Scheduling on the Tera MTA*

Gail Alverson Simon Kahan Richard Korry Cathy McCann Burton Smith

Tera Computer Company
Seattle, Washington USA

Abstract

This paper describes the scheduling issues specific to the Tera MTA high performance shared memory multithreaded multiprocessor and presents solutions to classic scheduling problems. The Tera MTA exploits parallelism at all levels, from fine-grained instruction-level parallelism within a single processor to parallel programming across processors, to multiprogramming among several applications simultaneously. Consequently, scheduling of resources occurs at many levels, and managing these resources poses unique and challenging scheduling concerns. This paper outlines the scheduling algorithms of the user level runtime and operating system and describes the issues relevant to each. Many of the issues encountered and solutions proposed are novel, given the multithreaded, multiprogrammed nature of our architecture. In particular, we present an algorithm for swapping a set of tasks to and from memory that achieves minimal overhead, largely independent of the order in which tasks are swapped.

1 Introduction

Scheduling on the MTA involves managing a wide range of resources. The MTA supports a multi-user system, where many parallel jobs execute concurrently. Each multi-threaded processor can support up to 16 jobs simultaneously that compete for the processor's instruction streams. In addition, memory must be scheduled because of the potentially broad spectrum of job memory requirements.

Most parallel systems are composed of two levels of scheduling [16, 24, 23, 9]. A kernel or low level scheduler divides the system resources among competing jobs, and a user level scheduler divides the parallel work among the job's available processing resources. Like other systems, the Tera MTA supports this division. The operating system scheduler manages processing and memory resources at a gross level. It determines which jobs are loaded in memory and how those jobs are scheduled on the processors. Once loaded onto a processor, each job competes with other jobs on the same processor for instruction streams, without intervention from the operating system. The competition is guided by runtime systems, one per job, that request and release streams in response to their job's parallel workload. In addition to managing resource acquisition, the runtime works in concert with the compiler to schedule both automatically-generated and user-generated parallelism within a job.

The workload anticipated for the MTA is varied. We expect most sites to have large, parallel jobs intermixed with interactive work. The operating system scheduler must excel in this environment. Large parallel jobs require high throughput, while short, interactive jobs require

*This research was supported by the United States Advanced Research Projects Agency Information Science and Technology Office ARPA Order No. 6512/2-4; Program Code No. OT10 issued by ARPA/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 ARPA or the U. S. Government.

quick response. Also, the scheduler must prevent starvation and ensure some measure of fairness among jobs of similar characteristics; to facilitate this, we provide a mechanism for job differentiation.

This paper describes the scheduling issues specific to our unique architecture. We outline both the scheduling policies of the operating system and the user level runtime. Section 2 familiarizes the reader with the Tera MTA architecture. Section 3 describes the programming environment made available to the user. Section 4 describes the scheduling decisions made by the user level runtime system on behalf of user programs. Section 5 describes the overall goals of operating system schedulers that manage the allocation of processors and memory resources. Section 6 describes the issues and policies involved in memory scheduling, and Section 7 discusses the processor scheduler. Conclusions are provided in section 8.

2 Tera MTA System Architecture

The Tera MTA architecture[1] implements a physically shared-memory multiprocessor with multi-stream processors and interleaved memory units interconnected by a packet-switched interconnection network. The network provides a bisection bandwidth that supports a request and a response from each processor to a random memory location in each clock tick.

Each processor of the Tera has support for 16 protection domains and 128 instruction streams. Each 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 likely necessary to attain peak utilization of the processor, the available domains allow the operating system flexibility in mapping applications to processors. While two or three parallel jobs may sustain a high average parallelism executing on a processor, so may a mixture of parallel and sequential jobs.

Each stream has its own register set and is 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. 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 many 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's memory is distributed and shared. All memory units are addressable by each processor.

Several architectural features are especially useful for scheduling. A stream limit counter is associated with each protection domain. It allows the operating system to pre-impose an upper limit on the stream resources acquired by each job; the limit is enforced by the hardware, so jobs can self-schedule their stream acquisitions without incurring operating system overhead. Tera provides three user level instructions to acquire and release instruction streams on the processor of the invoking stream: `stream_reserve`, `stream_create`, and `stream_quit`. The novel `reserve` instruction was introduced to assist with the automatic parallelization of programs. The instruction reserves the right to issue `create` instructions, which activate idle streams and assign them program counters and data environments. By first issuing a `reserve` instruction, compiler generated scheduling code is able to compute a division of parallel work appropriate for the exact number of streams available or requested — before those streams begin consuming processor resources and without synchronization overheads. The `quit` instruction returns a stream to the idle state and decreases the reservation count.

3 Programming Environment

Programs for the Tera are written in Tera C, C++ and Fortran. Parallelism can be expressed explicitly and implicitly.

Explicit parallel programming is assisted through the use of *future* variables and future statements. A future variable describes a location that will eventually receive the result of a computation. 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, causing any subsequent reads of the unavailable value to block. When the future completes, the appropriate value is written to the future variable and the location is marked available. Any parallel activities that are blocked waiting for the future may now proceed.[1] The user runtime is responsible for scheduling and executing future bodies [Section 4].

Implicit parallelism is identified by Tera's parallelizing compiler. The majority of parallelism exploited is loop level parallelism: parallelism obtained by executing separate iterations of a loop concurrently. Other forms exploited include block level parallelism, obtained by executing separate blocks of code concurrently, and data pipelining, achieved only when explicit synchronization is included.

The compiler schedules resources for much of the parallelism it generates. It generates user level instructions, `stream_reserve` and `stream_create`, to allocate hardware streams. It then schedules the work among the streams statically or using a self-scheduling approach. The compiler generates instructions to release the streams, `stream_quit`, when the work is complete. For example, a strategy for a simple parallel loop is to request up to 30 streams[2] 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 one stream after only a few instructions of overhead.

When the compiler exposes a large amount of parallelism it may acquire stream resources across a set of protection domains on distinct processors. To do so it generate calls to the user runtime (which in turn may call the operating system) to create the first stream in each of the protection domains. Compiler-generated instructions then acquire and release streams in each domain based on the parallelism exposed.

In summary, the user level runtime schedules resources in response to explicit parallelism; compiler generated code adaptively schedules resources wherever implicit parallelism is detected.

4 User Level Runtime System

The runtime represents a user's program in much the same way a lawyer represents her client: it is responsible for obtaining the best response time for the user's program while ensuring that (1) the program abides by rules imposed to ensure equitable access to processing resources and (2) the program responds to anomalies (signals, traps) without endangering its own correctness. Here, we describe features of the runtime pertaining only to the first constraint. That is, we describe the structure and policies of the runtime that support fast response to changes in parallelism without significantly reducing overall machine throughput.

4.1 Software Architecture

The software architecture of the runtime consists of a hierarchy of objects animating the processing behavior of the Tera. Many objects parallel those of the operating system [Section 5].

[1]The Tera MTA has full/empty bits on every word of memory, which support the efficient implementation of futures.

[2]As few as 20 or as many as 80 streams could be needed to saturate the processor, depending on the program.

A user program is represented as a *task*. A task's computing resources is a collection of *teams*, each of which occupies a protection domain. The operating system promises that teams of the same task are on distinct processors, as long as the number of teams of the program is less than the number of processors of the machine. Practically speaking, it serves no purpose to put two teams of the same program on the same processor: the stream limit imposed by the operating system is large enough to enable a parallel team to more than saturate the processor.

Each team includes a set of *virtual processors* (vps), each of which is a process bound to a hardware stream. Note the distinction between a stream and a vp: every vp has a dedicated stream, but each stream does not necessarily have an associated vp. A stream can belong to a team without being a vp: compiler generated parallelism makes use of such streams. Or a stream may simply be unused. Unless specified differently at startup, the task initially starts with one team and one vp. The runtime is responsible for acquiring more resources if there is parallelism in the program and resources are available.

The collection of tasks active on the machine at any given time is under the control of the operating system. The runtime systems representing each of these tasks compete with one another for resources.

4.2 Scheduling Work

Work appears in two types of queues in the runtime. New futures are spawned into the *ready pool* of the task. The ready pool is a parallel FIFO based on `fetch&add` [8]. Futures that had blocked and are now ready to run again are placed in the *unblocked pool* of the team on which they blocked. Distributed unblocked pools serve as a convenient means to direct special work, such as a command to exit, to a particular team.

The runtime uses a self-scheduling strategy for executing work from the ready and unblocked pools. Vps from every team repeatedly select and run work from the pools. A vp's unblocked pool is given precedence because we would like to complete old work before starting new work. Executing work can result in the allocation of additional instruction streams for compiler generated, finer grain parallelism; in the creation of new work; and in the blocking of work. The vp executing work that creates new work or unblocks old work is responsible for placing the work on the ready or on the unblocked pool. If the work blocks, the vp searches immediately for other work on the unblocked and ready pools; that is, blocking does not lead to busy-waiting.

Some programs, especially those using divide-and-conquer algorithms, may do better with an ordering different from FIFO. Tera provides the user with a language extension, `touch` (-future), to assist with the effective execution of those programs. Wagner and Calder[22], outline a more general version of touch, called leapfrogging. When touch is applied to a future variable, the vp stops executing its current work and executes instead the work associated with the future variable, unless it has already started. Thus, the vp executes the work on which *its* work is waiting. The policy saves blocking overhead at the expense of a few extra instructions of overhead when the work is already started and touch is not used. The use of touch makes the default FIFO ordering of work more LIFO in character.

In practice, the user need not explicitly touch futures in divide-and-conquer programs to achieve the efficiency provided by the mechanism. The compiler is able to infer divide-and-conquer recursive structure when it discovers the storage class of a future variable is automatic; it then generates code to perform the touch. The result is breadth-first scheduling of subproblems until vp creation lags subproblem creation; at that point the scheduling becomes depth-first. This dynamic schedule is both efficient regarding overheads and reactive to changes in the vp workforce.

4.3 Scheduling Stream Resources

The runtime is responsible for dynamically increasing and decreasing the number of vps it employs to execute the user program. This functionality is important because the average size of the ready pool can vary significantly over the lifetime of the program. Reasonably, a large ready pool merits more vps than a small ready pool for its timely execution, and vps acquired for periods of high parallelism should be released during periods of low parallelism to improve efficiency.

The goals of the virtual processor strategy of the runtime mirror those of the runtime itself: an increase in the number of vps is attempted whenever it might reduce response time of the program, subject to the constraint that overall throughput of the machine should not be significantly reduced. Were acquisition and release of streams instantaneous, a policy of creating a new vp for each and every future would ensure minimum response time and retirement of vps whenever pools emptied would minimize impact on throughput. But there are overheads, and the runtime cannot foresee how pool sizes will change and how much computation any piece of work will demand. Nor does it have information regarding the behavior of the other programs with which it is competing. So there is an inherent tradeoff between minimizing response time, through zealous resource acquisition with delayed release, and maximizing throughput — assuming resources lag demand — through conservative acquisition with immediate release.

Growth Policy. Because we wish to minimize the overhead involved when a vp adds work to the ready pool, the growth policy is implemented not by those vps but by a daemon mechanism that periodically assesses growth based on idleness and work available. The daemon creates twice the number of vps previously created whenever (1) there is *any* work to be done, and (2) no vps were idle since the last acquisition. For example, if there is initially one vp, after one period another will be created; after two periods, assuming both vps were kept busy, two more will be created; after three periods, supposing one of the four vps found nothing to do, even for a moment, no vps will be created; and after four periods, assuming all four vps were this time kept busy, one more vp will be created. If an acquisition request fails, in whole or in part, it is repeated after the next period has elapsed.

Our intention is to achieve a compromise in which the total overhead is at most a small constant fraction of the work intrinsic to the program, and the response time is similarly bounded from above by at most a small constant factor multiplied by the minimal response time, were all streams requested actually available. (For response time, an additive lower order term proportional to the logarithm of the maximum parallelism persists in our scheme, due to the delay of exponential growth versus instantaneous growth.) Such worst-case bounds should be achieved only when the parallelism is highly erratic in just the wrong way: we expect typical performance to be much closer to optimal.

Retirement Policy. Just as important as the acquisition policy is the release policy: idle vps are released according to a schedule that inverts the acquisition schedule. That is, were a large number of vps to become idle at once, one would disappear after one period, two more after a second period, and so on unless work appeared to interrupt the process. This is necessary to avoid the anomalous behavior that would result were there to be brief sequential periods separating slightly longer but highly parallel periods at just the right frequency to cause repeated rapid growth and collapse. Were idle streams released after a constant period of time independent of their number, the constant factor in the response time bound becomes logarithmic in the maximum parallelism.

Because idle vps on the Tera consume as much processing power as vps performing memory-bound computations, our implementation never really allows more than one vp per team to be idle concurrently: within a period, additional idle vps are killed, but remembered, and replaced the instant work appears. Creation and retirement within a team is through user level instructions, and so is fast enough to justify such a reactionary policy. Typically, this should make our upper bound for total work particularly pessimistic. A drawback is that streams, once released, may be acquired by competing teams and thus be unavailable when work reappears: when this happens, our bound for response time becomes arbitrarily optimistic. An alternative is to exploit the afore unmentioned `stream_quit_preserve` instruction which causes a stream to stop executing but does not release it.

Team Growth and Retirement. As mentioned previously, the runtime can independently acquire more stream resources — up to a limit imposed by the operating system — on any team belonging to its task. Interaction with the operating system is required, however, when the runtime seeks to add new streams running in a different protection domain (on a different processor). This is because team creation requires resources other than streams: resources such as text memory and protection domains must be globally controlled and initialized. The runtime interaction takes the form of an operating system call whose return value, like `stream_reserve`, must be checked for degree of success. Parameters to the call include the number of teams desired, a start-up function for each team's first vp (the operating system creates only the first stream on each team), and whether or not the teams are to be added before continuing execution (swapping the job if the teams are not available) or whenever they become available.

The runtime must decide when it is profitable to get a new team instead of creating more vps on existing teams. It currently bases its decision on the processor utilization of one of the teams of the task (extrapolating that utilization to all processors of the whole task). If the utilization is above some threshold, the runtime requests a new team.

The primary mechanism for team retirement is that a vp discovers it is the only vp on its team and that there is no work to do; it then waits for a period of time before relinquishing the team to the operating system and retiring itself.

If it improves performance, we may coalesce small teams.

5 Processor and Memory Scheduling Overview

The operating system allocates the processing and memory resources of the system among tasks competing for these resources. The memory scheduler determines which subset of the ready tasks to load into memory. The processor scheduler then determines from the set of in-memory tasks, which tasks to load onto available protection domains.

Each task is composed of one or more teams, and each team executes within a single protection domain. Generally, each team within a task executes on a separate processor unless the number of processors available is less than the number of teams, due to hardware failure. In order to execute a task, its data and program address space must be loaded into memory. Once loaded in memory, its teams must be loaded into protection domains on separate processors.

There are two types of swapping that occur, swapping between the I/O system and memory, and context switching of tasks in memory among the available protection domains. We use two letters to denote the direction and type of swapping. When a task is brought into memory, it is *im-swapped*, meaning it is brought in from I/O (i) to memory (m). When a task is swapped out of memory, it is *mi-swapped*. A memory-resident task is *mp-swapped* when it is loaded into a protection domain, and *pm-swapped* when it is unloaded from the protection domain. A task must be im-swapped before it can be mp-swapped, and it must be pm-swapped before it can be mi-swapped.

Workload. The operating system schedulers distinguish between large, parallel, computationally intense workloads and small, interactive jobs. Large memory tasks have the following characteristics: they execute for a long time, can tolerate being swapped out for long periods of time, are usually submitted via a remote job entry batch system such as Network Queueing System (NQS), and, overall, use a lot of resources. Generally, they contain a large amount of parallelism. The higher overhead and general non-interactive nature of large memory tasks points to infrequent processor and memory swaps.

Small tasks have the inverse characteristics: they are short-lived or use resources in bursts (interactive), have small memory requirements, are not very parallel, do not tolerate being swapped out for long, are usually submitted from a command shell, and, overall, do not consume a large amount of resources. Small memory tasks may be swapped in and out of memory more frequently. However, for interactive computing, the user expects an immediate response to a keystroke. These tasks require frequent access to processors to enjoy good interactive response.

Based on our understanding of exploitable parallelism found in most large tasks, it is expected that multi-team (large) tasks impose a far greater demand for streams on processors than single-team (small) tasks. Meanwhile, at sites that allow interactive connections, we expect to see a large number of single team tasks. Each of these single-team tasks requires a tiny fraction of the machine's resources for short periods of time.

Because large memory and highly parallel tasks have very different characteristics from small memory interactive tasks, we propose mechanisms for scheduling memory and processors that differentiate these two classes of workload. Studies of interactive and batch workloads by Ashok and Zahorjan [2] support this differentiation. The memory and processing resources are thus divided between large, batch oriented tasks and short, interactive tasks.

Memory and Processor Schedulers. Two memory schedulers are employed, the MB-scheduler for large (big) memory tasks and the ML-scheduler for small memory (little) tasks. Data memory is statically partitioned at boot time into two parts; one part for each scheduler. There is one MB-scheduler and one ML-scheduler per machine.

The protection domains of each processor are divided among single-team and multi-team tasks. A big job processor scheduler (called the PB-scheduler) schedules multi-team tasks, while a small job scheduler (PL-scheduler) schedules single-team tasks. As multi-team tasks execute on multiple processors, a single PB-scheduler using global knowledge of processor utilization is required per machine. The PB-scheduler co-schedules [16] multi-team tasks on the set of protection domains it controls on all processors. Since single-team tasks execute on a single processor, each processor runs its own copy of the PL-scheduler.

A new task is scheduled by the MB-scheduler if its memory requirement exceeds a site definable value, otherwise it is scheduled by the ML-scheduler. When a task is im-swapped, it is assigned to a processor scheduler. If the task is a multi-team task or the task is a single-team task but its stream reservation counter (which maintains a count of the number of streams reserved by the task when last active), is above a site definable parameter, the task is assigned to the PB-scheduler with the expectation that the task imposes a significant amount of parallelism on the processor. Otherwise, the task is assigned to the PL-scheduler with the shortest queue.

If a task scheduled by the ML-scheduler requests more memory than a site settable threshold, it is handed over to the MB-scheduler when it is next mi-swapped. A task scheduled by the MB-scheduler whose memory requirements drop below the threshold remains under the control of the MB-scheduler. This avoids thrashing between memory schedulers.

Each memory scheduler selects a PL-scheduler for a newly im-swapped single-team task.[3] Using the memory scheduler as a dispatcher provides a mechanism for load balancing and control over which processors receive new work.

All schedulers are event driven and similar in design to the Unix Esched scheduler [19] and its improved variants [20] [7]. These schedulers loop forever processing events as they arrive on their event queue. The scheduler receives an event and creates a kernel privileged stream, known as a kernel daemon, to examine the event and take the appropriate action. The scheduler removes the next event from the queue and continues spawning daemons to process the events until the queue is empty. Events are IPC messages sent using the privileged IPC mechanism. Waiting for IPC messages uses a mechanism that does not consume system resources.

Sections 6 and 7 describe the memory and processor schedulers.

6 Memory Schedulers

Despite its size[4], memory will be a scarce resource (possible **the** scarce resource) on the machine, given the execution of large memory applications. Therefore, good performance requires efficient

[3] Unless the team requires a reservation of a large number of streams.

[4] The Tera MTA has 1-2 GB memory per-processor.

memory utilization. Starvation avoidance is also important, especially for tasks with large memory requirements.

As discussed in the previous section, memory is controlled by two schedulers - the MB-scheduler for tasks with large memory requirements and the ML-scheduler for small interactive tasks with small memory requirements. We assume large memory tasks have long execution times and can tolerate long periods of time of being swapped out. We assume small memory tasks are of a class of interactive jobs that require frequent access to processor resources, and therefore, cannot be swapped out for extended periods of time. Each site designates what portion of memory is controlled by which scheduler. Generally, most memory will be given to the MB-scheduler. Both schedulers execute the same algorithm to schedule the memory within their domains.

The memory scheduler algorithm in general must divide available memory among the set of tasks over time. Thus the scheduling space can be represented in two dimensions, with memory on the x axis and time on the y axis. A *schedule* consists of a partitioning of memory over time among a subset of ready tasks in the system. An *allocation* to task j consists of assigning some $m(j)$ units of memory for $t(j)$ time. A valid schedule ensures that the total memory allocated at any one time does not exceed system capacity. The goals of the memory scheduler are to minimize the total unallocated memory over time and to ensure high task throughput.

Both memory schedulers maintain a *ranking* of tasks within their domain. A ranking or priority of tasks is commonly used in schedulers to provide a way for users to specify the relative importance of task resource allocation decisions and to avoid starvation by changing a task's ranking as it acquires resources. The memory schedulers use the ranking to determine the order in which tasks are considered for scheduling. The manner in which task ranking is defined is discussed in Section 6.2.

A memory schedule defines a set of tasks that will be resident at any one time. For each task, the scheduler determines when the task is scheduled to be in memory and for how long. The time during which a task is in memory is called its *dwell* time. The length of time tasks are in memory is dependent in part upon the overhead cost of swapping the task.

The Tera MTA requires that the entire address space of a task be resident while the task is executing. It is strictly a swapping system and provides no support for demand paging. Thus, our model of I/O assumes that no task can start executing until its entire address space is swapped in, and no tasks can continue to make progress once any of its address space is swapped out.

Before describing the memory scheduling policy, we define a model for swapping overhead applicable to the MTA and many other parallel systems and use this model to define the cost of swapping overhead. We show that based on the overhead of swapping, it is appropriate for a task's dwell time to be set in proportion to the amount of memory the task requires.

6.1 Swapping Overhead

Periodically, the memory scheduler will reassign a subset of memory to a different set of tasks. The cost of memory scheduling is determined by the cost of swapping out a set of tasks and swapping in another set of tasks at these memory quantum intervals.

We use a simple model of I/O that defines a constant available swapping bandwidth r, and computes the time to swap m memory units as m/r. Thus, we assume large memory tasks utilize all the I/O bandwidth that is available for swapping if there are no other swapping activities in progress simultaneously. We define the overhead of swapping out a set of tasks and swapping in another set of tasks as the total space-time (e.g., byte-seconds) that memory is not available for execution.

We define the following swapping algorithm for swapping out a set of tasks and swapping in another set of tasks. Choose the first task to be swapped in ($inTask$) and the first task to be swapped out ($outTask$), arbitrarily. Begin swapping out $outTask$ until either: (1) there is enough room for $inTask$ or (2) $outTask$ is fully swapped out. Now swap part or all of $inTask$ in to fill the memory vacated by $outTask$. If $inTask$ is fully swapped in, notify its processor scheduler that it is ready to begin execution, and choose another $inTask$. In any event, continue swapping out $outTask$. When $outTask$ finishes swapping out, choose another $outTask$ and continue. Alternate in this fashion between $inTask$ and $outTask$ to: (1) start each $inTask$ as early as possible, and (2) stop each $outTask$ as late as possible.

The overhead of the swapping algorithm is depicted in Figure 1. Tasks $a1$, $a2$, and $a3$ are to be swapped in and tasks $d1$,$d2$,and $d3$ are to be swapped out. For this example, the memory size of $a1$ is equal to $d1$, the size of $a2$ equals that of $d2$, and the size of $a3$ equals $d3$.

Figure 1(a) illustrates the overhead of swapping when these tasks are chosen for swapping (in or out) in ascending order of size. The total space-time overhead required to start a task consists of the time to swap out the task or tasks currently allocated the memory and the time to swap in the new task. For example, to swap out task $d1$ of size m requires m/r time during which m units of memory is unavailable. Each swap (in or out) of size m consumes an overhead depicted as a lightly-shaded rectangle of width m and height m/r for a total memory-time overhead area of m^2/r. To swap in task $a1$ of size m requires additional overhead of m^2/r for a total overhead cost of $2m^2/r$ to swap out one task for another of the same size. The total overhead cost of swapping is the summation of the individual swapping events.

Figure 1(b) illustrates the swapping overhead for a different ordering of tasks swapped in. Each partial swap out is accompanied by a partial swap in of the same size. The lightly shaded rectangles in Figure 1 depict the overhead of these swapping events. As before, the combined time space-time for a partial swap out/swap in pair of m memory is $2m^2/r$, but total cost of this overhead is different from that in Figure 1(a), since the values of m reflect a partial swap-in or out of a task. The partial swap-out/swap-in pairs occur sequentially. The northeast and southwest vertices of the swap-in/swap-out rectangles form a line with a slope of $2/r$. The dark-shaded area above this line represents the time during which a portion of memory is allocated to an incoming task, but the memory is not utilized, because the entire task is not memory resident. Similarly, the dark-shaded area below the line represents the time during which a portion of memory is allocated to an outgoing task, but the memory is not utilized, because the task is suspended while it is being swapped out. The arrows indicate the time at which a task is entirely swapped in (and can start execution) or the time at which a task to be swapped out stops executing. The total overhead of swapping is the summation of the space-time overhead above and below the line. This space pictorially forms a staircase shape, in which the width of each step is the size of memory for the incoming or outgoing task. The height of each step can be computed from the slope of the line. In general, let I be the set of tasks to be swapped in and D be the set of tasks to be swapped out. The total area representing the overhead above the line is $\sum_{\forall i \in I} m(i)^2/r$, where $m(i)$ is the size of task i. Similarly, the area below the slope is $\sum_{\forall j \in D} m(j)^2/r$. Thus, the total overhead incurred is:

$$\sum_{\forall k \in I \cup D} m(k)^2/r \tag{1}$$

It is easy to see that this algorithm achieves the minimal swapping overhead. Certainly, there is no benefit in discontinuing swapping out one task in order to swap out another, or discontinuing swapping in one task in order to swap in another; these actions only increase the overhead. Also, no benefit is gained by swapping multiple tasks simultaneously. (In fact, although our simple I/O model does not reflect it, additional overhead would be incurred from contention between two simultaneous swapping events.) Our policy, which completes the swap-in of one task before starting the swap-in of another and completes the swap-out of one task before suspending another task to be swapped out minimizes the time during which memory is not utilized. Furthermore, our swapping algorithm yields the same overhead for any sequential ordering of jobs to be swapped out and any ordering of jobs to be swapped in.

In summary, the swapping is simplified since the overhead of swapping is independent of the order with which tasks are swapped in or out. Memory scheduling is also simplified since the overhead of swapping out tasks whose dwell times expire concurrently is the same as the cumulative overhead of a schedule that staggers the expiration of dwell times. Furthermore, a task's contribution to the swapping overhead grows quadratically with its size. To maintain a

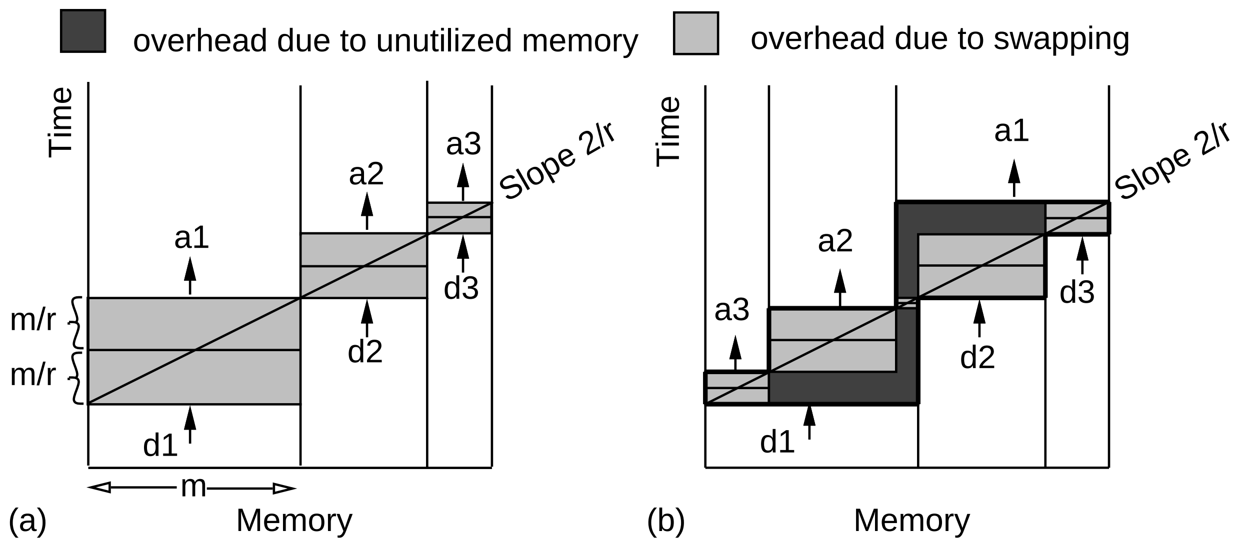


FIGURE 1: Overhead of Swapping Algorithm

constant percentage overhead per task, the dwell time of a task should be proportional to its size.

Finally, our I/O model assumes that an entire task's address space must be resident in memory before a task can execute. While other systems may provide a memory replacement policy that allows tasks to continue executing with a partially resident working set, we believe the working set for large, memory intensive applications is quite large. Thus, the swapping overhead for paging systems is similar. Furthermore, demand paging in distributed memory systems can severely degrade performance [4].

6.2 User Demand

Typically, a task's importance is represented by the priority level it is assigned. Tasks at higher priority levels receive preferential consideration in the allocation of resources. On the Tera MTA, the task's priority is expressed in terms of the desired performance of the task in the system. The user defines a demand of resource consumption for a task, in units of space-time memory residency per unit time. Presumably, higher demands justify increased monetary charges. If the system is saturated or underutilized, the demand will overestimate or underestimate, respectively, the average memory occupied by the task. The purpose of the demand parameter is to permit a quantifiable differentiation among tasks that is tied to system performance. The memory scheduler uses this demand to determine the order in which tasks are considered for execution.

Tasks are ordered by rank. The memory scheduler defines a task's rank as a linear function of the ratio between its demand of memory consumption and its achieved consumption. A task's rank is re-evaluated as it accumulates time in memory. Let $demand(i)$ be the demand for task i. Then the rank of task i at a time t is defined as the sum of two terms:

$$rank(i,t) = timeAtCross(i,t) + (dwellTime(i) \cdot memory(i))/demand(i) \quad (2)$$

$timeAtCross(i,t)$ for task i is the time at which the memory resource consumption acquired so far matched its demand:

$$timeAtCross(i,t) = totalConsumption(i,t)/demand(i) \quad (3)$$

where $totalConsumption(i,t)$ is the total memory consumption acquired by task i up to time t. $dwellTime(i)$ is the dwell time for task i, and $memory(i)$ is the size of task i. $timeAtCross$ for a new task is the current time. $timeAtCross(i,t)$ only needs to be updated when a task is swapped out.

To illustrate this ranking, Figure 2(a) shows an example memory schedule for two tasks. The shaded areas represent allocations to other tasks not of interest in this example. Figure 2(b) shows the resource consumption profile for tasks 1 and 2 as a result of the schedule. For example, after Task 1 executes for time t, it has acquired $m \cdot t$ memory resources where m is its size. Task 2 has acquired $m \cdot t/4$ of memory-time by the same time, t.

Figure 2(b) illustrates pictorially the computation of rank for Tasks 1 and 2 after the execution of the memory schedule. A task's demand is represented by a sloped line. Conceptually, the first term of rank, $timeAtCross$, attempts to order tasks according to their urgency in acquiring memory resources to meet their desired rate. This is determined by a horizontal line from the tasks memory consumption profile to its demand slope. Tasks with a earlier timeAtCross are considered more crucial. Note that timeAtCross can be either in the past or in the future.

The second term of rank compensates for the different rate at which tasks acquire resources during a quantum ($dwellTime$) and their different resource demands. The shaded rectangles have height $dwellTime(i) \cdot memory(i)$, and thus width $dwellTime(i) \cdot memory(i)/demand(i)$. Given two tasks with the same $timeAtCross$, the task with the higher ratio of per-quantum memory consumption to demand will be considered less crucial. It can wait longer before being scheduled and still achieve its demand. From this example, after the current schedule is complete, the rank for task 2 is less than the rank for task 1, indicating that task 2 will be considered for scheduling before task 1.

6.3 Memory Scheduling Policy

Both the MB-scheduler and the ML-scheduler use the same algorithm for allocating memory to the set of tasks under its control. Each scheduler decides which tasks to move in or out of memory based on a ranking of the tasks in the system, where the ranking reflects the ratio of

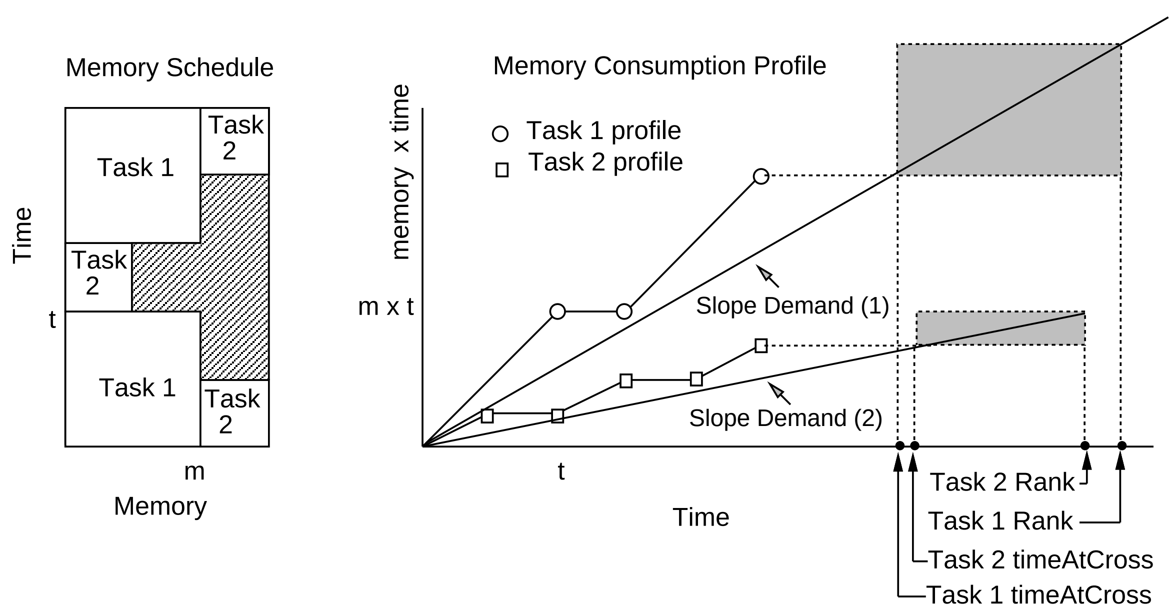


Figure 2: Task Ranking Strategy

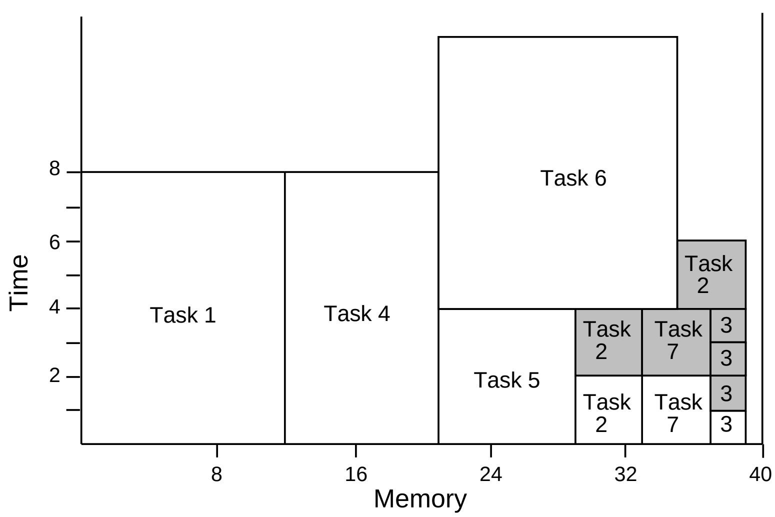
the task's demand and the achieved rate of memory consumption. The goal of the memory scheduler is primarily to provide high memory utilization while achieving the overall goal rate of execution for the tasks.

We define a minimum dwell time t_{min} as the miniminum dwell time for the smallest task. As discussed in Section 6.1, a task's memory dwell time depends on its size; the larger the task the longer the dwell time. Specifically, the dwell time for a task with memory requirement m is defined as $2^{\lceil log_2 m \rceil} * t_{min}$ where $2^{\lceil log_2 m \rceil}$ is the smallest power of two greater than its memory requirement. Requiring dwell times to step by powers of two allows for buddy-style coalescing of memory-time allocations between small tasks allocations and large task allocations.

The scheduler uses a first-fit strategy for scheduling available memory. The scheduler is invoked whenever a block of memory becomes available. A list of ready-to-run mi-swapped tasks is maintained, sorted by rank, where tasks with a lower rank are considered first. The scheduler selects the first task to be scheduled. If the memory available is greater than the task's memory requirement, the task is scheduled to be swapped in, the available memory is reduced by the size of the task, and the task's rank is updated to reflect its current residency. Otherwise, the next task in the list is examined. This procedure continues until either all the memory is scheduled or the available memory is less than the size of the smallest task. Thus the scheduler's job can be thought of as packing rectangular boxes along a single dimension M wide, where M is the amount of memory available. The scheduler sorts the scheduled tasks by dwell time expiration. When the dwell time expires for a set of tasks, a memory quantum has expired. At each quantum expiration, the scheduling algorithm is repeated to allocate the available memory to a (possibly) new set of tasks. The algorithm also ensures that the currently executing tasks are not mi-swapped if no higher ranking tasks is waiting to be scheduled.

The memory scheduler is notified when an executing task blocks. If the task's memory residence time exceeds a minimum residency requirement, the task is mi-swapped and placed on a blocked list. When the task is unblocked, it is inserted in the ready-to-run task list.

Figure 3 illustrates an example schedule for 7 tasks. The 7 tasks are listed in rank order by their sizes. The x axis represents memory, and the y axis represents time. Each rectangle represents an allocation of memory to a task. The duration of the schedule varies with the dwell times of the tasks scheduled. Note that task 6 cannot be scheduled initially, since its size is greater than the memory available after tasks 1 through 5 are scheduled. Also, even though task 4 requires only slightly more memory than task 5, the dwell time of task 4 is twice the



Task	1	2	3	4	5	6	7
Size	12	4	2	9	8	14	4

Figure 3: Example Memory Schedule

dwell time of task 5, since a task's dwell time is a factor of the lowest power of two greater than its size.

Each horizontal line in Figure 3 represents a quantum expiration time at which a set of tasks becomes eligible to be swapped out of memory. At this time, the scheduling algorithm is repeated. Task 3's quantum expires first. Assuming there are no other tasks in the system, task 3 will be rescheduled (this is represented in the figure by a shaded rectangle). Similarly, tasks 2 and 7 will be rescheduled until time 4. At that time, assuming tasks 3 and 5 drop in rank below task 6, tasks 3,5, and 7 will be swapped out, and task 6 will be scheduled.

When a new task arrives, it is placed in the task list in sorted order using its assigned rank. When a task exits, the scheduler tries to fill the task's memory with another task on the task list if the remaining dwell time is long enough to support the overhead of swapping the new task into memory.

The memory schedulers must also handle requests by the tasks for dynamic memory allocation. A subset of the available memory is reserved to handle these requests. Still, if a task's remaining dwell time is short or if the memory is not available, the task's memory allocation request fails. A message is sent to the task's processor scheduler to pm-swap the task. Once the task is pm-swapped, it is mi-swapped, and if it had sufficient remaining dwell time, another task from the task list is chosen, if possible, to occupy the vacated memory for the remaining dwell time. The total memory requirements for the task is incremented so that the next time the task is swapped in, it receives the larger allocation. If a small memory task requests a total allocation in excess of the class of tasks handled by the ML-scheduler, then the task is placed under the control of the MB-scheduler.

When memory becomes available, either through a task's exiting or freeing memory, the MB-scheduler looks at the remaining dwell time of the freed memory. If a large fraction of the dwell time remains, the scheduler looks for an appropriately-sized task in the swapped-out task list.

6.4 Starvation Avoidance

The scheduler's first-fit scheduling strategy is a straightforward and simple policy designed to provide high memory utilization. However, this simple strategy alone is not sufficient to avoid starvation, since there is no guarantee that a large enough block of memory will eventually become available to schedule a large memory task.

In order to avoid large task starvation, when the available memory is not sufficient to schedule the next task on the swapped-out task list, the earliest possible time is determined at which enough memory will be available to schedule the task, given the current schedule. The scheduler will refuse to schedule subsequent tasks that do fit in available memory if their dwell time would result in the higher ranking task not being schedulable at the later time. This gives huge jobs a chance to run within a time interval corresponding to their rank.

6.5 Extensions

The memory schedulers do not consider a task's processor requirements when scheduling tasks to be swapped in. An extension could consider the number of teams required by the task as well as the task's memory to try to schedule sets of jobs in memory concurrently that don't overutilize or underutilize the processing resources.

Currently, memory is statically divided between the MB-scheduler and the ML-scheduler. To increase the utilization of memory, the MB-scheduler could release unallocated memory to the ML-scheduler for use until the next MB-scheduler quantum expires. Similarly, a more flexible partition could be implemented, where the amount of memory reserved for the small memory tasks varies with the changing load exerted on the system by the batch and interactive tasks. However, research by Ashok and Zahorjan [2] suggest that the benefits of such dynamic partitions are not easily attained.

7 Processor Scheduler

Early research on processor scheduling for parallel machines highlighted the need for co-scheduling [16]; that is, ensuring that all parallel activities of a job execute at once. This avoids inefficiencies due to synchronization losses when some activities are blocked waiting for progress from another parallel activity that is not scheduled. On traditional multiprocessors, scheduling requires that either a processor be allocated to a single job or that context switching among multiple jobs assigned to a processor be globally coordinated. The former strategy is known as space sharing, in which the processors are partitioned among the jobs in the system [21]. The latter strategy is generally realized by round-robin scheduling policies that rotate possession of the processors among the jobs in the system in a coordinated manner [11]. Studies comparing the performance of space-sharing and time-sharing systems [25, 17] conclude that space sharing policies make more efficient use of processors that might otherwise be idle when running a single job. However, in dividing the processors among multiple jobs, space sharing policies often allocate fewer processors to jobs than the job's parallelism may allow. Tucker and Gupta [21] describe "process-control", an approach that adapts a job's parallelism to the processors available to it. Also, efficient space sharing policies must be able to adapt the partition sizes and configurations based on the changing system load [25, 12, 13, 15, 14, 15, 18].

Tera's multi-threaded architecture makes it possible to combine space-sharing and time-sharing and realize the benefits of both. Multiple tasks execute simultaneously on a single processor. Thus, there is no need for global coordination of context switching. The operating system space shares or partitions the available protection domains among the tasks in the system. Since each processor has multiple protection domains, space-sharing is accomplished on the basis of a protection domain rather than a processor. A task is never allocated fewer protection domains than it has teams. Time-sharing is performed at a very fine-grained level. Teams from different tasks executing on a single processor compete for hardware streams on the processor. The scheduling of streams is controlled by the hardware and directed by user level instructions. The operating system is not involved in hardware stream allocation within a processor.

The goals of the Tera processor schedulers are: (1) efficient use of processing resources; (2) fairness and starvation avoidance; (3) good interactive response time; and (4) mechanisms for job differentiation.

Achieving high processor utilization requires between 20 and 80 active streams. Teams from single-team tasks typically have few streams (normally one or two), while teams from multi-team tasks have many more streams (usually between 20 to 40). Given 16 protection domains per processor, a processor containing only single-team tasks likely would not have enough active streams to keep the processor fully utilized. Thus, keeping a processor busy requires some number of teams from multi-team tasks.

To obtain the desired mix of teams from multi-team and single-team tasks, the protection domains on each processor are divided between the multi-team task scheduler (PB-scheduler) and the per-processor single team task scheduler (PL-scheduler). Based on our understanding of exploitable parallelism found in large tasks, we expect a processor to be able to service somewhere between two to three teams from multi-team tasks. Call this number the *target-mt-per-proc*.

The PB-scheduler places target-mt-per-proc teams from multi-team tasks on each processor while the per-processor PL-scheduler uses the remaining protection domains. If the PB-scheduler requires an additional protection domain, it requests it from the PL-scheduler. The PL-scheduler responds by swapping out a single-team task from a protection domain of its choice and furnishing it to the PB-scheduler. Similarly, when the PB-scheduler has an unneeded protection domain, it can return it to the PL-scheduler.[5] When a task changes from single team to a multi-team task, the task migrates from its PL-scheduler to the PB-scheduler. The converse

[5]This behavior can be changed by per-site parameters, letting system administrators tune their systems for their local requirements.

does not occur because we expect the multi-team task to continue to exhibit a need for large numbers of streams and teams in the future.

The processor schedulers do not look at utilization explicitly. The load of a team (number of streams) varies so dynamically with respect to the frequency at which the operating system makes scheduling decisions, that the information can't be used as a valid predictor of utilization. As a result, we assume that the teams from multi-team tasks exert equivalent loads on the processors. Similarly, the load from single team tasks is assumed to be uniform. The runtime system can adjust its load based on total processor utilization since it makes these decisions with much greater frequency.

The overhead of mp-swaps and pm-swaps is on the order of a traditional context switch and of a much smaller magnitude than memory swapping. When a pm-swap occurs, the user runtime notifies all executing streams to save their register state in parallel. When completed, the runtime hands control to the operating system, which saves a handful of registers comprising the privileged protection domain state. When a mp-swap occurs, the operating system initializes the privileged protection domain state and starts a single user stream; the user stream arranges for the remaining streams to restore their register state in parallel and resume execution. Loading and unloading of individual teams also proceeds in parallel, in a data flow fashion. In general, mp/pm-swaps are fast and don't require the movement of memory.

Thus a single stream, single team task has very low overhead while a very parallel task will We first describe the single-team processor scheduler, then the multi-team processor scheduler.

7.1 Single-Team Task Processor Scheduler (PL-scheduler)

Since single-team tasks execute in a single protection domain, their scheduling does not require global knowledge or coordination between different processors. This allows scheduling of single-team tasks on a per processor basis by independent instances of the PL-scheduler.

The PL-scheduler uses a priority mechanism for job differentiation of memory resident tasks that follows the Unix standard [3] rather than the ranking mechanism used in scheduling memory. A newly created task is given a high priority. When a task's processor quantum expires, its priority is decremented. A task's priority rises when it unblocks (e.g. I/O occurs) or it has not received service for a long time (anti-starvation). Site administrators can modify the amount the priority changes for each of these events.

As discussed in Section 5, each memory scheduler selects a PL-scheduler for a newly im-swapped single-team task. Once a task has been assigned to a PL-scheduler it remains there until mi-swapped, the task exits, or the PL-scheduler explicitly sends it to another PL-scheduler. When a new task is assigned to the PL-scheduler, the scheduler allocates a protection domain and mp-swaps the task. If no protection domains are available, it places the task on its *Ready* queue.

After mp-swapping a task, the PL-scheduler sets an alarm for a task quantum. If the task quantum alarm expires, the PL-scheduler reduces the priority of the task, and sees if any other ready tasks have a higher priority. If another task is eligible and the loaded task is swappable, [6] it pm-swaps the task and places it on the Ready queue (assuming it is not blocked on I/O). The PL-scheduler then mp-swaps the higher priority task.

When a task unblocks, e.g. due to I/O completing, the PL-scheduler mp-swaps the task if a protection domain is available; otherwise the scheduler moves the task onto the Ready queue and increases its priority.

Task suspension involves an external agent, such as Unix job control, requesting the task to be removed from the processor. When the PL-scheduler receives a suspension event, it pm-swaps the task but does not place it on the Ready queue; the task remains suspended until it is explicitly resumed. Meanwhile, the PL-scheduler loads in the highest priority task from the Ready queue. When the PL-scheduler get a resume event for the suspended task, it allocates a protection domain and loads the task or, failing that, puts it on the Ready queue.

[6] For example, not exiting, suspending, etc.

Having memory schedulers select processors seeks to maintain a balanced load when single team tasks are im-swapped. In the absence of memory swapping, load imbalances could occur. Based on previous research [5] [6], PL-schedulers migrate work in two situations. When a new task is created (normally as part of the Unix *fork*() system call), the PL-scheduler selects another PL-scheduler at random and checks the length of that scheduler's Ready queue. If the queue is empty, the child is sent to that processor; otherwise it remains on the current queue. Similarly, when the length of a PL-scheduler's Ready queue exceeds some site settable maximum, the scheduler checks a randomly selected PL-scheduler. If the queue is empty, the PL-scheduler migrates tasks from its Ready queue. The number of tasks that migrate depends on the amount of available protection domains on the recipient.

To prevent the starvation of tasks, the PL-scheduler periodically sweeps its Ready queue and increases the priority of tasks that have not run since the last check. This allows lower priority tasks to rise in the queue and eventually get a chance at the processor.

7.2 Multi-Team Task Processor Scheduler - The PB-scheduler

Scheduling multi-team tasks presents the problem of allocating protection domains on multiple unique processors in order to co-schedule the task's teams. The PB-scheduler allocates target-mt-per-proc protection domains on each processor to the multi-team tasks.

The primary goal of the PB-Scheduler is high processor utilization. The scheduler also ensures that: (1) every processor has at least one and no more than target-mt-per-proc teams from multi-team tasks; (2) teams from the same task will be on different processors except when the number of teams is greater than the number of processors; and (3) every multi-team task runs at least once during its memory dwell time (the allotted time the task is memory resident). This last goal seeks to minimize wasted system overhead that results when a task is im-swapped then mi-swapped without executing.

The decision whether or not to put fewer than the target-mt-per-proc teams on each processor, goal (1), relates to the operation of the user runtime. The user runtime may request a team during the execution of its program, due to the program's increasing parallelism. If the PB-scheduler packs each processor to its maximum, each such request would cause the PB-scheduler to refuse the request or pm-swap the job until resources came available. By leaving some protection domains available, the scheduler can service these requests promptly. New domains being freed when the program's parallelism diminishes can also be used to service growth of other programs. Determining the correct balance of how full to pack the processor is difficult, and is a parameter that will be tuned with experience.

To accomplish goal (3), the PB-scheduler uses round robin scheduling with a processor quantum small enough in comparison with the memory dwell time used by with the MB-scheduler to ensure every task runs at least once while memory resident. Each round of the round robin is called a *cycle*.

7.2.1 Bin Packing Algorithm

Assigning teams to protection domains can be thought of as a bin packing problem. The target-mt-per-proc can be thought of as the number of bins available while the number of processors is the size of each bin. Let D be the number of protection domains allocated to the PB-scheduler per-processor and P be the number of processors. Each task is represented by the number of protection domains it requires.

The goal is to fit the tasks into the minimum number of bins. Since the problem is NP-complete [10], a standard heuristic solution uses a decreasing first fit algorithm (see Standard Algorithm in Figure 4a). Sort the tasks by decreasing number of teams. Let x_i be the number of teams in task$_i$. For each i, put x_i in the first bin that has room for it. If all the bins are full, skip to the next task. To improve this algorithm, lay the bins end to end and allow a task to span bins. In effect, there is one bin of size D *times* P. This change allows tasks larger than the number of processors P to be loaded; such situations can result from running a checkpoint file on a degraded system (see Improved Algorithm in Figure 4b).

One problem of the traditional first-fit algorithm is that it only allocates contiguous[7] protection domains to a single task. This becomes a problem as tasks exit and new ones arrive. Under the improved algorithm, if exiting tasks free up non-contiguous protection domains (for example, Task 1 and 3 in Figure 4), an arriving Task 4 with four teams that could fit into free domains still would not be scheduled (see Figure 5a).

We modify the algorithm to remove that restriction (See Figure 4c). We use a bit vector to represents all the protection domains in the system; high bits indicate available protection domains The algorithm iterates through the bit vector selecting free protection domains that are also on unique processors. This allows Task 4 to be loaded in our example (see Figure 5b).

7.2.2 Scheduling a Cycle

At the start of a cycle, all tasks have equal opportunity to be scheduled. Tasks waiting to run reside in the *Waiting* queue. The PB-scheduler assigns as many tasks that fit into available protection domains and sets an alarm for the processor quantum. All the tasks run for the same amount of time.

When the processor quantum expires, the PB-scheduler conceptually removes the loaded tasks and attempts to schedule tasks in the following order:

1. tasks in the anti-starvation queue (described below)
2. tasks that recently arrived and were scheduled in the previous quantum (described below)
3. tasks waiting to run in this cycle (*waiting* queue)

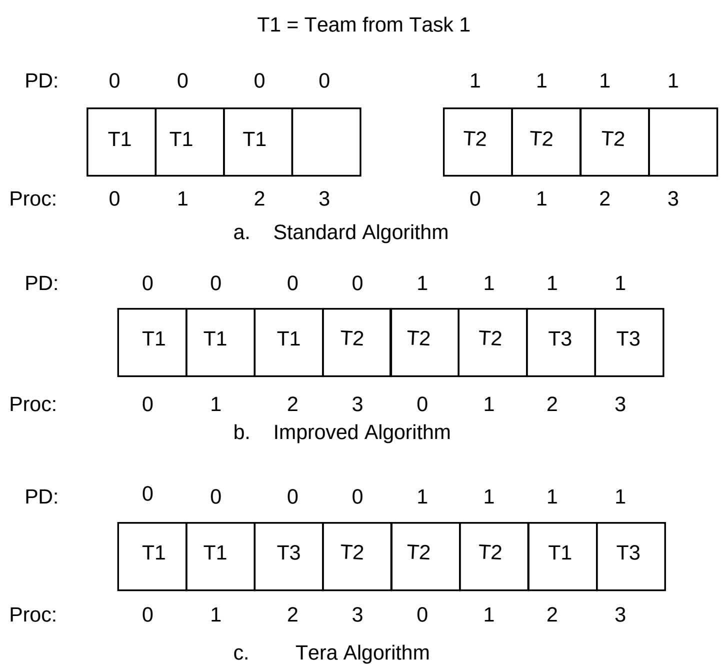


FIGURE 4: Examples of Bin Packing Algorithm

[7]Bins span processors so that each bin consists of the same numbered protection domain. By contiguous we mean that the task is assigned to protection domains next to each other in the bin.

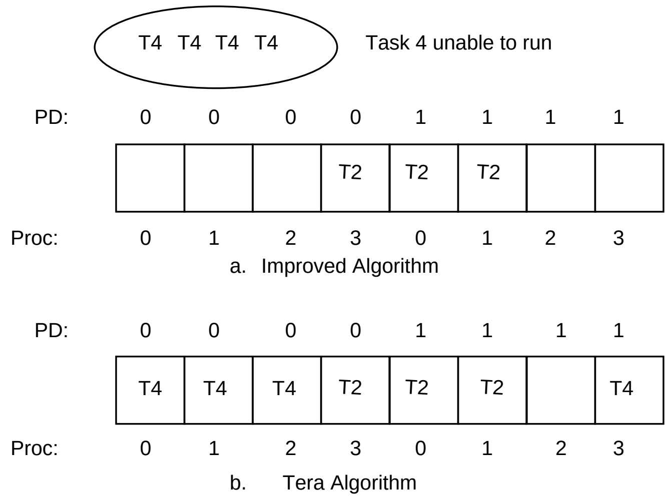


FIGURE 5: Bins after Task 1 and 3 Exit and Task 4 Arrives

4. tasks that have already run in this cycle (*ran* queue)
5. tasks currently loaded

For tasks that are currently loaded and scheduled again for the next quantum, the scheduler makes an effort to rearrange protection domain assignments to allow the loaded task to remain in the same protection domains. If this succeeds, the loaded task will continue to execute without any interruptions. If it fails, the task incurs the overhead of a pm-swap followed by a mp-swap.

After scheduling the next quantum, daemons pm-swap those tasks not scheduled for the next quantum and mp-swap the new ones in parallel. Pm-swapped tasks are placed in the ran queue. When the waiting queue is empty, the cycle is complete and the ran queue becomes the waiting queue.

The PB-scheduler tries to assign protection domains to newly arriving tasks regardless of how much time remains in this processor quantum. Since the new task will, on average, only receive a half of a quantum, the PB-scheduler places the task in a list to be scheduled before those on the waiting queue.

Starvation of tasks in the waiting queue is possible using this scheme. Suppose task A in the waiting queue needs protection domains on all the processors and task B needs half of the processors and is currently loaded. Task C, which also needs half of the processors, arrives, gets loaded and scheduled for the next quantum. Meanwhile, task B exits and the quantum expires. Task C is scheduled again while task A continues to wait. This pattern of arrivals and departures, however unlikely, could continue indefinitely, and task A would starve.

To avoid starvation, the PB-scheduler periodically checks tasks in the waiting queue for starvation. Any tasks that have not received service since the last check are placed in a starvation queue. This queue is first in line for scheduling in the next quantum.

8 Conclusions

Scheduling appears at all levels in a Tera System: in compiler generated code, in the runtime system, and in the operating system. The overall goal of our design efforts has been to ensure that these three major scheduling systems work effectively and efficiently both within their respective domains and in their interactions with one another.

The compiler schedules instructions. It also acquires and schedules stream resources for the implicit parallelism of a program. The overheads involved in scheduling done by the compiler are typically negligible, and the time frame of a compiler's schedule is normally shortest of all.

The runtime acquires and schedules stream resources for executing explicit program parallelism. It also imposes an order on the execution of parallel work. There is slightly more overhead associated with this kind of parallel work than that scheduled by the compiler. Appropriately, the granularity of the work is expected to be somewhat larger, and the scheduling time frame, too, is greater than that considered by the compiler.

The operating system schedules memory and protection domain resources among jobs contending for the system. The time frame considered by the processor scheduler is much greater than that of the runtime; and, again, this is natural since each piece scheduled may consist of many futures or compiler scheduled parallel loops. And the time frame of the memory scheduler is still larger: the overhead of swapping large jobs in and out of memory is the greatest overhead of all.

Utilizing schedulers, each specifically designed for a particular level in a hierarchy of time frames and work granularities, is a natural approach for scheduling parallel work on a multiprogrammed machine. We expect our designs to be particularly efficient, however, because they are able to execute almost independently of each other: Interactions between the memory scheduler and processor scheduler are minimal. Between the operating system and the runtime systems, typical interactions are limited to occasional requests for additional protection domains; creation of additional streams, trap handling, and load estimations are handled by the runtimes independently. Compiler generated code, once assigned by the runtime to specific teams, operates independently of the runtime, reserving, creating and destroying streams immediately, via hardware instructions. We believe the decomposition of scheduling obligations in this way will lead to negligible overheads and very high processor utilizations.

This paper discussed several notable, even novel, characteristics of our schedulers: The memory scheduler distinguishes jobs by rank, where a job's rank is a function of its demand for memory and its current allotment. We described an algorithm for swapping jobs that minimizes the memory overheads and showed that swapping overhead grows quadratically with the size of jobs being swapped. Surprisingly, the algorithm's performance is independent of the order of swapping events: this means we can alter the order, should the need arise, without incurring additional overheads or computation. The processor schedulers use efficient mechanisms for bit vector iteration provided by the Tera MTA to efficiently co-schedule jobs onto protections domains. The runtime growth policy is based on relatively recent competitive-style algorithms, and is extremely simple while still promising robust adherence to predictable performance bounds.

Our schedulers appear reasonable in simulation, though due to the relatively slow speed of simulation, we can run only small workloads. We expect to refine our algorithms with experience on the real machine: we eagerly await the arrival of the Tera MTA prototype.

References

[1] 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.

[2] I. Ashok and J. Zahorjan. Scheduling a mixed interactive and batch workload on a parallel,shared memory supercomputer. In *Supercomputing 1992*, Nov 1992.

[3] M. J. Bach. *The Design of the Unix Operating Systems*. Prentice-Hall, Inc., 1986.

[4] D. Burger, R. Hyder, B. Miller, and D. Wood. Paging tradeoffs in distributed-shared-memory multiprocessors. In *Supercomputing '94*, November 1994.

[5] D.L. Eager, E.D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, May 1986.

[6] D.L. Eager, E.D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Evaluation*, March 1986.

[7] R. Essick. An event-based fair share scheduler. In *Winter 90 USENIX Conference*, January 1990.

[8] A. Gottleib, B. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2), April 1983.

[9] A. Gupta, A. Tucker, and S. Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *ACM SIGMETRICS Conference*, May 1991.

[10] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Alhorithms*. Computer Science Press, 1984.

[11] S. Leutenegger and M. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *ACM SIGMETRICS Conference*, May 1990.

[12] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed, shared memory multiprocessors. *ACM Transactions on Computer Systems*, May 1993.

[13] C. McCann and J. Zahorjan. Processor allocation policies for message-passing parallel computers. In *ACM SIGMETRICS Conference*, May 1994.

[14] V. Naik, S. Setia, and M. Squillante. Performance analysis of job scheduling policies in parallel supercomputing environments. In *Supercomputing 1993*, November 1993.

[15] V. Naik, S. Setia, and M. Squillante. Scheduling of large scientific applications on distributed memory multiprocessor systems. In *6th SIAM Conference on Parallel Processing for Scientific Computation*, March 1993.

[16] J. Ousterhout. Scheduling techniques for concurrent systems. In *3rd International Conference on Distributed Computing*, Oct 1982.

[17] S. Setia, M. Squillante, and S. Tripathi. Processor scheduling on multiprogrammed, distributed memory parallel systems. In *ACM SIGMETRICS Conference*, May 1993.

[18] K. Sevcik. Characterization of parallelism in applications and their use in scheduling. In *ACM SIGMETRICS Conference*, May 1989.

[19] J. H. Straathof, A. A. Thareja, and A. K. Agrawala. Unix scheduling for large systems. In *Denver USENIX Conference*, January 1986.

[20] J. H. Straathof, A. A. Thareja, and A. K. Agrawala. Methodology and results of performance measurements for a new unix scheduler. In *Washington USENIX Conference*, January 1987.

[21] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *12th ACM Symposium on Operating System Principles*, December 1989.

[22] D. Wagner and B. Calder. Leapfrogging: A portable technique for implementing efficient futures. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[23] J. Zahorjan, E. Lazowska, and D. Eager. Spinning versus blocking in parallel systems with uncertainty. In *International Sympossium on Performance of Distributed and Parallel Systems*, December 1988.

[24] J. Zahorjan, E. Lazowska, and D. Eager. The effects of scheduling discipline on spin overhead in shared memory parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, April 1991.

[25] J. Zahorjan and C. McCann. Processor scheduling in shared memory multiprocessors. In *ACM SIGMETRICS Conference*, May 1990.