Piranha Scheduling: Strategies and Their Implementation

David Gelernter, Marc R. Jourdenais, and David Kaminsky

YALEU/DCS/RR-983
September 1993

Piranha Scheduling: Strategies and Their Implementation

David Gelernter, Marc R. Jourdenais, and David Kaminsky
Department of Computer Science
Yale University
New Haven, CT 06520

September 1, 1993

Abstract

Piranha is a execution model for Linda[1] developed at Yale by Kaminsky and others[4] to reclaim idle cycles from networked workstations for use in executing parallel programs. Piranha has proven to be an effective system for harnessing large amounts of computing power. Most Piranha research to this point has concentrated on efficiently executing a single application at a time. In this paper we evaluate strategies for scheduling multiple Piranha applications. We examine methods for predicting idle periods and the effectiveness of scheduling strategies which make use of these predictions. We present a prototype scheduler for the Piranha system implemented using the process trellis software architecture for networks of workstations.[6]

This work was supported by AASERT Grant F49620-92-J-0240 and NASA Training Grant NGT-50719.

1 Introduction

The proliferation of low cost workstations in academic and corporate environments has given many institutions aggregate computational power rivaling that of yesterday's supercomputers for a fraction of the cost. Most of this latent power, however, is wasted, as workstations go idle much of the time. The Piranha system seeks to reclaim these lost cycles by running parallel programs which use the idle resources of a local area network. The problem of how to effectively schedule multiple Piranha jobs waiting to make use of these lost cycles has not previously been explored. The current Piranha implementation at Yale uses simple randomized or first-come-first-served policies for assigning Piranha jobs to physical nodes in the network. While these are time-honored and fair scheduling strategies, it is not unreasonable to believe that we could reduce job turnaround times with smarter strategies which take into account the behavior of the physical nodes in the system and the attributes of the jobs to be scheduled. Since the Piranha system provides a mechanism for an external scheduling program to override the default policies, we have the opportunity to create an independent scheduler which implements these "smarter" strategies.

As we consider a smarter scheduler, two significant questions arise:

1. What strategies are appropriate for scheduling Piranha jobs on a network of workstations?

[1]Linda is a registered trademark of Scientific Computing Associates, New Haven

2. How should we build an external scheduling program in order to provide support for dynamic and intelligent scheduling policies?

We first explore a collection of strategies for scheduling Piranha jobs and evaluate them through simulation. We then propose an implementation of an external Piranha scheduler which is capable of incorporating these strategies (and much more complex strategies as well) in a straightforward and modular way using the Network Trellis software architecture.

2 Piranha

Piranha is a system developed by David Kaminsky and others at Yale that allows parallel programs to adapt to changes in availability of resources as they run. A Piranha program, or *job* is assigned to a collection of workstations by a scheduler. On each workstation the job has a representative worker process, or *piranha*, that lies dormant until the node becomes *available* for computation. A node is determined to be *available* or *unavailable* by a function of recent processor load and input device idle time. When the node becomes available, the local piranha becomes active and begins computing on behalf of the job. Piranha programs often follow a master-worker paradigm. A master, or *feeder*, process outputs descriptions of *tasks* representing the work to be done by the job to a Linda tuple space. When a piranha becomes active, it executes code which reads these task descriptions and completes tasks until its node becomes unavailable. When its node becomes unavailable (a user uses an input device or increases the processor load past a threshold) the piranha must *retreat*. This entails stopping work on the current task and outputting enough information to the task space to enable some other piranha to complete the task. Depending on the application, the next piranha to attempt a previously retreated task may have to start the task from the beginning or may be able to pick up where the previous piranha left off. In some applications, task redundancy may eliminate the need to even attempt to complete the retreated task. In this paper, we refer to a contiguous period of time in which a node meets the Piranha availability criteria as an *availability period* (AP).

3 The Scheduling Problem

The Piranha multiple job scheduling problem is similar to that faced by CPU job schedulers. Each job has certain attributes and our goal is to complete the jobs while maximizing use of available resources and minimizing turnaround time. The scheduling problem in the Piranha system is made difficult by the fact that the scheduler does not have complete control over which system resources are available. The scheduler does not know how many or which processors it will have available for Piranha computation at any given time.

For example, when a research group decides to have a meeting, their workstations will be available for an extended period of time. Suddenly the scheduler finds that it has a significant increase in Piranha cycles available. The scheduler cannot, however, know with certainty how long the meeting will last (or even that a meeting is the reason the workstations are available) unless explicitly told by some external agent, and thus cannot use this information to make intelligent scheduling decisions. As an example of such a decision, the scheduler might have in its job queue a particularly demanding Piranha program which requires an hour to complete each task. For this job, a task must be completely finished or restarted from the beginning (partial results in this case are useless). The scheduler has other jobs in its queue which have much shorter task completion times. If the scheduler knows for certain that the meeting will last just over an hour, this is the perfect time to schedule the demanding job on the workstations in the research group. It will have enough time to make progress before

anyone returns to his or her workstation. If the scheduler decides to schedule the job on the available workstations and the humans return in just under an hour, however, almost an hour of computing time on each node has been wasted. No tasks have been completed, and all piranhas must retreat as humans return to their workstations. The scheduler would have been better off scheduling a job (or jobs) with smaller tasks on the workstations so that progress would have been made on at least one job.

A CPU scheduler faces a different kind of uncertainty. The processors on which it wishes to schedule jobs are generally available to it. The CPU scheduler is concerned with the mix of CPU and I/O bursts within jobs. The scheduler attempts to keep the CPU busy by swapping jobs into the CPU when they need to do computation (a CPU burst) and out of the CPU when they need to wait for I/O (an I/O burst). Swapping a job out of the CPU tends to be less costly in terms of lost computation than retreating a Piranha task, so CPU schedulers need be less concerned with "retreats" than a Piranha scheduler. The Piranha scheduler is concerned with losing cycles due to node unavailability, while the CPU scheduler is concerned with losing cycles due to jobs occupying the CPU during their I/O bursts.

3.1 Related Work

The problem of scheduling multiple jobs on a distributed network of workstations has been explored in several research systems. A few of the more prominent systems are enumerated in [9]. The *Worms* project used programs composed of multiple segments, each executing on a different machine. A worm program segment would seek out idle machines on the network and replicate itself on the new machine, causing the worm to "grow." The *Condor* system also harnesses time on idle processors, but does so by scheduling independent background jobs on each idle processor. Condor does not support the distribution of parallel programs among multiple nodes. Perhaps the closest system to Piranha is Waldspurger's *Spawn* project at MIT.[9] This system supports the forking of processes in a parallel program onto idle machines, using a market scheduler. Like Piranha, Spawn spreads parallel applications across multiple idle nodes in a network. Market schedulers, such as the one used by Spawn and other distributed systems such as *Enterprise*,[5] have proven effective as distributed schedulers of multiple jobs in a network environment.

Distributed market scheduling policies can be effective in certain contexts, but are not necessarily the best way to schedule Piranha jobs. Distributed market schedulers strive to converge to effective global scheduling policies.[9] If it is possible to construct a global scheduler with minimal loss of efficiency, such a scheduler would be able to directly implement the goal policy. Market schedulers provide a straightforward means of implementing simple priority scheduling, by assigning jobs varying amounts of system currency depending on their attributes. It is not clear, however, how considerably more complex strategies, such as those taking into account node availability patterns as well as external calendar events, could be incorporated into such a scheme. The Network Trellis system provides support for creating a logically global yet physically distributed scheduler which is capable of supporting arbitrarily complex scheduling strategies. In a sense it provides us with a hybrid solution, giving much of the potential scalability and efficiency of a distributed scheduler with the added flexibility and determinism of a global scheduler. With this in mind, we will focus on global rather than distributed scheduling policies for the remainder of the paper.

3.2 The Piranha Scheduler

Our scheduler will periodically make global scheduling decisions for the Piranha system. The Piranha system keeps a ready queue of uncompleted jobs. The scheduler maps jobs from this queue to physical

nodes. When a job is mapped to a node, all Piranha computation on that node is done on behalf of that job until a new mapping is made. The scheduler periodically reads the ready queue and informs the Piranha system of new mappings. If a new mapping assigns a different job to the node, the old job is forced to retreat. The scheduler is thus preemptive. We assume that the computational demands of the scheduler will not significantly affect whether the nodes in the system are available or unavailable.

The choice of period for the scheduler is a tradeoff between accuracy of scheduling policy and system overhead. If the period is very small, the scheduler will be invoked frequently and use a relatively large number of system resources which may otherwise be used by the Piranha jobs themselves. This is particularly true if the scheduler executes on the same nodes used for Piranha computation. Even a more remote scheduler, however, may degrade the performance of the communication network. A frequently invoked scheduler is also likely to cause more retreats if it varies scheduling decisions more quickly. The primary benefit of a frequently invoked scheduler is that it reacts more quickly to changes in the job queue, such as the addition of new jobs and the completion of old jobs, and changes in node availability, making its decisions a more accurate reflection of the intended policy. A scheduler with a large period risks having significant changes in the system go unnoticed for large periods of time. It may also, however, be much less intrusive, consuming fewer system resources and causing fewer retreats. As an alternative to a purely periodic scheduler, an interrupt mechanism could be used to invoke the scheduler when important events, such as changes in the job queue, occur. This paper will focus on periodic schedulers.

We have chosen to schedule on the level of job assignments due to the current capabilities of the Piranha system. We also envision schedulers that work on the task level. Such a scheduler would control the order in which individual tasks were executed within a job. This is a finer level of control than is currently allowed by the system, but is an intriguing possibility for the future.

4 AP Prediction

In order to maximize use of system resources, an effective Piranha scheduler must ensure that when there are jobs in the system some job is ready to take advantage of a node that becomes available. In addition, the scheduler must consider retreats. Retreats in the worst case cause all work by the active piranha on the current task to be lost. Ideally, the scheduler would like to assign to a node a Piranha job which has a task length no greater than the length of the expected APs on the node. This ensures that the a task will be completed during each AP before a retreat occurs, and computational progress will be made. To match Piranha job task times to node AP lengths, the scheduler must have some way of predicting the length of availability periods on the nodes in the system as well as a way to predict the length of tasks for individual jobs.

In order to predict the availability of nodes, we first appeal to intuition. Nodes tend to be idle at predictable times of day. Users rarely sit at their workstations overnight. During lunch hour there may be a waning of activity. Certain users tend to use their computers consistently at the same time each day. Nodes tend to be idle on weekends and holidays. If a user has been working frantically for the past hour, it is more likely that he or she will continue to work than to suddenly stop. Guided by these observations, we can look for correlations between node activity from minute to minute, hour to hour, day to day, and week to week. As we will see, there is a significant correlation between historical usage patterns on all of these levels.

For scheduling Piranha jobs we are most interested in predicting the length of APs over the next n minutes where n represents the period of the scheduler. In this paper we focus on average AP length. Other quantities of potential interest are maximum and minimum AP length. For the data discussed below, maximum and minimum AP length do not diverge radically from average AP length

in the short term. Over windows of ten minutes, on average, maximum AP length exceeds average AP length by approximately 2%, and minimum AP length is 5% less than average AP length. We can thus expect prediction of maximum and minimum AP length to resemble average AP length for small n. For larger windows of sixty minutes, however, maximum AP length is on average 35% greater than and minimum AP length 19% less than average AP length. This implies that future study of minumum and maximum AP lengths for larger windows may be worthwhile.

Figures 1 to 3 show success rates of simple prediction strategies of average AP length over n future minutes, where n varies. Usage data was collected from fifteen SPARCStations in the Yale Department of Computer Science. Raw load average and idle time data were collected by David Kaminsky over the course of approximately three weeks. For the purposes of this analysis, a machine was determined to be available if its short-term, medium-term, and long-term system loads were under 0.7, 0.4, and 0.1, respectively. The machine's keyboard and mouse must also have been idle for at least 60 seconds. The graphs show the success rate for four prediction strategies in predicting average AP length over varying n. Fraction matched is the fraction of predictions which fell within a given tolerance of the actual average AP length.

Three of the strategies predict that the next n minutes will be exactly like some historical window of n minutes. REC_HIST (RECent_HISTory) is a strategy predicting that the next n minutes will be exactly like the last n minutes. For example, if the average AP length in the previous twenty minutes ($n = 20$) was four minutes, REC_HIST predicts that the average AP length in the next twenty minutes will also be four minutes. DTD_HIST and WTW_HIST are similar to REC_HIST except that they use different historical windows for prediction. DTD_HIST (DayToDay_HISTory) predicts that the next n minutes will be exactly like the historical window of n minutes at the same time the previous day. WTW_HIST (WeekToWeek_HISTory) predicts that the next n minutes will be exactly like the historical window of n minutes at the same time a week ago.

The two "BURST" strategies resemble common CPU burst prediction strategies.[8] L_BURST (Length_BURST) is a strategy predicting that the average AP length in the next n minutes will be the same as the length of the last AP. X_BURST (eXponential_BURST) predicts that the average AP length for the next n minutes will be an exponential average (with $\alpha = 0.5$, an equal weighting of history and most recent AP) of the previous AP lengths. AVAIL (always AVAILable) is included for comparison. It predicts that each node will always be available. This strategy, although apparently uninspired, is not a particularly bad, since workstations in such an environment are available the vast majority of the time (78% on average in this data set).

In general, the relative success of these different strategies seems to be consistent for a wide variation of parameters. Figure 1 is a representative sample of the simple strategies using a tolerance of 0.1. It seems that the best predictor of the next n minutes is what happened in the previous n minutes (REC_HIST) when $n \leq 10$, while what happened the previous day at that time (DTD_HIST) is best for $n > 10$. The same relation holds for a higher tolerance of 0.2, as can be seen in Figure 2. The poorest predictors are consistently AVAIL and WTW_HIST.

For a minimum tolerance of 0.0 (rounded to the nearest minute, predictions must equal reality), as we can see in Figure 3, the top two strategies remain at the top, but their crossover point has increased to about $n = 15$. AVAIL moves into third place, surpassing the presumably smarter strategies. It seems that scheduling algorithms which need to predict AP length will do best by using knowledge of recent history when n is small. This is not a surprising result, since it is common for CPU schedulers to successfully estimate the length of a job's next CPU burst based on the length of recent bursts.[8, p. 162] As n increases, average AP length tends to match AP length at the same time the previous day. For example, the best prediction of what will happen between noon and one o'clock today is that it will be just like what happened between noon and one o'clock yesterday.

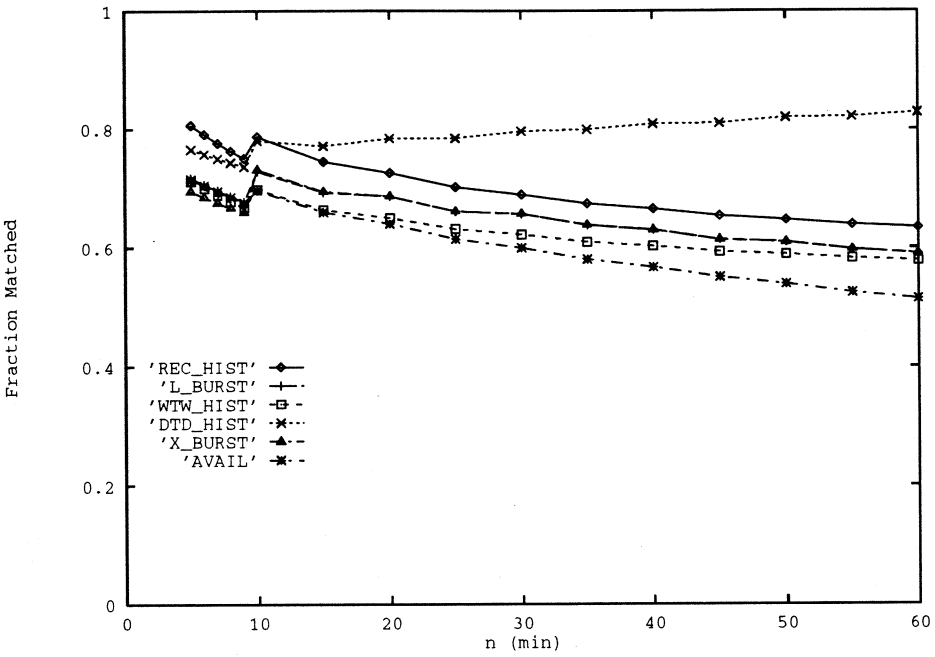

Figure 1: *Predictions of average AP length using simple strategies (tolerance 0.1)*

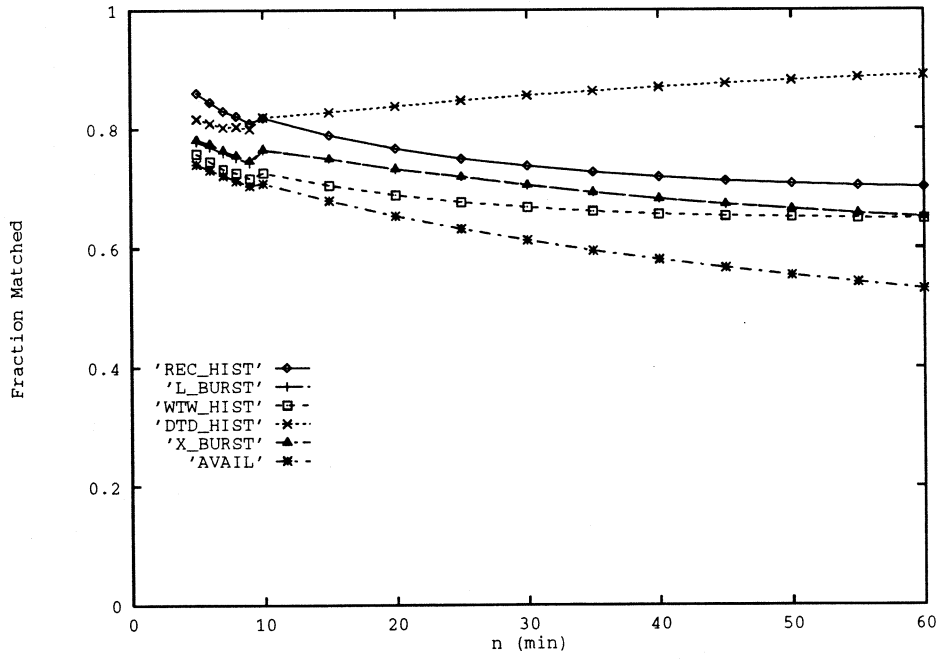

Figure 2: *Predictions of average AP length using simple strategies (tolerance 0.2)*

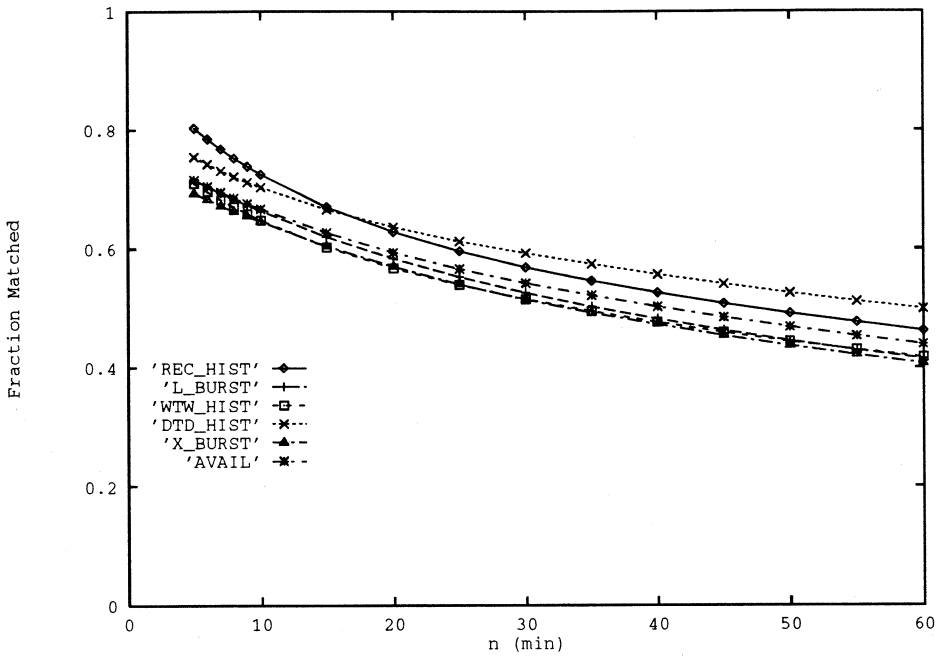

Figure 3: *Predictions of average AP length using simple strategies (tolerance 0.0)*

It is probable that more complex prediction algorithms can cleverly weigh week-to-week, day-to-day, and recent history information so as to improve on this simple prediction method. Our experience is that simple weighting of the different strategies does not work very well, but more intelligent combinations could be attempted. Such combinations might incorporate minimum and maximum AP lengths as well as average lengths. We can also envision smarter prediction strategies which take into account external calendar events, such as holidays and scheduled meetings, which draw users away from their workstations.

5 Task Time Prediction

In our attempt to match AP time to task execution time we must not forget that predicting task time can also be nontrivial. Since Piranha tasks are not restricted in what they are allowed to compute, in general it is impossible to correctly predict the amount of time it will take to execute an arbitrary Piranha task. It is also the case, however, that many Piranha applications involve executing a large number of nearly identical tasks. In such cases, the average execution time for previous tasks is a good predictor of the execution time of future tasks. The current Piranha implementation collects the average task time for each executing Piranha job and makes it available to the scheduler. This provides us with a reasonable, albeit imperfect, predictor of task times for a given job.

6 Scheduling Algorithms

Given our knowledge of AP and task time prediction, we now turn to the more practical problem of how to use this information in scheduling Piranha jobs. We want a scheduling algorithm which will minimize turnaround time for Piranha jobs, while also making efficient use of available processor time.

Turnaround time minimization is an important goal because it is in effect the response time to the user. A user submits a Piranha job because he or she wants a computation done more quickly than it could be done by more conventional (sequential, for instance) means. We must return results to a user as quickly as possible in order to make the use of Piranha worthwhile to the individual user.

6.1 Simple Priority Algorithms

If we had nodes which were perpetually available and thus we never had to worry about retreats, our scheduler would, ideally, use a Shortest Remaining Time (SRT) priority scheduling algorithm, since such an algorithm is provably optimal for turnaround time, assuming the applications scheduled scale well. The SRT algorithm requires accurate estimates of task lengths over the entire length of the job and a count of the number of tasks to be computed. Similar to SRT and also requiring complete knowledge about job length is the Shortest Processing Time (SPT) strategy. Job length estimates may be available for a limited class of Piranha programs which have a predetermined number of tasks and consistent task lengths. In general, however, the scheduler cannot determine how many tasks a given job will produce, and thus cannot estimate the remaining time the job needs with any accuracy. As a result, SRT and SPT are generally impractical strategies.

Even given perfect job length information, SRT can be less efficient than other strategies when the possibility of retreats is introduced. We can demonstrate this phenomenon with a simple example. Assume there are two jobs in the system, JOB1 with 100 tasks, each of length one time unit, and JOB2 with 20 tasks, each of length two time units. Assume for simplicity that there is only one node in the system and the scheduler is invoked every ten time units. Let us also assume that, for the first 200 time units, the node alternates between being available and unavailable. After this initial period, the node is continuously available. The SRT algorithm would schedule JOB2 first, since it has only 40 time units of computation left, while JOB1 has 100 units remaining. JOB2, however, would make no progress for the first 200 time units, because it would not be able to complete a task (we are assuming an interrupted task must be restarted from the beginning). After this JOB2 would continue uninterrupted. Then JOB1 would execute and complete. The average turnaround time in this situation is $(200 + 40 + 100)/2 = 170$ time units. If, on the other hand, we use a scheduler which assigns highest priority to the Piranha job with the shortest average task time, JOB1 will be scheduled first and finish after the first 200 time units (since 100 time units were available, one at a time) and JOB2 would finish 40 times units later, for an average turnaround time of $(200 + 40)/2 = 120$ time units. As a result, we can conclude that SRT is not always an optimal policy for scheduling Piranha jobs. The Shortest Task Time (TSK) alternative is an intriguing one, since it requires less information about the jobs it schedules than SRT, and can also do better in a chaotic environment of availability and unavailability.

In the absence of any job or task length knowledge, we may use a fair policy such as Round Robin (ROR) or Shortest Elapsed Time (SET). With these policies, as with SRT, shorter jobs tend to finish first. If all jobs are approximately the same length, a Longest Elapsed Time (LET) algorithm, or the similar First-Come First-Served (FCFS), will closely approximate SRT. Although these strategies can produce schedules with much worse turnaround time than SRT, they have major advantages. First, neither ROR, SET, LET, nor FCFS requires any knowledge of the execution times of the jobs they are scheduling. Second, ROR and FCFS do not allow starvation, while SRT may cause a particularly long job to starve indefinitely. The starvation problem can be alleviated by introducing aging to increase the priorities of long jobs as they wait, but this also compromises the optimality of SRT. ROR has the potentially troublesome attribute that its performance is extremely sensitive to the size of the quantum used. If the quantum used is poorly matched with task times, we could have a significant number of retreats caused by the scheduler itself. While CPU schedulers using ROR can resume

interrupted jobs from the point of interruption, a Piranha scheduler interruption can cause a job to lose progress already made on the current task. If the ROR quantum is not greater than or equal to all possible task lengths, two jobs with task lengths longer than the quantum could perpetually alternate and never make progress! A scheduler using ROR must be cognizant of this possibility.

Another possibility for scheduling is randomized algorithms. Two simple strategies are Random (RND) and Fair (FAIR). RND randomly chooses a job from the current job queue to be scheduled on a given node. A job is chosen for each individual node separately each time the scheduler is invoked. FAIR is a default strategy (along with FCFS) built into Kaminsky's current Piranha system.[7] If a node has no job scheduled on it, it is randomly assigned a job from the job queue. If a node already has a job scheduled on it and a new job enters the system, it switches to the new job with probability $1/n$, where n is the number of jobs in the new queue.

The randomized algorithms introduce the notion of dividing the processor pool between different jobs running concurrently. We can envision algorithms which, instead of dividing up the processor pool randomly, divide the pool by assigning jobs to nodes according to which jobs are expected to do best on which nodes. Such algorithms may be focused on different hardware, for example vector processors, available on some nodes and not others. A job involving a great deal of vector processing may have high priority on vector processors but not on other nodes. Another possibility is assigning jobs to nodes according to appropriateness of task length. A job with short tasks would be given high priority on a node with short APs, but a job with long tasks would be given high priority on a node with long APs. We explore this possibility in the next section.

6.2 Adaptive Algorithms

One possible adaptive strategy is the "fit" strategy, which explicitly tries to avoid retreats using the best prediction strategies outlined in Section 4. A fit version of SRT (SRTFIT) would prioritize jobs by the original SRT priority, but only schedule a job on a node if that node's expected average AP length over the next scheduling window exceeded the expected task time of the job. This would indicate that the job would be able to complete tasks within anticipated APs. If the highest priority job would not fit then the next highest priority job that did fit would be used. Again, a fit version of any priority algorithm could be created analogously, such as LETFIT for SETFIT.

In the previous section we saw an example showing that the TSK strategy may be an improvement over the SRT algorithm on a node which frequently switches from availabile to unavailable. We also know that in an environment where nodes are always available SRT is superior. We can use this observation to construct a strategy that takes advantage of the strengths of both simple strategies. We can use a Shortest Remaining Time Adaptive (SRTA) strategy which uses SRT unless a node crosses a threshold of availability (for instance, that the node is expected to be available less than 50% of the time over the next n minutes). When above the threshold, SRT is used for scheduling the node. When below the threshold, TSK is used. Similarly we could make other priority algorithms adaptive by combining them with TSK in the same way to come up with, for example, Longest Elapsed Time Adaptive (LETA). Node percent availability can be predicted using the same windowing strategies discussed in Section 4.

7 Scheduling Strategy Evaluation

In evaluating our scheduling strategies, we are faced with a choice between two approaches, analytic and empirical. The analytic approach requires a formal theoretical model to describe node availability patterns. Given our current understanding of node availability patterns, creating an accurate formal

Each node begins with ACTIVE[node]=0
For each time unit:

- A job may or may not enter the system job queue
- Each node has a job JOB[node] associated with it (-1 means no job scheduled) from the job queue by the scheduler. One job may be assigned to several nodes at the same time.
- For each node:
  - If JOB[node] is not the same job previously assigned to the node, ACTIVE[node] = 0
  - If AVAILABLE[node] then ACTIVE[node] = ACTIVE[node]+1 else ACTIVE[node] = 0
  - If ACTIVE[node] == JOB[node].taskTime then
    - JOB[node].timeRemaining = JOB[node].timeRemaining - JOB[node].taskTime
    - ACTIVE[node] = 0
  - If JOB[node].timeRemaining == 0 then JOB[node] is removed from the system job queue

Figure 4: *Piranha scheduling simulation algorithm*

model promises to be a difficult task. Alternatively, we can use actual node availability data we already possess and simulate Piranha scheduling to evaluate our strategies. The empirical approach is the more straightforward of the two and the approach we have chosen.

7.1 Simulation

In order to explore the relative utility of different scheduling algorithms for a global Piranha scheduler, we developed a simple model for simulating Piranha scheduling and compared the various scheduling algorithms mentioned previously. The simulator is written in C++. It runs sequentially simulating node availability patterns with the same node usage data used in the AP analysis and a randomly generated queue of artificial jobs. Each job in the system has associated with it a job *id*, a task time, and number of tasks. The same task time applies to all of a job's tasks, so that the total computational time required by a job is equal to the product of its task time and number of tasks. Job attributes are assigned randomly by a job feeder object according to program parameters. The schedulers map job *ids* to node *ids*. Jobs make progress according to the simulation algorithm described in the following section.

7.2 Simulation Algorithm

The algorithm used to simulate scheduling multiple Piranha jobs is shown in Figure 4.

ACTIVE[n] here reflects the number of time units for which node n has been available and working on the same task. *AVAILABLE[n]* is true if node n is available for Piranha computation at this time

unit. In the simulation, *AVAILABLE[n]* is filled using the node usage data mentioned earlier. Jobs make progress when they have been assigned to a node for an AP long enough for them to complete a task. If a job starts a task on an available node and does not finish before the node becomes unavailable, the computation done on that task is considered lost. Since the usage data was collected once every minute, the time step unit for the simulation was one minute.

7.2.1 Assumptions

Several assumptions are made to simplify the simulation. We assume scheduling decisions go into effect instantaneously. In a practical system, there is a delay between the time a scheduling decision is made and the time it becomes the actual state of the system. This delay includes a communication delay between the scheduler and piranhas, as well as delays caused by scheduler-invoked retreats. Delays caused by scheduler-invoked retreats cause potential computational time to be lost, as a node does no useful Piranha computation while a retreat occurs.

We also make simplifying assumptions about the nature of jobs. We assume all of a job's tasks take an equal amount of time. This may be an effective approximation for a large subset of Piranha applications, but not for all Piranha applications. We also assume that all work on a current task will be lost when a retreat occurs. This is a pessimistic assumption, as some real Piranha applications make use of partial results. Additionally, we assume that jobs scale linearly, i. e. tasks are consumed four times faster when executing on four nodes than when executing on one node. Although this is a good approximation for computationally-intensive Piranha jobs, it is not always valid for more communication-intensive applications, which tend to scale sublinearly.

7.3 Simulation Results

The simulator was run to evaluate several different scheduling strategies with varied parameters. Runs were done for different average job sizes and task lengths. Figures 5, 6, and 7 show representative data. Average turnaround is normalized by TSK turnaround time in each graph.

These graphs show the results of averaging twenty different simulator runs, each with a different random seed and each running twenty jobs through the simulator. Each job had a randomly generated number of tasks in the interval [1,1000] and a randomly generated task time in the interval [1,10].

Figure 5 compares the average turnaround time for several different scheduling strategies. Not surprisingly, SRT consistently produces the best turnaround time. Perhaps the most interesting result is that TSK convincingly beats all non-SRT-based algorithms. This is somewhat surprising because TSK makes no attempt to optimize for turnaround time. This would not be as surprising if TSK were scheduling in a highly variable environment, but the nodes used in this simulation were available on average 78% percent of the time. In the absence of any information about the jobs to be scheduled (i. e. no task or job length information) it seems the best strategies among those explored are FCFS and LET. It is important to note that some of these algorithms, most particularly ROR, but also others such as RND, can pay performance penalties if the scheduler is invoked too frequently. Frequent scheduler invocation can, as pointed out earlier, cause jobs with long task execution times to swap out before they have a chance to complete a task. It is possible that ROR could be made to perform appreciably better with a finer tuning of the rescheduling quantum. To be effective in general, however, such a tuning would have to be dynamic, as the set of jobs in a system at one point may correspond best to a different quantum than a set of jobs in the system at a later time.

Figure 6 shows a comparison of adaptive SRT algorithms (SRTA) run using different thresholds. Let T represent a threshold of predicted percentage of time the node will be available. When a node sinks below the T threshold, the scheduler will switch from SRT to TSK. Figure 6 shows results of

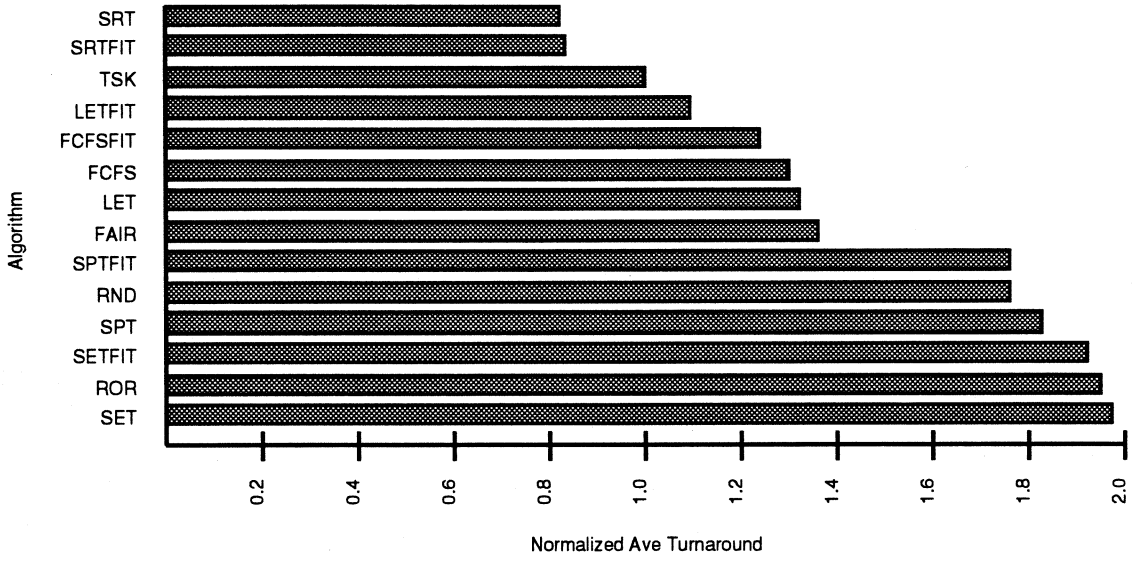

Figure 5: *Normalized average turnaround times for simulations of priority scheduling Algorithms*

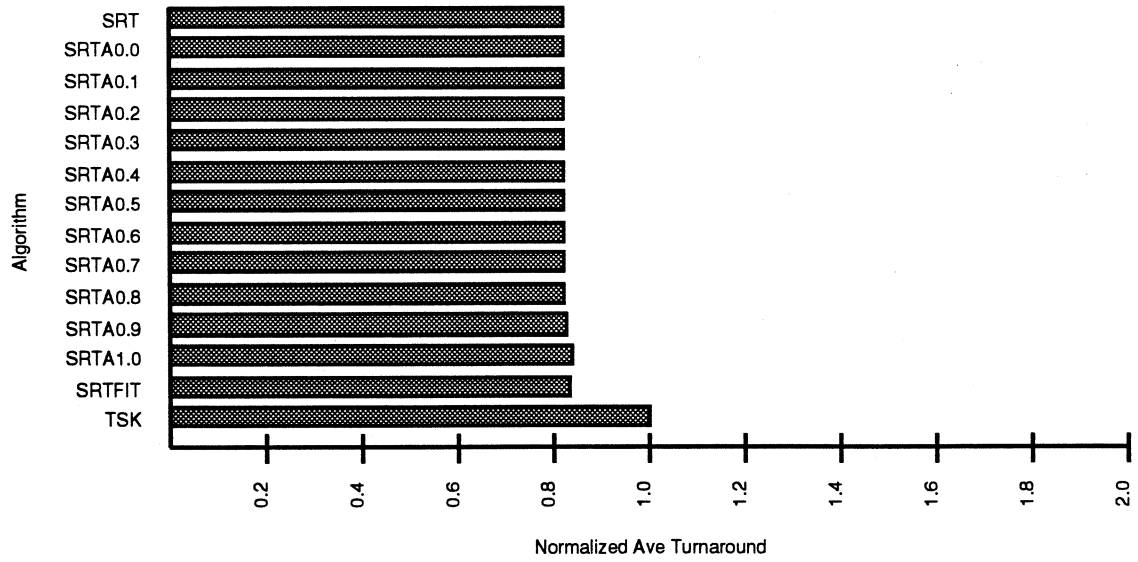

Figure 6: *Normalized average turnaround times for simulations of adaptive variants of SRT*

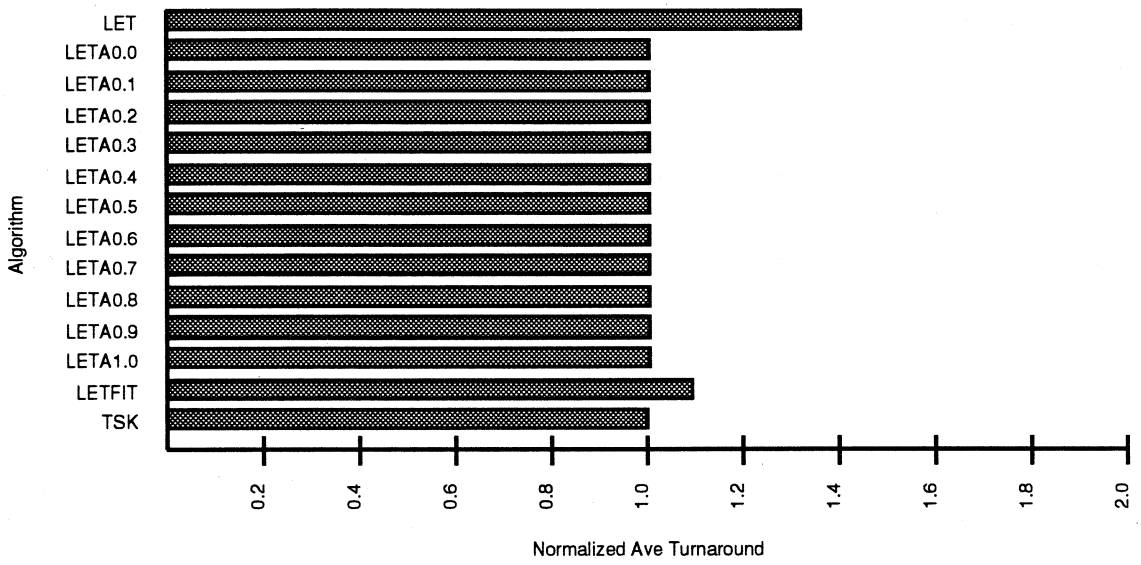

Figure 7: *Normalized average turnaround times for simulations of adaptive variants of LET*

simulating SRTA using the threshold values $T = 1.0$ (SRTA.1.0) to $T = 0.0$ (SRTA.0.0) by steps of 0.1. Also included for comparison are the simple version of SRT and the fit adaptive version SRTFIT. We can see that none of the algorithms is an improvement over the simple SRT algorithm. This is discouraging, since the point of introducing the adaptive versions was to become more efficient in the use of system resources. The more willing the strategy is to convert to TSK (the higher the threshold) the less efficient and more like simple TSK the strategy becomes. SRTFIT is almost uniformly less effective than the SRTA variants. Variaton of T has little effect.

Figure 7 shows an analogous plot for the adaptive variants of the LET algorithm. Unlike the SRT case, the adaptive variants here are a decided improvement over the simple LET strategy. Once again, the fit variant is inferior to the other adaptive variants. Although we see that using LETA with a high threshold is decidely better (a 24% improvement) than using unadaptive LET, it is also true that it simply approaches TSK's efficiency as the mix of LET and TSK in the strategy increasingly favors TSK.

Our attempts at adaptive strategies have been unsuccessful. The "FIT" variants do not do as well as the "A" variants, and the "A" variants simply drag a strategy closer to the performance of TSK. No adaptive strategy has proven to be an improvement over all of its constituent strategies. This indicates that we have not been able to apply the knowledge gathered for AP prediction to good effect. There are several possible explanations for this poor performance. First is our predictive capability. These adaptive strategies are predicated on our ability to predict future availability patterns. Although we demonstrated some success with AP length prediction in Section 4, we were still wrong nearly 20% of the time, even with the best strategies discussed. This inaccuracy in predictive capability handicaps the adaptive algorithms. Another possible explanation for the lack of success is the nature of the node availability patterns. The adaptive algorithms are meant to reduce retreats. The fact that the nodes studied were available for a large percentage of the time on average implies that retreats would be rare in any case. As a result, the relative gain from using adaptive algorithms could be small at best. This gives us a pessimistic outlook for predictive adaptive algorithms, but does not rule out

the possibility that different adaptive approaches could be successful. One such possibility is to run different strategies depending on time of day or calendar events.

We can draw some conclusions from the results presented so far about scheduling Piranha processes. The simulation results imply that even simple, knowledge-weak strategies such as TSK can do rather well compared to more knowledgeable strategies. As a result, we can feel confident that a simple TSK strategy will not do much worse than the impractical but successful SRT family of strategies. In the simulations, TSK did on average 22% worse than the best algorithm used, SRT. If task times for individual jobs do not vary widely, the average task time gathered by the system for a job will be a good predictor of future task times. As a result, the TSK strategy can be a practical strategy. If, however, we cannot predict task times with reasonable accuracy or facility, we effectively have no information about the jobs we are scheduling and must resort to less successful but fair algorithms such as FCFS and LET.

7.4 Implementing an External Scheduler

An external scheduler must gather the necessary information to make scheduling decisions and communicate those decisions to the Piranha system. Such a scheduler should monitor the nodes, collecting information necessary to implement adaptive scheduling policies. It should be modular, allowing us to add and remove different scheduling modules, node monitors, and even self-monitoring code. It should provide for the specification of the global policies we have been considering, but run as a distributed program in order to provide scalability. As mentioned previously, we have at our disposal a software architecture which allows us to specify the very sort of application we need to implement a Piranha scheduler, the process trellis architecture.

8 The Process Trellis Architecture

The process trellis software architecture was developed originally for use on shared memory parallel machines by Michael Factor.[3] The process trellis architecture was designed primarily to support the development of heuristic, real-time, parallel monitors. The trellis system provides a hierarchical, modular framework for the construction of parallel programs. Although the first uses of the system were for constructing monitoring applications, the trellis architecture is appropriate in general for the development of modular, flexible, predictable, and efficient parallel programs that fit a hierarchical decision-making model. The trellis architecture was extended by Jourdenais[6] to run effectively on networks of workstations and to incorporate new communication primitives which allowed the construction of *sensor-actuator* trellises.

A short introduction to the process trellis software architecture follows. See [3] for a more detailed presentation.

8.1 General Description

The trellis system organizes a collection of heterogeneous decision processes into a data analysis hierarchy. A process trellis program is described by a directed information flow graph. Each process in the trellis has a location in the hierarchy, defined by the processes named as its *inferiors*. Cycles are not allowed in the inferior-superior graph. Each process also has associated with it a state and a state calculation function used to maintain that state. State changes may occur only when a trellis node's inferior changes state, a superior explicitly *queries* the node, which forces it to recalculate state, or an external entity executes a *probe* which instructs the node to recalculate state. Superiors

have knowledge of the states of their inferiors, so data flows in this way up the trellis hierarchy. The restricted circumstances under which a trellis node may change its state or data may flow through the trellis allow the programmer to easily determine which nodes may affect the state of which other nodes, and thus support the construction of large, modular systems.

The trellis communication protocol is constructed so as to minimize costly nonlocal communication and to allow prediction of actual running times of arbitrary trellis programs. This results in trellis applications which run efficiently and predictably. An analytic model incorporating the communication protocol makes it possible to heuristically schedule processes so that any process can execute when necessary and thus obey soft worst-case real-time constraints.

The programmer using the trellis architecture need not concern himself or herself with the details of explicit parallelism or synchronization while programming an application. These issues are handled implicitly by the trellis system.

8.2 The Network Trellis System

The Network Trellis System described in [6] extends the original trellis system in two significant ways. First, this system allows the construction of trellis programs for execution on networks of workstations and provides mechanisms for prediction of performance on that platform. Second, the Network Trellis System allows the construction of *sensor-actuator* trellises by introducing the notion of *commands* to the trellis communication protocol. Commands are generalizations of queries, in that like queries they cause activation of inferiors, but unlike queries they may carry with them arbitrary data from superior to inferior. Commands allow a symmetric flow of information up and down an application's trellis structure and thus enable support of both monitoring (sensor) and control (actuator) applications where before only monitoring applications were supported.

9 The Network Piranha Scheduler

The Network Piranha Scheduler (NPS) is the first real application of the Network Trellis software architecture. To implement a global Piranha scheduler we need a program structure which is suited to efficient monitoring of a network of workstations and also capable of fusing the gathered information in a straightforward, modular way. Since the Network Trellis System now allows us to construct trellis programs residing on networks of workstations, low level data-gathering processes in a trellis program can reside on the nodes that are the sources of their data. We also need support for relaying scheduling decisions down the trellis structure from the high level decision processes to the bottom level processes residing on target processors. The sensor-actuator extensions provide this support.

We will first describe the structure of a fully functional NPS trellis and then discuss the implemented prototype NPS, which incorporates the most vital subset of this functionality.

9.1 Description

Figure 8 shows the structure of the complete NPS. The NPS has six major types of processes. There are *monitor* processes which gather availability data, one to a physical node. These same processes also inform the Piranha system of scheduling decisions for their respective nodes. There is an *enabler* process which activates the monitor processes at regular intervals to cause them to collect new data from the local system. One or more *scheduler* processes are, of course, essential. Each scheduler process makes mapping decisions based upon job descriptors received from the Piranha system and system availability information received from the monitor processes. The scheduler processes send commands

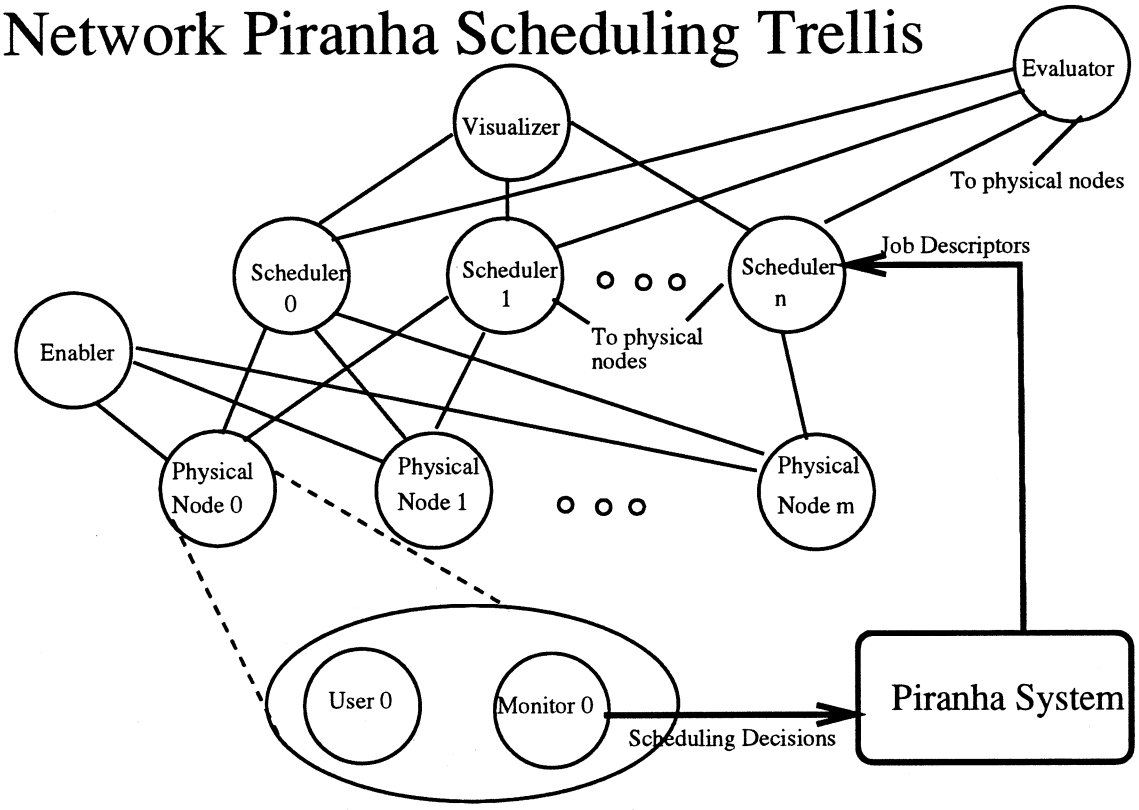

Figure 8: *The Network Piranha Scheduler (NPS)*

to monitor processes instructing them which Piranha job should be assigned to their respective nodes. We may wish to have more than one scheduler process available to enable us to switch strategies quickly or to run different scheduling strategies on different subsets of physical nodes. We have one or more *evaluator* nodes which keep comparative data on the scheduling decisions of the various scheduler processes, evaluating them in the context of usage patterns in the system. Evaluator nodes may also be useful for switching the system to a new scheduling strategy if the current strategy is ineffective. At the same level as the evaluators it is desirable to have one or more *visualizer* processes to show graphically how jobs are being assigned in the system and which physical nodes are being used at any given time. Also desirable are bottom level *owner* processes which are distributed one to a physical node and keep track of the preferences of the owner or administrator of the physical node. Via the owner processes, users could make their nodes unavailable to Piranha processes for certain periods of time, or submit complaints about the Piranha system for later perusal by a human or computer administrator. Note that any of the nodes discussed so far could be subdivided into a hierarchy of nodes for more complex decision-making. For example, the owner process could feed into a hierarchy of classification nodes to sort, evaluate, and react to various user complaints; the monitor processes could feed into a collection of data filtering processes which convert raw data into a form more immediately useful to the schedulers; and the scheduler could be subdivided into a hierarchy of decision-making routines.

Because of the modularity of the trellis architecture, it is a simple task to add or remove multiple scheduler, evaluator, visualizer, and monitor nodes with minimal impact on the remainder of the application. This allows for quick and easy modification of the application as it develops from a

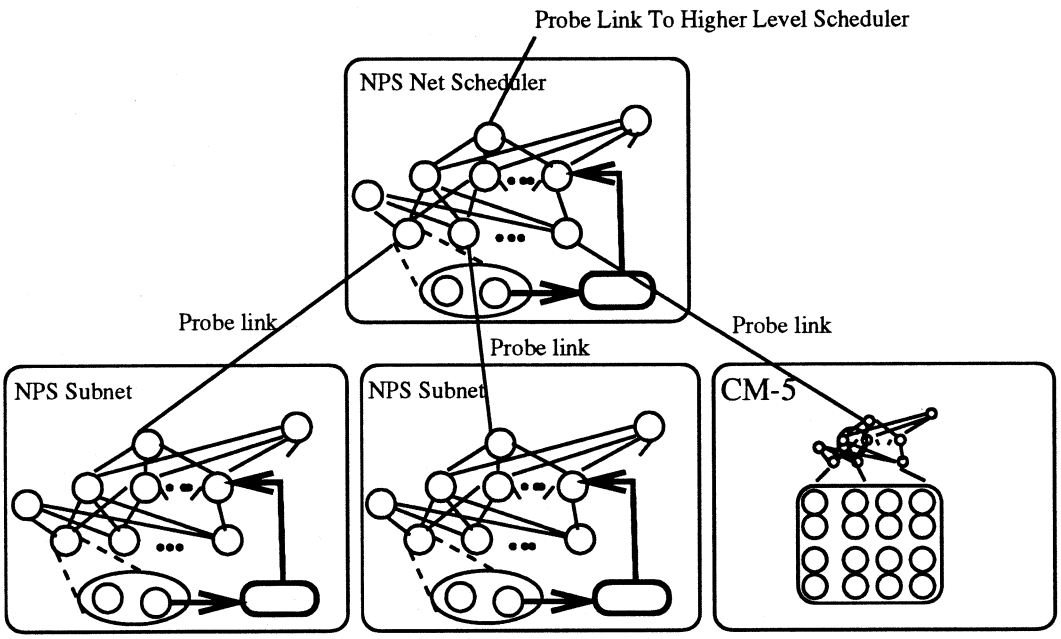

Figure 9: *The Hierarchical Network Piranha Scheduler*

prototype into a practical system. It also allows for easy visualization of the system as it changes.

The entire NPS trellis described above, like any trellis, can be viewed as a module in a larger trellis. One can imagine having separate NPS trellises for scheduling tasks on individual subnets in an academic department. These local trellises are connected by probes to a higher level scheduling trellis which can communicate with the local scheduler and evaluator processes to move jobs from congested subnets to less congested subnets or to locate specialized hardware (such as vector coprocessors) which may be appropriate for a job but not available in the subnet to which it was submitted. At higher levels, NPS trellises could conceivably link academic departments at a university, forming an institution-wide scheduling system. They may even link separate universities. Local NPS trellises could run with a relatively fast invocation period to schedule local jobs. Higher level NPS systems could run at a relatively slower rate, occasionally becoming active to determine if any large-scale adjustments are desirable. We can also anticipate having future NPS implementations run on more exotic machines, such as the CM-5, for which Freeman has already developed a prototype Piranha system[1], thus linking widely different types of machines into the same processor pool. Figure 9 shows the possible trellis structure of such a system. Although many issues such as heterogeneity and fault tolerance remain to be worked out before such a system is viable, a system providing the necessary support is a logical extension of our current work with the Network Trellis System.

The prototype NPS schedules Piranha jobs on a group of four (arbitrary) networked RS6000 machines. It has the essential enabler, monitor, and scheduler processes implemented. Owner processes are incorporated into the trellis, but at this time make no functional contribution. Multiple scheduler processes can be run, but the monitor processes will only obey one scheduler process (the one

with scheduler $id = 0$). Specification of strategies for the scheduler processes is done via parameter files. An evaluator node exists which keeps track of estimated turnaround time and system usage by each scheduler, as well as statistics on each physical node's availability. The monitor processes send scheduling decisions to the Piranha system via a system-wide open tuple space[7] which allows loose coupling of the Piranha and NPS systems. There are two major reasons why we chose open tuple space for communication. First, the Piranha system already uses open tuple space for internal communication. It was straightforward to mesh the NPS with the Piranha system using existing communication channels. Second, use of open tuple space allows for a loose coupling of the two systems. The systems do not depend upon shared file or socket names which may be difficult to communicate at run time. They also execute asynchronously. This loose coupling provides flexibility and modularity. The visualizer node has not been implemented at this time.

9.2 Experimental Results

We have shown earlier in this paper that the TSK scheduling algorithm will likely give us a substantial improvement over the native Piranha system schedulers. The question remained, however, whether this theoretical gain in scheduling policy could overcome the overhead of operating a loosely coupled external scheduler in place of the internal scheduler.

We could anticipate loss of efficiency in several ways. First of all, the NPS is an additional program running on the same nodes used by the Piranha system and thus increases load on those nodes and effectively steals cycles from the piranhas themselves. Additionally, the simulations assumed more accurate knowledge of task attributes. In the simulations, it was assumed that average task time information was available for the TSK scheduler. Although in reality the Piranha system does provide average task time statistics, it takes time for the Piranha system to accumulate these statistics. In the current implementation, the NPS is blind with respect to task times (they default to zero) until a job has run for about thirty seconds. Since the NPS scheduling with the TSK strategy sees a job's task time as zero until the job executes for thirty seconds, the strategy turns into a de facto round robin with quantum thirty seconds until all tasks have accumulated a nonzero listing for average task time. This will cost the scheduler in efficiency, as having task time knowledge immediately upon seeing the job for the first time would result in a purer implementation of the TSK strategy. In the real world, there is a nonzero delay between the time the scheduler recognizes the need for a scheduling change and the time the Piranha system reacts to its scheduling messages. Also, depending on the period of the scheduler, some time may be wasted between a change in the job queue an the invocation of the scheduler to react to that change. Additional delay is incurred as each node with a scheduling change must retreat.

In an effort to determine whether the use of an external scheduler incurred undue overhead on the execution of Piranha jobs, we ran timings to compare the execution time of one job with and without the external scheduler running. The presence of only one job removes the influence of differences in scheduling strategies.

The cmatrix program, a Piranha program which performs a matrix multiply, was used in the following experiments. Cmatrix was written by David Kaminsky and Nick Carriero. Each cmatrix task multiplies one row of the first matrix with one column of the second matrix. The task time and the number of tasks thus increases as the dimension of the matrices increases.[2] The program is run with the dimension of the square matrices as an argument.

For this experiment the turnaround time of cmatrix run with dimension 500 was collected for repeated runs with the external NPS scheduler on and with the external scheduler off. The NPS

[2]Cmatrix does not use the most efficient algorithm available, but it is useful for purposes of demonstration.

scheduler was run with a period of ten seconds. Runs were done during a period of low activity on four IBM RS6000/560 machines so that the machines were generally available for Piranha computation. On average the use of the external scheduler caused a noticeable degradation in performance relative to the internal scheduler. With the external scheduler on, average turnaround time was 581.7 $\pm$ 53.7 seconds. With the external scheduler off, average turnaround time was 530.5 $\pm$ 16.5 seconds. In an effort to determine whether the overhead incurred with the external scheduler was primarily due to the cycles being stolen by the NPS or to the communication of scheduling decisions throughout the Piranha system, an external scheduler which ran on a node separate from the computation nodes was also timed. The external non-intrusive scheduler had an average turnaround of 559.7 $\pm$ 49.7 seconds. Although not decisive, these numbers imply that the system is incurring overhead from both the computational demands of the NPS on the Piranha nodes and the communication mechanisms.

To evaluate the net gain of using the external scheduler over the internal scheduler, we repeatedly ran a batch of cmatrix jobs with different dimensions through the Piranha system. When on, the external scheduler used the TSK strategy. With the external scheduler off, the native FAIR randomized strategy was used. A batch of four cmatrix jobs was submitted at the beginning of a trial. JOB1 ran with dimension 300, JOB2 with dimension 400, JOB3 with dimension 500, and JOB4 with dimension 600. Turnaround times were averaged over the trials.

Figure 10 shows the results of the batch experiment. EXT(all) refers to the average turnaround of all jobs over all trials using the external scheduler. EXTn refers to the average turnaround of JOBn over all trials using the external scheduler. Analogously, the NAT(all) and NATn entries refer to the same averages using the native scheduler instead of the external scheduler. As can be seen in the graph the external NPS did provide a net improvement overall in turnaround time with respect to the native scheduler. The turnaround times for the individual jobs can be seen to reflect the policies under which they were scheduled. Under the TSK policy, shorter jobs finished first and turnaround times scaled with the cube of the matrix dimension. Under the randomized strategy, as expected, turnaround times were clustered more closely around the overall average. We can see that the NPS is a viable system for scheduling Piranha jobs with potentially improved net turnaround performance.

10 Future Work

The problem of scheduling Piranha jobs touches on many issues relevant to distributed systems, including the modeling of usage patterns of nodes in the system and the efficient use of idle resources. We have presented here some simple methods for predicting usage patterns. More complicated algorithms remain to be discovered and tried. Determining which scheduler is best for scheduling Piranha jobs in a network environment is a task with many possibilities and parameters to consider. We have presented here an empirical analysis of a limited set of these possibilities. Another approach, the development of a formal theoretical model and an attempt to prove optimality of schedulers within that model, is undoubtedly worth consideration.

Piranha is a young system, and the Network Piranha Scheduler a mere infant. The NPS must grow in order to become a generally useful system, and Piranha, which has already proven useful, must encompass more users and applications before large scheduling problems will even appear in actual use. The communication interface between Piranha and the NPS and the functionality of nodes in the NPS could both stand improvement. A convenient user interface, including the visualizer node, must also be incorporated into the NPS before it becomes a practical administrative tool. The NPS has been shown to be a viable system capable of improving on the native Piranha scheduler. There is, however, room for improvement in streamlining the implementation of the NPS to reduce computational and communication overhead.

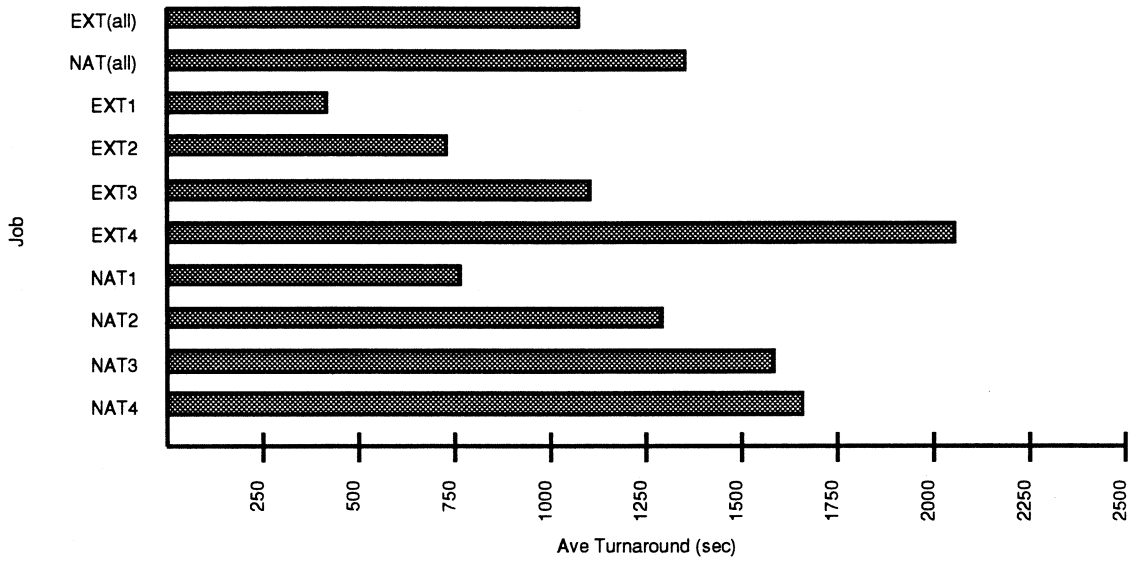

Figure 10: *Average turnaround times for batch experiment*

11 Conclusion

We have explored several possible simple strategies and some more complex adaptive strategies for scheduling Piranha jobs on a network of workstations. Using simple prediction strategies for determining node usage patterns, the adaptive strategies try to appropriately match Piranha jobs to nodes with compatible availability patterns. The scheduling strategies were evaluated through simulations in which simulated workstations exhibited real availability patterns. It was found that the adaptive strategies examined were not improvements over the simpler constituent strategies. Of the practical strategies, priority scheduling by shortest task time was found to be most effective in general. Although the "smarter" adaptive strategies were not effective, we still believe there is potential for better (possibly substantially more complex) adaptive strategies to take advantage of node availability patterns and better allocate resources in the Piranha system. The exploration of such strategies remains a topic for future research.

We have found the process trellis software architecture, particularly the Network Trellis System, to be an excellent vehicle by which to implement an external Network Piranha Scheduler. Because of its modularity and programming simplicity, the Network Trellis System allowed us to quickly generate a distributed Network Piranha Scheduler whose structure is easily described and modified. The trellis architecture also provides for the possibility of creating a hierarchy of network schedulers and thus a scalable scheduling system. A wide-area Piranha system supported by a hierarchy of Network Piranha Schedulers is a reachable future goal.

Preliminary results show that the NPS is a viable system, at least on a small scale, and that the NPS can provide improvements over the native scheduling system.

12 Acknowledgements

The authors would like to thank Nick Carriero for supplying helpful guidance and Michael Factor for laying a solid foundation with the original work on the trellis system.

References

[1] Nicholas Carriero, Eric Freeman, and David Gelernter. Adaptive parllelism on multiprocessors: Preliminary experience with Piranha on the CM-5. Technical Report YALEU/DCS/RR-969, Yale University Department of Computer Science, May 1993.

[2] Michael Factor, David Gelernter, and Dean F. Sittig. Multiple trellises and the intelligent cardiovascular monitor. Technical Report YALEU/DCS/TR-847, Yale University Department of Computer Science, February 1991.

[3] Michael E. Factor. *The Process Trellis Software Architecture for Parallel, Real-Time Monitors.* PhD thesis, Yale University, December 1990.

[4] David Gelernter and David Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with piranha. Technical Report YALEU/DCS/RR-883, Yale University Department of Computer Science, December 1991.

[5] B. A. Huberman, editor. *The Ecology of Computation.* North-Holland, Amsterdam, 1988.

[6] Marc Jourdenais. Extending the process trellis software architecture to distributed environments. Yale CS690/691 Research Project Report, May 1993.

[7] David Kaminsky. Personal communication, July 1993.

[8] Abraham Silbershatz and James L. Peterson. *Operating System Concepts.* Addison-Wesley, alternate edition, 1988.

[9] Carl A. Waldspurger et al. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2), February 1992.