co-scheduled on a processor and share a register set. Since the registers are shared, one remote reference does not idle a register set. In contrast, the Tera has only one program thread per register set. However, because lookahead allows up to eight remote references to be unresolved at once, a single remote reference also need not idle a Tera register set.

***T.** *T [14] is a hybrid architecture, adopting characteristics of both parallel von Neumann and dataflow machines. The architecture aims to limit the cost of remote loads and synchronization latency, particularly that due to barriers.

Each node of *T has two asynchronous processors. The synchronization processor directly handles messages corresponding to short activities. A common short activity is that of a join synchronization, or a remote memory request. The processor queues other messages for the second processor: the data processor. The data processor typically fields longer, more compute intensive activities, having been relieved of quick synchronization activities by the first processor. Because the synchronization processor handles all multithreading aspects, the data processor can be a standard RISC processor, leveraging the high speed provided by current RISC processors.

Similar to TAM, the *T model presents a single register set that is shared among the streams running on a processor. This sharing makes streams very lightweight, which allows very fine grain parallelism to be effectively exploited. Similar to a dataflow processor, a stream that performs a remote load or remote synchronization leaves no state behind on the issuing processor. The request contains a continuation to restart the stream when the response is returned. Thus, remote operations do not consume register file locations, allowing other streams to take advantage of more registers. On the other hand, remote operations can be more costly, since a stream with several live values may have to store them before issuing the load and restore them upon restarting the continuation (after the operation completes).

Summary. There are a number of approaches to tolerating latency with multithreading. Each approach has its merits. We believe a principal strength of the Tera system is its ability to exploit heterogeneous parallelism, including parallelism from simultaneous jobs. Although other systems exploit parallelism at several levels — generally the fine and/or medium grain levels — Tera appears unique in its goal to exploit parallelism at all levels.

3 Tera Architecture Overview

The Tera architecture implements a physically shared-memory multiprocessor with multi-stream processors and interleaved memory units interconnected by a packet-switched interconnection network [3]. The network looks like a pipeline from the processor's perspective and its bandwidth is sufficient to allow arbitrary data placement relative to the processors provided the resulting latency can be tolerated. The network provides a bisection bandwidth which allows a request and a response for each processor to cross the bisection plane each clock tick.

Each processor of the Tera has support for 16 protection domains and 128 streams. A protection domain implements a memory map with an independent set of registers holding stream resource limits and accounting information. Consequently, each processor can be executing 16 distinct applications in parallel. Although fewer than 16 applications are usually necessary to attain peak utilization of the processor, the extra domains allow the operating system flexibility in mapping applications to processors.

Streams each have their own register set and are hardware scheduled. 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 completes, the stream to which it belongs thereby becomes ready to execute the next instruction. Provided there are enough instruction streams in the processor so that the average instruction latency is filled with instructions from other streams, the processor is fully utilized. Similar to the provision of protection domains, the hardware provision of 128 streams per processor is more than is necessary to keep the processor busy at any one time. The extra streams enable running jobs to fully utilize the processor while other jobs are swapped in and out independently; they enable a processor to remain saturated during periods of higher latency (for example, due to synchronization waits and memory contention).

Tera associates four state bits 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 (the 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.

Tera supports three modes of interacting with the full/empty bit, where again, the mode is selected by the pointer.

- (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 cell is full; leave the bit full.
- (Synchronized) Read only when the word is full and set the bit to empty. Write only when the word is empty and set the bit to full.

When a memory access fails 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.

Other features of the Tera architecture, including its wide instructions, support for process creation, counters, and trap mechanisms, will be defined as they are encountered in the following sections.

4 Very Fine-grained Parallelism

The wide instructions and multiple pipelines of the Tera architecture support parallelism at a very fine grain. Because it is tedious to program, reasonably easy to detect, and present in even the most sequential application, the Tera compiler is solely responsible for detecting and scheduling this level of an application's parallelism.

Tera instructions are packed with three operations: Memory, Arithmetic, and Control (MAC). A typical M-op is a LOAD, a typical A-op is a FLOAT_ADD_MUL, and a typical C-op is a JUMP. The C-op slot is flexible. It can also house simple integer and floating point arithmetic operations which allows addressing calculations to be performed in parallel with floating point computation. A three operation Tera instruction can issue in each clock tick.

The hardware guarantees that the results of A and C operations will be available (or bypassed) when needed by a succeeding instruction. However, M-op dependences are handled through a a 3-bit dependence lookahead field in the instruction. The hardware allows each stream up to eight simultaneously outstanding memory operations where the value of the lookahead field controls this degree of parallelism. The major benefit of lookahead is that it allows memory latency to be tolerated within a stream. It also allows instructions from the same stream to be co-resident in the pipeline, handles register dependence on loads, and handles any sort of dependence between memory operations. Making dependence explicit in the instruction allows the hardware to plan ahead, scheduling streams with satisfied dependence constraints rather than finding out too late and stalling the pipeline. The lookahead value is set by the compiler and can differ for each instance of a memory operation.

The Tera hardware also supports very fine-grain parallelism in the functional unit pipelines. Whereas conventional pipelines are filled with instructions from one instruction stream, the Tera pipelines may hold instructions from several streams. As seen with the HEP, having multiple streams filling the pipelines often increases their utilization.

5 Fine-grained Parallelism

The Tera compiler and hardware work together to automatically detect and exploit fine-grained parallelism. Before we describe the compiler's actions, it is useful to define the critical hardware features that it employs.

Tera provides operations that allocate, activate and deallocate streams on a single processor: RESERVE, CREATE, and QUIT. The RESERVE instructions are novel, and were introduced to assist with the automatic parallelization of programs. The instructions reserve the right to issue CREATE instructions, which activate idle streams and assign them program counters and data environments. The QUIT instruction returns a stream to the idle state.

Significant amounts of parallelism can be automatically extracted from existing scientific programs using current compiler technology. The dominant form of parallelism exploited by compilers is "loop level parallelism" — parallelism obtained by executing separate iterations of a loop concurrently. Additional parallelism is found by detecting when separate blocks of code can be executed concurrently. Compilers are also capable of inserting explicit synchronization to achieve a form of data pipelining[12].

The optimal mapping of this parallelism onto the target architecture is often dependent on parameters not known at compile-time. In particular, for a given number of processors, the number of iterations and the execution times of various tasks influences how parallelism is implemented, *e.g.* how loops are scheduled and what parallelism is deemed "not useful" and executed serially.

On multithreaded architectures in which the physical processor is shared with other parallel activities from the same and other jobs, the static mapping problem is further complicated by uncertain dynamic resources. Consider an isolated parallel loop:

```
DO K = 1,N
   X(K)= Q + Y(K)*(R*ZX(K+10) + T*ZX(K+11))
ENDDO
```

The compiler might implement this loop so that $s-1$ additional streams are "created" and each stream performs approximately N/s iterations. However, if there are not $s-1$ idle streams at the moment this loop begins execution then this implementation will not execute all iterations of the loop.

A solution is to have the compiler generate code that does not require additional streams but can exploit them if available. This approach has been used on several of the mini-super computers such as Alliant and Convex, where it is considered important that executables be independent of machine configuration. Our motivation is to allow very dynamic allocation of resources from one parallel activity to another, even across jobs, so that the system is very responsive to changes in available parallelism.

One strategy of the compiler for a simple parallel loop is to use up to 30 streams[1] and divide the loop evenly among however many streams are acquired. If the processor is busy and no additional streams are available, then the loop executes serially with $s = 1$ streams after only a few instructions of overhead:

```
    reserve_upto s r0 29
    s += 1
    k0 = ceiling(n/s)
    store CHUNK, k0
    store DONE$, s
    k = k0-1
    while (--s > 0) {
        create l1 k=k sp=sp n=n
        k += k0 }
    k = -1
l1:  load k0, CHUNK
    n = min(k0+k,n)
    while (++k < n) {
        // usual scalar optimizations apply...
        X(k)= Q + Y(k)*(R*ZX(k+10) + T*ZX(k+11))
    }
    d = DONE$--
    if(d > 1) quit
```

The Tera instruction set includes the A-op RESERVE_UPTO t u st where t and u are registers and st is an immediate constant. This instruction attempts to reserve as many streams as are available, up to $u + st$, for use by the current protection domain. The number of streams successfully reserved is stored into register t. Register `r0` is always 0.

[1]As few as 10 or as many as 80 streams could be needed to saturate the processor, depending on the program.

The CREATE instruction is parameterized by a pc-relative jump target and three registers passed to the new stream. In this example, the CREATE instruction is simply skipped if the RESERVE fails to allocate any streams.

Variables in upper case are in memory and variables with a **$** indicate uses of the full/empty bit to provide ultra-lightweight synchronization. Each read is a read-when-full-set-empty, and each write is a write-when-empty-set-full. The shared variable **DONE$** holds the number of streams involved in the loop. Each stream decrements the variable as it completes its share of the loop and each stream except the last QUITs. Note that as soon as a stream quits one such loop, it is ready to be reserved and applied to another loop possibly in a different job. Several independent instances of such loops should present a fairly uniform demand on the processor.

The hardware provides two counters for each job loaded in a processor. One holds the current number of active streams in the job and the other holds the number of streams reserved for that job. When a CREATE instruction is issued, a fault occurs if the reserved count is not larger than the current count. Otherwise, the current count is incremented. The RESERVE instructions are used to increase the value of the reserved counter to allow streams to be CREATEd. The QUIT instruction decrements both the current counter and the reserved counter. The code fragment above depends on following the software convention that a create is only issued after a successful reserve. Violating this convention may cause incorrect behavior in the errant job but will not affect other jobs.

For many forms of parallelism this approach is very effective, having low overhead in both time and code-space. Some forms of parallelism are best implemented with a specific number of streams. In addition to the RESERVE_UPTO operation, we also support the operation RESERVE t u st that attempts to reserve $u + st$ streams but does not reserve any unless $u + st$ are available. It also has a _TEST variant that sets a condition code indicating whether streams were reserved. The compiler must generate code to handle the case when the RESERVE fails. One approach is to generate two versions of the code, a parallel one and a serial one and use the condition code to select at run time which version to run. Since this may be costly in terms of code size, an alternative is to have a more expensive but reliable software allocation mechanism to fall back on. It is expected that the reserve rarely will fail and so the expected cost of acquiring streams is still low.

6 Medium-Grained Parallelism

Tera's medium grain parallelism generally has a granularity of greater than 100 instructions and can be specified by the user, discovered by the compiler, or a combination of both.

A general-purpose parallel language should encourage the programmer to express the parallelism of an application wherever it is natural. It should hide from the programmer, as much as possible, configuration details of the hardware and implementation details of the system. Together, these goals imply that a parallel language should:

- Support an unbounded number of parallel activities — so that an application's parallelism need not be dependent on a certain number of resources and can be discovered and generated incrementally;
- Achieve efficient processor utilization through automatic load balancing — so that the programmer need not be concerned with scheduling issues;
- Support lightweight synchronization — so that communication between activities can be used when needed, and used without penalties that may influence the algorithm or decrease its parallelism.

To illustrate how the Tera system addresses these properties, the remainder of this section describes several straightforward language extensions and their implementation on our hardware.

6.1 Expressing Parallelism

Explicit parallel programming is fostered through future variables and future statements. A future variable describes a location that will eventually receive the result of a computation. In our extended C,[2] these variables may have any primitive type (*e.g.* char, int, float, or pointer) and are identified with the `future` type qualifier [6]. A `future` statement is used to create a new parallel activity and direct its result to a future variable.

When a future statement starts, the future variable is marked as unavailable (the full/empty bit associated with the memory cell is set to empty) and any subsequent reads of the unavailable value block. When the future completes, the appropriate value is written to the future variable and the location is marked available (the full/empty bit is toggled to full). Any parallel activities that were blocked waiting for the result may now proceed.

Future variables provide a powerful form of synchronization for "software pipelining" but do not work well for mutual exclusion and bounded buffer producer/consumer situations. To address this need we have added a `sync` type qualifier which also can be combined with any primitive type. These "synchronized" variables provide direct access to the hardware full/empty bit. A read from an empty synchronized variable blocks until the variable becomes full and then resets the variable's state to empty. Similarly a write to a full synchronized variable blocks until the variable becomes empty and then resets the state to full. Synchronized variables provide a very powerful base on which more complex structures can be created.

6.2 Implementing Parallelism

The combination of futures and synchronized variables presents a virtual machine that satisfies the aforementioned language goals. It remains to describe an implementation that accomplishes load balancing and lightweight synchronization.

Since parallel activities are created dynamically and execute for an unpredictable amount of time, a static scheduling policy is impractical. Instead, we use a dynamic self-scheduling approach and rely on the fact that the decision of which parallel activity to execute next is not crucial. If a parallel activity blocks shortly after it is started due to synchronization constraints, an efficient synchronization strategy can get a new parallel activity running.

We call each medium-grain parallel activity a *chore*. Again, chores can be created either by the compiler or through future statements. On creation, chores are placed in a ready pool, which corresponds to an unordered collection of ready-to-execute continuations.

The runtime environment uses a work-pool style strategy for executing chores from the ready pool. When a job starts executing, the runtime reserves some number of instruction streams, called *virtual processors*, to work on the job. Virtual processors repeatedly select and run chores from the ready pool. Executing a chore may result in the allocation of additional instruction streams for finer-grained parallelism and in the generation of new chores.

Changing the Parallel Resource Allocation. The Tera runtime aims to react to the dynamic needs of an application by adjusting its resources to these needs.

The runtime can dynamically increase and decrease the number of virtual processors it employs. This func-

[2] Though the discussion of futures uses a variant of C, we have transliterated our language extensions to FORTRAN. In addition, the parallel activity scheduling described here is independent of language construct and could be used to support Ada tasking or concurrent threads systems such as PRESTO[5] or Linda[2].

tionality is useful as the average size of the ready pool can vary significantly over the lifetime of the program. Reasonably, a large ready pool merits more virtual processors than a small ready pool for its timely execution. Resources acquired for a period of high parallelism, in contrast, should be released during a period of low parallelism to limit their inefficient use. The runtime monitors the ratio of the number of chores waiting to run relative to the number of virtual processors and acts accordingly.

On any given processor, additional streams (for use as virtual processors) can be bound by the use of RESERVE and CREATE hardware instructions. Streams can be released with the QUIT instruction. Until the runtime reaches a stream limit imposed by the operating system, it can thus grow and shrink its stream resources by communicating directly with the hardware. The limit ensures that the operating system controls large-grain fairness in resource allocation between jobs.

Interaction with the operating system is required when the runtime seeks to add new streams running on a different physical processor. A group of streams cooperating on a processor is called a team. While the runtime can grow (shrink) a team independently of the operating system (through RESERVE and CREATE instructions), it must interact with the operating system to acquire (release) new teams. This is because team creation requires resources such as protection domains that must be globally controlled and initialized. The runtime interaction takes the form of an operating system call that, like RESERVE, must be checked for success. More on the nature of the call (team allocation) and the counters that the operating system uses to decide whether the additional resources should be granted is detailed in Section 7.

6.3 Efficient Synchronization

A chore may block because it is waiting for a future statement to complete or because it is trying to access a synchronized variable which is in the wrong state. Since there may be many chores executing and accessing data in a variety of ways it is necessary to support efficient synchronization and blocking. Our goals for synchronization are the following:

- If the synchronization will soon succeed, then a busy–wait should be used so that the cost of synchronizing will be little more than the cost of a normal memory reference.
- If the synchronization will not succeed for a long period of time, then a heavier weight, non busy-waiting strategy is desirable.

Synchronization is accomplished in the Tera through the use of the full/empty bit associated with each word of memory. If the bit is in the desired state, the synchronization takes place immediately. If not, the hardware and software take action using a novel retry-then-trap scheme.

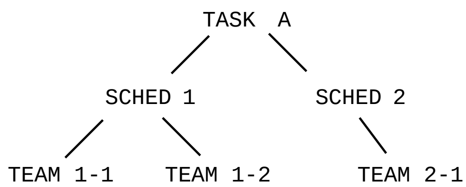
Our scheme is tuned for optimistic synchronization – the assumption that most synchronization attempts will either succeed immediately or be waiting a short while. When a synchronizing memory access is made and the full/empty bit is in the wrong state, the request returns to the processor with a failed status. The hardware places the request in a retry queue and automatically retries it. Retry requests are interleaved with new memory requests at a fixed rate. Interleaving requests avoids flooding the communication network with requests that recently failed but at the same time enables a quick retry. By retrying automatically in hardware, no additional instruction issues are required; the processor can continue issuing instructions at the normal rate of one per tick.

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 reaches the protection domain's limit, the stream traps using the lightweight trap handling facility. The retry limit value is set to balance the cost of saving and later restoring the state of the blocked chore which is necessary to implement the blocking synchronization.

When a stream s traps because of a retry failure, the program counter and register state of the chore being executed by the stream are saved. The values are placed on a list associated with the synchronization address and the memory cell has one of its data trap bits set. When a second stream t attempts to access the memory location, it immediately takes a trap. The trap handler finds the continuation for the chore that failed to synchronize, often placing it back in the ready (unblocked) pool for another virtual processor to execute. This implementation is similar to that adopted for the split-phase transactions of I-Structure storage [4]. The use of the data trap bit avoids the need to either poll the memory location or to explicitly program a check on each memory reference that could unblock another stream.

7 Coarse-Grain Parallelism

The operating system schedules coarse grain parallelism resources — *tasks*, *scheds*, and *teams* — leaving the scheduling of fine grain parallelism resources — *streams* — fully under the control of the compiler and runtime. A task is the system-wide unit of resource assignment. It provides an execution environment for running a program. Each task consists of one or more scheds. A sched is the operating system's smallest individually schedulable unit. The distinction between task and sched helps separate the function of resource allocation from that of scheduling. Each sched includes one or more teams, where a team is a group of streams that execute on the same processor. (More specifically, a team is a group of streams that execute within a single protection domain.) The following diagram illustrates the Tera system resource model.



Task A contains two scheds, Sched 1 and Sched 2. These are units of A that can be scheduled separately. Sched 1 contains two teams, Team 1-1 and Team 1-2. The teams may be on the same processor but typically are on different processors. Each team owns a dynamically sized set of streams of its processor. Sched 2 contains one team, Team 2-1. Team 2-1 may be on the same or a different processor than Teams 1-1 and/or 1-2.

Tera exploits coarse-grain parallelism by concurrently executing independent tasks (applications). We argued in the introduction that parallelism within a single task can rise and fall. Given this dynamic behavior, it is desirable to share physical processors between different tasks so that processor resources can migrate rapidly from tasks with decreasing parallelism to tasks with increasing parallelism. To foster sharing between tasks — without costly software context switching — Tera provides 16 protection domains per processor. This allows up to 15 user tasks to run in parallel, with the last domain reserved for system activities. Resources of a task are expanded and decreased using the fine-grain and coarser-grain techniques described in earlier sections.

Scheduling of the protection domains amongst the set of ready applications is handled by two schedulers, a big sched scheduler (pb-scheduler) and a small sched scheduler (ps-scheduler). The schedulers correspond to the operating system's characterization of applications as big sched tasks or little sched tasks. *Big sched* tasks are long lived, resource consuming, parallel applications; for example, jobs submitted via a remote job entry batch system. *Small sched* applications are short lived, use fewer resources, are not very parallel, and often demand quick turnaround time; for example, shell commands. A processor's protection domains are shared between the pb-scheduler and the ps-scheduler. Very generally, the pb-scheduler places as many teams from big sched tasks on a processor as that processor can support. The ps-scheduler then absorbs the slack; it uses the remaining protection domains for small sched jobs. Because small sched jobs typically have one (or few) threads, their stream utilization is more predictable and they serve as a guaranteed load for the processor.

Except for swapping scheds, the Tera operating system is unobtrusive. It will not add or remove resources from a running task unless given permission by the task's runtime. Typically the permission is in the form of a request from the runtime to add (or remove) a team to the current task. The runtime can ask for a team to be added immediately, or for a team to be added sometime in the near future. It can ask to be swapped out if the team is not granted, as it cannot make progress otherwise. To assist the operating system in decisions about granting requests (and swapping slow, contention-filled, or otherwise cumbersome scheds), the system monitors the parallelism of active scheds. The Tera architecture provides several accounting and performance monitors for this purpose. The monitors permit the operating system to track stream usage and processor usage within and across tasks.

The simplest performance counter is a user accessible time-of-day clock, which is incremented every clock cycle. Two per-protection-domain counters are used to account for resource usage. The *issue* counter tracks the number of instructions issued by streams in a protection domain. By sampling this over time and scaling by the time delta, the fraction of the processor load due to each team is obtained. The *stream* counter can be similarly used to obtain the average number of streams used by each team on a processor; it is advanced every tick by the number of active streams associated with the protection domain.

Two additional per-processor counters are used to measure processor utilization. A *phantom* counter counts the number of issue slots in which no stream was ready to issue due to long memory requests or pipeline hazards. One minus the ratio of this value divided by a time quantum provides a precise measure of processor utilization. The *ready* counter is incremented every tick by the number of streams which are ready to issue. This value divided by the time quantum provides a measure of average excess parallelism.

The operating system uses the counters to influence dynamic changes in resource allocations. For instance, when the operating system determines that a processor is underutilized, it may add or move a team to that processor for some currently active task in the system. Additionally, the operating system uses the counters to influence placement of teams when new jobs are initialized or resumed after a swap.

8 Summary

The Tera system is an integrated solution to exploiting heterogeneous parallelism. The multithreaded architecture includes fast stream switching, memory and functional unit pipelines, memory synchronization bits, performance counters, and user access to stream resources. Using this platform, the compiler detects and statically schedules parallelism at the very fine grain and fine grain levels: parallelism ranging from packing operations in an instruction to exploiting parallel loops. Similarly, the runtime supports and augments parallelism generated by the compiler with coarser grain activities: activities that are programmed explicitly, or are less uniform. The operating system concurrently shares the machine's resources among multiple tasks. Sharing resources fosters high machine utilization, as the temporary shortage of parallelism in one task can be compensated by the parallelism of another.

A key feature of the Tera system is its system integration. Examples include: the ability of the compiler and runtime to directly reserve and release stream resources; the runtime's requests to the operating system to increase or decrease its number of virtual processors; and the operating system's monitoring of processor utilization, which places (or moves) applications onto processors that are lightly loaded. We believe that by communicating with other components of the system, each component can better do its particular job.

The Tera system is now in its early development. A preliminary compiler, runtime system, and related tools exist, along with portions of the operating system. A growing set of applications are running on the Tera hardware simulator. Multithreaded processors — having the ability to mask the latency of one stream with the activity of another — even more than traditional single threaded processors, appear to give us the flexibility needed to effectively support heterogeneous parallelism.

References

[1] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Maa, and D. Nussbaum. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. In *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991.

[2] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8), August 1986.

[3] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *1990 International Conference on Supercomputing*, June 1990.

[4] Arvind and R.A. Iannucci. A critique of multiprocessing von Neumann style. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, June 1983.

[5] B. Bershad, E. Lazowska, H. Levy, and D. Wagner. An open environment for building parallel programming systems. In *ACM/SIGPLAN PPEALS 1988*, New Haven, Conn., September 1988.

[6] D. Callahan and B. Smith. A future-based language for a general-purpose highly-parallel computer. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*. MIT Press, 1990.

[7] D. Culler, A. Sah, K. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine. In *Proceedings of Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991.

[8] D. E. Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii, May 1988.

[9] R. Halstead Jr. and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii, May 1988.

[10] J.T. Kuehn and B.J. Smith. The Horizon supercomputing system: Architecture and software. In *Proceedings of Supercomputing 1988*, Orlando, Florida, November 1988.

[11] M. Kumar. Effect of storage allocation / reclamation methods on parallelism and storage requirements. In *Proceedings of the 14th International Symposium on Computer Architecture*, May 1987.

[12] S. P. Midkiff and D. A. Padua. Compiler generated synchronization for DO loops. In *Proceedings of the 1986 International Conference on Parallel Processing*, August 1986.

[13] E.F. Miller Jr. A multiple stream registerless shared-resource processor. *IEEE Transactions on Computers*, C-23(3), March 1974.

[14] R. Nikhil, G. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. Technical Report Computation Structures Group 325-1, MIT Laboratory for Computer Science, November 1991.

[15] B. J. Smith. A pipelined, shared resource MIMD computer. In *Proceedings of the 1978 International Conference on Parallel Processing*, 1978.