Implications of I/O for Gang Scheduled Workloads

Walter Lee, Matthew Frank, Victor Lee, Kenneth Mackenzie, and Larry Rudolph [⋆]

M.I.T. Laboratory for Computer Science
Cambridge, MA 02139, U.S.A.
{walt, mfrank, wklee, kenmac, rudolph}@lcs.mit.edu

Abstract. The job workloads of general-purpose multiprocessors usually include both compute-bound parallel jobs, which often require gang scheduling, as well as I/O-bound jobs, which require high CPU priority for the individual gang members of the job in order to achieve interactive response times. Our results indicate that an effective interactive multiprocessor scheduler must be *flexible* and tailor the priority, time quantum, and extent of gang scheduling to the individual needs of each job.

Flexible gang scheduling is required because of several weaknesses of traditional gang scheduling. In particular, we show that the response time of I/O-bound jobs suffers under traditional gang scheduling. In addition, we show that not all applications benefit equally from gang scheduling; most real applications can tolerate at least a small amount of scheduling skew without major performance degradation. Finally, we show that messaging statistics contain information about whether applications require gang scheduling. Taken together these results provide evidence that flexible gang scheduling is both necessary and feasible.

1 Introduction

While gang scheduling provides better performance than uncoordinated scheduling for compute-bound parallel jobs with frequent synchronization, it leads to poor performance of I/O-bound jobs that require short, high-priority bursts of processing. Rather than religiously following the gang scheduling paradigm, a scheduler for a parallel computer should be *flexible*, capable of delivering acceptable performance for both compute and I/O-bound jobs.

Uniprocessor schedulers assign I/O-bound jobs higher priority than compute-bound jobs in the hope of reducing the average response time and without decreasing the machine utilization. It is well known that scheduling shortest job first minimizes the average response time. Traditional multiprocessor gang schedulers, on the other hand, schedule jobs in a strict round-robin fashion and ensure

⋆ This research is funded in part by ARPA contract # N00014-94-1-0985, in part by NSF Experimental Systems grant # MIP-9504399, in part by a NSF Presidential Young Investigator Award.

that each member of a gang be allocated a processor at the same time. The CPUs allocated to gang members that perform I/O often sit idle. The disks allocated to gang members I/O requests often sit idle until the member gets a chance to execute again.

In effect, the presence of I/O-bound jobs complicates scheduling decisions by exerting pressure on the system not to gang schedule. The pressure comes in two forms. The first comes from the opportunity to improve response time by interrupting gang scheduled, compute-bound jobs in order to execute a I/O-bound job. The second comes from the need to schedule fragmented cpu resources. These features of flexible gang scheduling motivate the need to find out how compute-bound jobs benefit from gang scheduling. We study this issue and show that many applications can in fact tolerate the nearly gang environments provided by a flexible gang scheduler.

The studies of both I/O-bound jobs and compute-bound jobs demonstrate that flexible gang scheduling can be an improvement over traditional gang scheduling. Central to the realization of a flexible gang scheduler is the ability to determine dynamically the level to which individual applications benefit from gang scheduling. We show how one can extract such information from raw messaging statistics.

Although gang scheduling improves the performance of many workloads, it conflicts with the goal of providing good response time for workloads containing I/O-bound applications. The results in this paper motivate the need to analyze the costs and benefits of gang scheduling each job by showing that gang scheduling jobs increase the response time of I/O-bound applications and by showing that some jobs benefit only marginally from a dedicated machine abstraction. In addition, we show that a scheduler can collect the necessary information for a cost-benefit analysis from raw messaging statistics.

The rest of the paper is organized as follows. Section 2 describes our experimental environment. Section 3 studies the impact of gang scheduling on I/O-bound applications. Section 4 studies the performance of compute-bound applications in near-gang scheduled environments. Section 5 explores the use of messaging statistics to aid scheduling decisions. Finally, Section 6 and Section 7 present related work and conclude, respectively.

2 Experimental Setup

In this section, we describe the experimental environment used in Sections 4 and 5. The environment also provides the basis for the more abstract simulation models used in Section 3. We provide information about the Fugu multiprocessor, the scheduler, and the multiprocessor simulator used by the experiments.

Fugu is an experimental, distributed-memory multiprocessor supporting both cache-coherent shared memory and fine-grain message passing communication mechanisms [13]. The applications studied in this paper use only the message-passing mechanism. Messages in Fugu have extremely low overhead, costing roughly 10 cycles to send and roughly 100 cycles to process a null active message via an interrupt. The Fugu operating system, Glaze, supports virtual memory, preemptive multiprogramming and user-level threads. The message system is novel in that messages received when a process is not scheduled are buffered by the operating system at an extra cost.

The Fugu scheduler is a distributed application organized as a two-level hierarchy with a global component and local, per-processor components. The cost of the global communication and computation is amortized by pre-computing a *round* of several time-slices of work which is then distributed to the local schedulers. Results for this paper employ a four-processor configuration running small workloads, so the cost of the global work is small and the round size is kept minimal. The scheduler uses an Ousterhout-style matrix coscheduling algorithm to assign work to processors. Jobs have fixed processor needs and are assigned to processors statically, one process per processor, at the time the jobs begin. Each job is marked with a *gang* bit that indicates to the scheduler whether constituent processes may independently yield their time-slices when they have no work to do.

Experiments are run on an instruction-level simulator of the Fugu multiprocessor. The simulator counts instructions, not strictly cycles. Since the scheduling issues we are interested in are orthogonal to any memory hierarchy issues, we believe instruction counts will give us the same qualitative results as cycle counts.

3 Gang Scheduling and I/O Jobs

In this section we study the implications of gang scheduling in the presence of I/O-bound jobs. We find that the requirements of gang scheduling lead to a tradeoff between disk utilization and cpu utilization. Traditional uniprocessor schedulers, based on multilevel feedback queues, manipulate job priorities to effectively overlap disk requests with processing. Because gang schedulers ignore information about job behavior, they make suboptimal choices which lead to slowdowns for both I/O-bound and compute-bound jobs.

Section 3.1 discusses a variety of ways in which gang scheduling can lead to poor I/O and cpu utilization. Section 3.2 demonstrates the tradeoffs that gang scheduling must make between I/O and compute-bound jobs. Our results suggest that gang schedulers require considerable information to make good decisions. Along with the priority information collected by traditional uniprocessor schedulers, a gang scheduler can benefit from knowledge about the coscheduling requirements of compute-bound jobs.

3.1 Costs of Gang Scheduling

The costs of gang scheduling can be divided into two categories, under-utilization of disk resources, which we call *priority inversion*, and under-utilization of cpu resources, which we call *cpu fragmentation*. Disk resources can best be utilized if processes of I/O-bound jobs are given priority to use the cpu whenever they

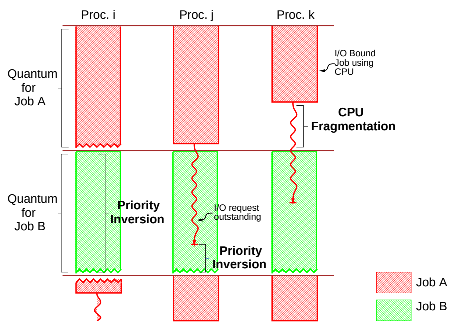

Fig. 1. Adverse effects of gang scheduling in the presence of I/O. In processor `i` (left), the process for job A reaches the end of its quantum before it is able to issue an I/O. The disk is left idle for the entire duration of the quantum for job B. In processor `j` (middle), an I/O request from job A finishes before the end of quantum B. The higher priority, I/O-bound process must wait till the end of job B's quantum. When job A makes a request before the end of its quantum (processor `k`, right), it leaves behind fragmented CPU resources.

are ready to run. This policy ensures that a process's next I/O request will come as soon as possible after the previous one finishes. Note that it is the thread or process, not the job, that makes an I/O request. When a job consists of multiple threads or processes, it is likely that only a subset of them will block on an I/O operation. Since gang schedulers schedule whole jobs, they cause priority inversion problems whenever they permit a compute-bound job to use the cpu while processes of an I/O-bound job is ready to run.

There are two different causes of priority inversion. Either the scheduling quantum length for an I/O-bound job can be set too short, or the scheduling quantum length for a compute-bound job can be set too long. The left hand side of Figure 1 demonstrates the first of these problems. Here, the quantum for job A, an I/O-bound job, ends shortly before process `i` of job A is ready to make an I/O request. The disk sits idle for the entirety of quantum B before job A is permitted to resume. If quantum A had been slightly longer, a disk access could have been overlapped with job B's computation.

A second form of priority inversion occurs when the scheduler sets the quantum length for a compute-bound job too long. This problem is shown in the middle part of Figure 1. In this case, process `j` of job A makes an I/O request. Shortly afterward, job A's quantum expires and the scheduler switches to running job B. When the I/O request finishes, the scheduler does not return to job A because job B's quantum has not yet finished. The time remaining in the quantum is devoted to the compute-bound job, which unnecessarily delays the occurrence of the next I/O operation from job A.

In contrast, the right hand side of Figure 1 demonstrates the cpu fragmentation problem that occurs when the quantum for an I/O-bound is too long. In this case process `k` of job A makes an I/O request considerably before the end

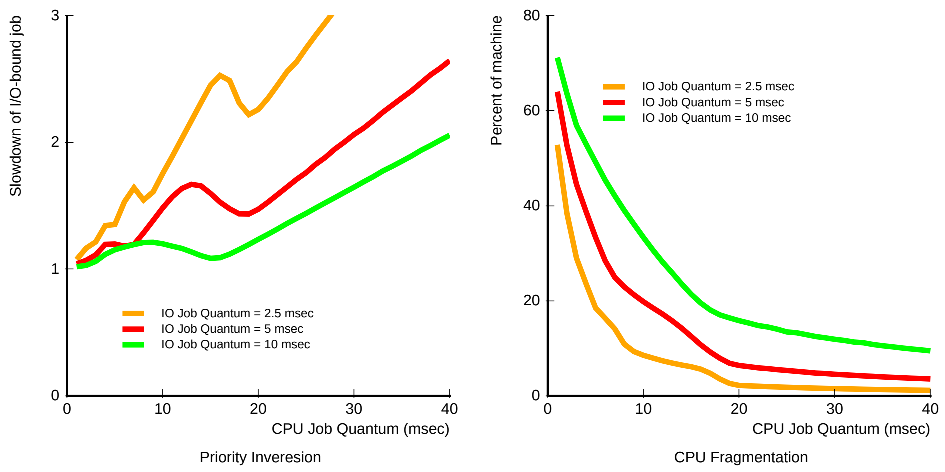

Fig. 2. Increasing the cpu-bound job's quantum length increases priority inversion but reduces cpu fragmentation. An I/O-bound job and a compute-bound job are scheduled against each other on a 32-processor, gang-scheduled machine. Each process of the I/O job uses the CPU for an average of 5 msec between making 20-msec I/O requests. Three experiments are shown, with the scheduler quanta for the I/O-bound job set to 2.5, 10, and 20 msec. The scheduler quanta for the compute-bound job is varied on the X axis for both graphs. The left graph plots the level of priority inversion, represented as a slowdown factor of the I/O-bound job as compared to running the job on a dedicated machine. The right graph plots the amount of cpu fragmentation as a percentage of the total available cpu resources.

of quantum A. Because job B requires gang scheduling, it is unable to make progress because the rest of the processors are still running processes of job A. Processor `k` remains idle until the beginning of quantum B.

The next subsection examines these issues quantitatively and finds that dealing with priority inversion requires that the quanta be allocated dynamically to suit the I/O requirements of the workload. A more flexible scheduling scheme can deal with the problems of priority inversion and resource fragmentation by allowing the characteristics of each job to drive the schedule.

3.2 I/O-CPU Utilization Tradeoffs

By varying the quantum length for different jobs, the effects discussed above can be observed. In particular, priority inversion, which causes poor disk utilization, occurs when either the quantum length for an I/O-bound job is too short or when the quantum length for a compute-bound job is too long. Cpu fragmentation, which causes poor cpu utilization, occurs when the quantum length of the I/O-bound job is too long.

Because a variable quantum policy requires considerably more flexibility than

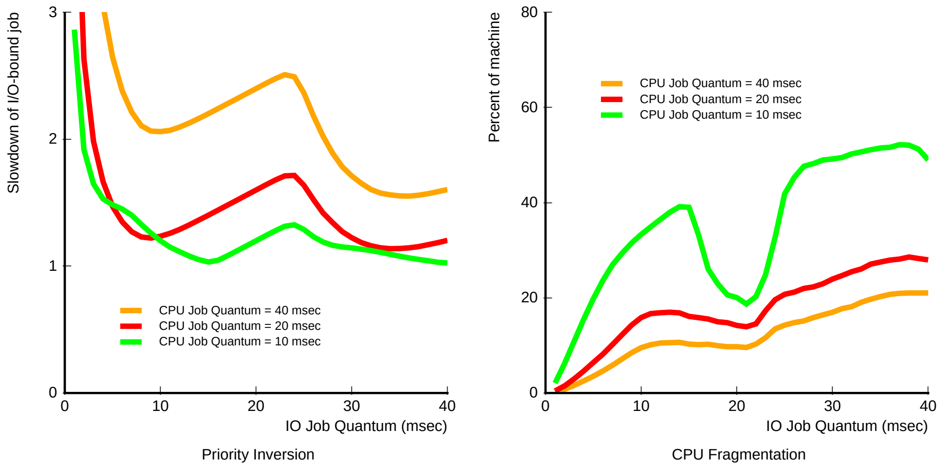

Fig. 3. Increasing the I/O-bound job's quantum length generally reduces priority inversion but increases cpu fragmentation. The workload parameters are identical to those in the last figure. Three experiments are shown, with the scheduler quanta for the compute-bound job set to 10, 20 and 40 msec. For both graphs, the scheduler quanta for the I/O-bound job is varied on the X axis. The left graph plots the slowdown of the I/O-bound job, which reflects the level of priority inversion; the right graph plots the amount of cpu fragmentation.

is traditionally available in gang schedulers, the experiments reported in this section were run on a simple event-driven simulator. The experiments consist of gang scheduling a synthetic I/O-bound job against a synthetic compute-bound job. Like a traditional gang scheduler, the scheduler in the experiments switches back and forth between the two jobs in a round-robin fashion; however, the quantum lengths for the two jobs are not required to be the same, and in fact they are varied across different runs of the experiment. When a process for the I/O-bound job is blocked on an I/O operation, its remaining time quantum is donated to the process of the cpu-bound job.

The I/O-bound job alternates between short bursts where it requires the cpu and I/O requests where it simply waits for a disk request to finish. Its cpu time is modeled by an Erlang-5 distribution, which resembles a normal distribution, with a mean of 5 msec. The latency of I/O requests is fixed at 20 msec. The compute-bound job makes no I/O requests, and it represents a job with heavy synchronization so that it makes progress only when all its processes are scheduled simultaneously.

We vary the gang scheduler quantum allocated to each of the two jobs, and we infer the level of priority inversion and cpu fragmentation by observing the slowdown for each job, defined to be the ratio of the run times of the job when it is run in the experimental environment versus when it is run in a dedicated machine. Priority inversion relates directly to the slowdown of the I/O-bound job. The greater the priority inversion, the higher the slowdown of the I/O-bound job. CPU fragmentation is computed by subtracting the amount of useful work done by the cpu-bound job from the amount of cpu resources allocated to it. The results are shown by the two pairs of plot in Figures 2 and 3.

In the first experiment, three different settings – 2.5 msec, 5 msec, and 10 msec – are used for the quantum length for the I/O-bound job. The quantum length for the compute-bound job is varied from 1 msec to 40 msec. Figure 2 shows the results. In general, as the quantum length of the compute-bound job is increased while the quantum length of the IO-bound job is held constant, the level of priority inversion increases and the level of cpu fragmentation decreases. This behavior can readily be explained in terms of the proportion of resources allocated to the I/O-bound job. As the quantum length for the compute-bound job increases, the I/O-bound job gets a smaller share of the cpu. This change in ratio causes an increase in priority inversion and leads to a degradation in performance of the I/O-bound job. At the same time, the decreasing share of cpu allocated to I/O-bound job reduces cpu fragmentation because fragmentation only occurs during the scheduling of I/O-bound jobs.

Most of the curves for both graphs in Figure 2 follow the monotonic trend expected from the explanation in the previous paragraph. The "waviness" in the priority inversion plot, as well as the bumps in the cpu fragmentation plot, are a result of the harmonics between the periodicity of the I/O-bound job (at a frequency of about 25 msec) and the scheduling quanta.

Figure 3 presents the result of the second experiment, where the quantum length of the I/O-bound job is varied from 1 msec to 40 msec while the quantum length for the compute-bound job is fixed at either 10 msec, 20 msec, or 40 msec. The general results can be explained as before. Increasing the quantum length for the I/O-bound job increases the share of cpu allocated to the I/O-bound job, which generally reduces priority inversion but increases cpu fragmentation. The deviation from this expectation, more prominent in this figure than in the previous one, comes from the harmonics between the periodicity of the I/O-bound job and the scheduling quanta.

Note that Figure 3 illustrates clearly the inherent tradeoff between the level of priority inversion and the amount of cpu fragmentation. In the regions where the level of priority inversion is low (namely, the 10 msec curve and the 20 msec curve with IO-job quantum length between 10-15 ms and 25-40 ms), the amount of cpu fragmentation is high.

3.3 Summary

Two lessons follow from these experiments. First, CPU fragmentation can be a significant effect, especially when one optimizes for the response time of IO-bound jobs. Second, proper quantum lengths depend on the characteristics of each job as well as the workload. In order to provide interactive response time, a multiprocessor scheduler needs to carefully monitor the requirements of each of its jobs and react accordingly. Today's gang schedulers lack this reactive capability, making them unsuitable for workloads containing I/O-bound jobs.

Even an adaptive quantum length is not sufficient to deal completely with the problems of priority inversion and cpu fragmentation. A more flexible scheduling policy is called for, where higher priority jobs can interrupt lower priority jobs in order to keep disk utilization high. In addition, the cpu fragmentation problem can be partially alleviated if compute-bound jobs can be scheduled into the fragmented slots.

Interrupting processes of a low priority job and scheduling them in fragmented slots will only be beneficial, however, if that job is amenable to scheduling skew. If a compute-bound parallel job synchronizes frequently, interrupting one of its processes may improve disk utilization only at the cost of a large drop in cpu utilization. The next section explores the issue of skew in more depth.

4 Application Performance In Near-Gang Scheduled Environments

The presence of I/O-bound jobs exerts pressure against perfectly gang scheduling compute-bound jobs. This pressure appears in two forms. The first comes from the opportunity to reduce priority inversion by interrupting gang scheduled, compute-bound jobs to run I/O-bound jobs. The second comes from the need to schedule fragmented cpu resources. Together, they motivate the desire to flexibly gang schedule, and they lead to two questions about compute-bound jobs which relate to the cost of flexible gang scheduling. The first question concerns how well parallel jobs tolerate interruptions. The second question considers how fragmented resources can be utilized by parallel jobs.

This section explores the degree to which compute-bound applications benefit from gang scheduling. The more a job benefits from gang scheduling, the less it can tolerate interruptions, and the less efficiently it can utilize non-gang, fragmented resources. Our goal is to identify characteristics of an application which relate to its degree of benefit from gang scheduling.

Many studies have measured the benefits of gang scheduling relative to uncoordinated scheduling [1, 3, 9, 17]. Our study differs in that we are interested in the *marginal* benefit of a pure gang scheduled environment when compared to a gang scheduled environment with disruptions.

In order to get a quantification tool, we measure the performance of applications under various near-gang scheduled environments on a four-processor machine. These environments are produced by introducing perturbations [2] to a fully ganged environment. We set up four environments, each with a different set of perturbation characteristics.

In two of the environments, Sub_{FX} and Sub_{RR}, each perturbation removes a quantum of processing time from a single processor. In Sub_{FX}, the processor

[2] We use the term *perturbation* to refer to both positive deviations (granting of additional resources) and negative deviations (revocation of originally allocated resources)

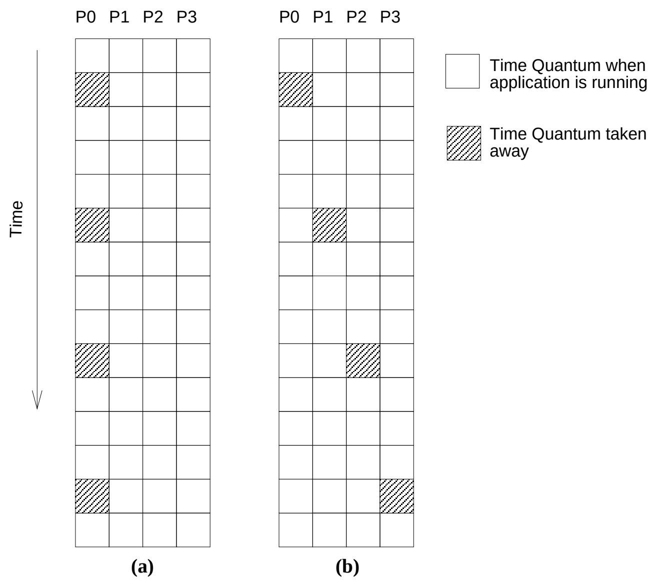

Fig. 4. Experimental setup for (a) fixed-processor takeaway and (b) round-robin takeaway

is fixed. We call this experiment fixed-processor takeaway. In Sub_{RR}, the processor is selected in a round-robin fashion. We call this experiment round-robin takeaway. See Figure 4.

In environments Add_{FX} and Add_{RR}, each perturbation gives an extra time quantum of processing time to a single processor. In Add_{FX}, called fixed-processor giveaway, the processor is fixed. In Add_{RR}, called round-robin giveaway, the processor is selected by round-robin.

The exact times of the perturbations are randomly distributed across the run time of the application in batches of four. Within a batch of four, the time of the first perturbation determines the times for the other perturbations. A fixed interval of three time quanta separate perturbations within a batch. Perturbations are batched in closely spaced groups of fours so that round-robin perturbations maintain coarse-grain load balance.

Quantum size is fixed at 500,000 instructions across the runs.

The motivations for the setup of these environments are as follows. The takeaway experiments indicates how compute-bound jobs behave when some of their processes are interrupted by I/O-bound jobs. The giveaway experiments indicates whether compute-bound jobs can utilize fragmented cpu resources. The results of the fixed-processor experiments are compared with the results of the round-robin experiments to examine the issue of load balance.

We run each application under the four scheduling environments, and we compare the run time of each to the run time under perfect gang scheduling. The results are presented in two sections below, one for a set of synthetic ap-

Type	Num	Description	Parameter
Emp	1	Constituent processes work independently without communication.	—
Barrier	1	Consists entirely of synchronizing barriers.	—
Workpile	4	Consists of a fixed amount of global work broken into independent units of work. The units of work is distributed dynamically to maintain load balance under arbitrary scheduling conditions.	Granularity of work unit (14%-2400% of a quantum)
Msg	4	Phases of request/reply communication are separated by barriers. Requests are asynchronous, but all replies must be received before a process proceeds to the next phase.	Communication pattern

Table 1. Information on the synthetic applications used in the giveaway/takeaway experiments. Applications are grouped into four types. Each entry gives the name of the type of application, the number of applications in that type, a description of the type of applications, and the parameter whose value differs between different members of that type.

plications and one for a set of real applications, which includes three applications from the SPLASH benchmark suite. As expected, we find that for load balanced applications with fine grain synchronization, the perturbations effect applications significantly. Real applications, however, exhibit internal algorithmic load-imbalance and are often somewhat latency tolerant. Because of these factors, the effects of perturbations on these applications are between a factor two and four smaller on a four-processor machine.

4.1 Synthetic Applications

Table 1 describes the set of synthetic applications used in this experiment. Based on the experimental results, each application can be classified as one of three types. Figure 5 presents the characteristic plots of the three types of applications. Each line on the graph plots the number of perturbations versus the change in run time for an environment. We have plotted the lines for all four experiments on the same graph. The three types of applications are:

i. **Synchronization intensive** This type of applications makes little progress unless it is being gang scheduled. When time quanta are taken away from a processor, all other processors stall as well. The entire application slows by the amount of time taken away. When time quanta are given to a processor, the processor stalls also, so the application receives no benefit from the extra time at all. *Barrier* and all of the *Msg* applications fall into this category.
ii. **Embarrassingly parallel** This type of applications exhibits the same poor behavior as synchronization intensive applications when time is given to or taken away from a single processor. However, the behavior is caused not

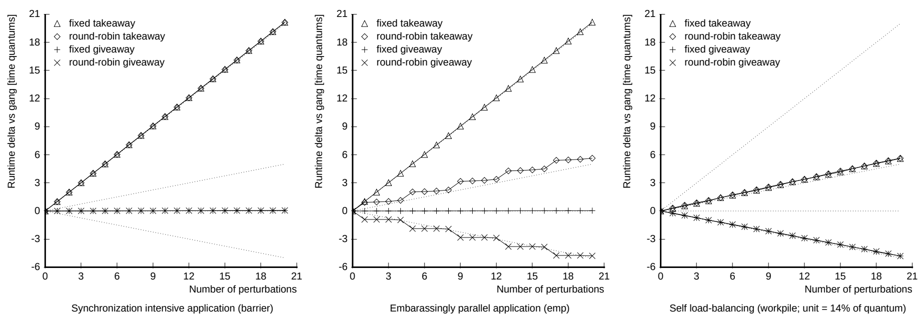

Fig. 5. Characteristic plots from giveaway-takeaway experiment for three types of applications. The actual application from which each plot is taken is listed in parenthesis below the plot. The four dotted lines are reference lines representing total cpu time taken away (worst case slowdown), 1/4 (1/P) of time taken away (best case slowdown), zero (worst case speedup), and -1/4 of time given away (best case speedup).

by synchronization but by load imbalance. In the round-robin experiments, where load balance is maintained, application of this type performs much better. When time quanta are taken away round-robin, run time degrades by 1/P quantum (here, P=4) per quantum taken away. The factor of 1/P arises because the single quantum of lost processing time is jointly recovered by the P processors. Similarly, when extra quanta are given to the job round-robin, run time improves by 1/P. *Emp* and coarse-grain *Workpile* (work unit = 24 time quanta) belong in this category.

iii. **Self load-balancing** This type of applications performs optimally under all scheduling conditions, because it suffers from neither synchronization nor load imbalances. Performance degrades by 1/P quantum per quantum taken away, and it improves by 1/P quantum per quantum given away. Three of the four *Workpile* applications fall into this category. Their granularity of work unit ranges from 14% of a time quantum to 240%.

Each class of applications above may also be identified by their minimum scheduling requirements. Synchronization intensive applications require gang scheduling. Embarrassingly parallel applications require fair scheduling of the constituent processes. We call this scheduling criteria *interprocess fairness*. Self load-balancing applications can utilize any processor resource; they have no requirement at all.

Of course, applications from real life will not fit cleanly into one of the above classes. An application with a moderate but nontrivial synchronization rate, for example, will have behavior which falls somewhere between that of a synchronization intensive application and an embarrassingly parallel application. Similarly, workpile-like applications with limited load-balancing mechanism will have behavior which falls somewhere between that of an embarrassingly parallel application and a self-scheduling application. We can indeed run the experi-

App.	Quanta/Barrier	Type Msgs	Msgs/Proc/Quantum
Enum	50	Non-blocking	254
Water	10	Blocking	12
LU	10	Blocking	3
Barnes	50	Blocking	28

Table 2. Characteristics of the real applications

ments with more exhaustive sets of parameter values to quantify some of these effects, but such studies have been done before before [6, 9, 18], and here we are more interested in the qualitative difference in behavior at extreme ends of the application spectrum.

4.2 Real Applications

The takeaway/giveaway experiments are applied to four real applications as well. One, *Enum*, finds the total number of solutions to the triangle puzzle (a simple board game) by enumerating all possibilities breadth-first. The other three, *Barnes*, *Water*, and *LU*, are scientific applications from the Splash benchmark suite implemented using CRL, an all-software shared-memory system [12]. See Table 2 for statistics describing the applications. Because the applications are non-homogeneous in time, we obtain each data point by taking the average result from 20 runs, each with a different set of time quanta given or taken away. Because the applications are also non-homogeneous across processors, we run the fixed processor takeaway experiment on processors 0, 1, and 3.

Figures 6-9 show the results of the experiments. To better understand the applications for the purpose of explaining the results, we obtain a trace for each application run under gang scheduling, and we plot the progress made on each processor versus time. These traces are presented next to the experimental results. Because the progress plot for *Water* follows such a regular pattern, only a magnified subsection is presented.

Enum Of the four sets of results, *Enum* stands out by itself. Its experimental plot closely resembles that of an embarrassingly parallel application. In reality, *Enum* has three characteristics which make it embarrassingly parallel:

– It is load balanced, as suggested by its progress plot.
– It has infrequent barrier synchronization (compared to the length of a time quantum).
– It communicates with non-blocking messages.

The results for *Enum* are actually consistently worse than that of a perfect embarrassingly parallel application for three reasons. First, even in the absence of synchronization, failure to gang schedule incurs overhead in the form buffering

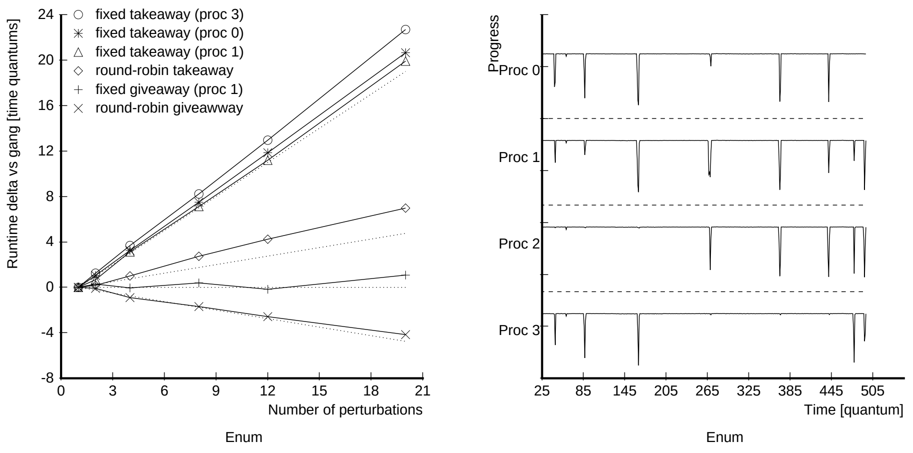

Fig. 6. Experimental plot (left) and progress plot under gang scheduling (right) for *Enum*. Note that in the experimental plot, each line starts at perturbation = 1. The four dotted lines are reference lines representing time taken away (worst case slowdown), 1/4 of time taken away (best case slowdown), zero (worst case speedup), and -1/4 of time given away (best case speedup).

cost in our system. At about 250 instructions per buffered message, this overhead can be up to three quanta when 20 quanta are taken away. Note that this cost is smaller for the quantum giveaway experiments because the buffer overhead is spread over P-1 processors.

Second, as load balanced as any real application can expect to be, *Enum* still has some load imbalances. The effect of imbalances on run time is evident by comparing the run time of the takeaway experiment from processor 3 with the run time of the takeaway experiments from processors 0 and 1. Processor 3 is the bottleneck processor for over 60% of the application (as evident by the lack of valleys in much of the progress graph). As a result, taking away time from processor 3 results in a slower run time than taking away time from processor 0 or 1.

Finally, for the round-robin experiments, the benefit from maintaining inter-process fairness is lost if a barrier interrupts a set of round-robin perturbations. Consequently, the slowdown is noticeably higher than the expected 25% of the time taken away.

Water, LU, and Barnes The results for *Water*, *LU*, and *Barnes* are similar. Because these applications exhibit significant load imbalances (as seen by the deep and long valleys in their progress plots), their results do not directly resemble that of any of the synthetic applications. In fact, load imbalance and blocking messages are two common features which explain most of the results for these applications.

In the fixed-processor quantum takeaway experiments, the amount by which

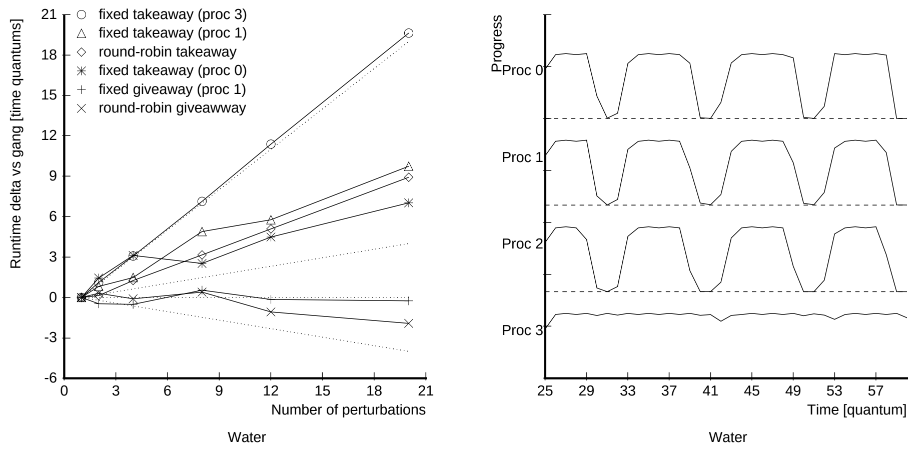

Fig. 7. Experimental plot (left) and progress plot under gang scheduling (right) for *Water*.

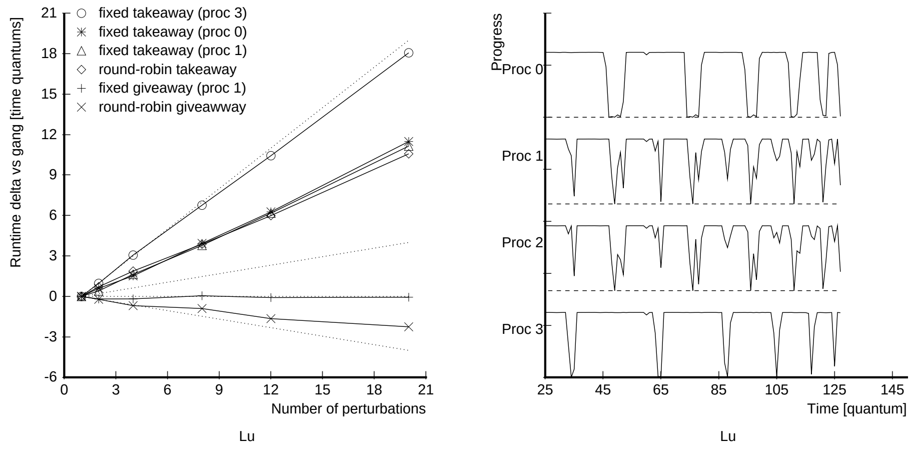

Fig. 8. Experimental plot (left) and progress plot under gang scheduling (right) for *LU*.

the application slows down depends largely on the degree to which the processor taken away is a bottleneck. In the progress plot, a processor bottleneck is marked by full progress (absence of a valley) at a time when one or more processors are not making progress (valleys). For *Water*, processor 3 is the clear bottleneck, so taking away cpu time from processor 3 slows down the application by 100% of the time taken away. For *LU*, processors 0 and 3 alternate as bottlenecks. However, when processor 0 is the bottleneck, it is only slightly behind the other processors. So any other processor with time taken away readily overtakes processor 0's role

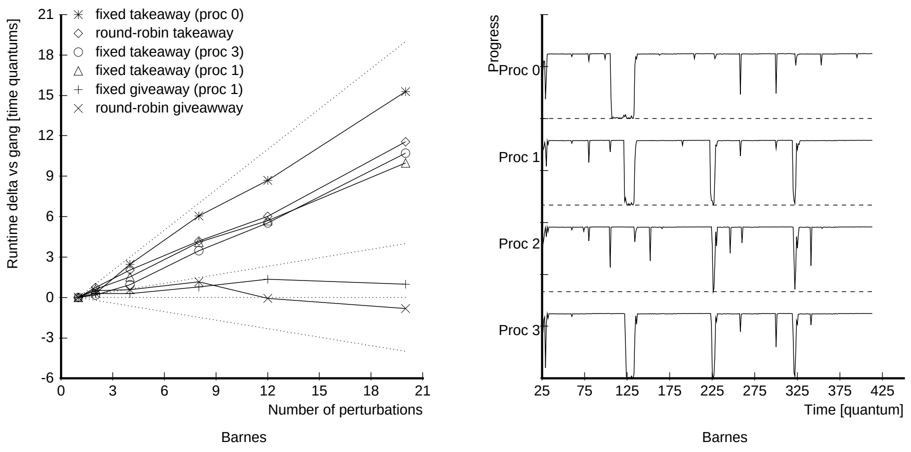

Fig. 9. Experimental plot (left) and progress plot under gang scheduling (right) for *Barnes*.

as half the bottleneck. The plot reflects these bottleneck conditions: processor 3 slows down by close to 100% of time taken away, while processors 0 and 1 slow down by considerably less, with processor 0 slower than processor 1 by a small amount. Finally, in *Barnes*, processor 0 is the major bottleneck for the latter 2/3 of the application, while neither processor 1 or 3 is a significant bottleneck. As a result, takeaway from processor 0 is considerably worse than takeaway from processors 1 or 3.

When quanta are taken away from a non-bottleneck processor, run time degrades due to blocking requests sent to it by a bottleneck processor. In all three of our applications, this effect degrades performance by about 50% of the time taken away. This 50% ratio also holds for round-robin takeaway: the application communicates with blocking messages too frequently to benefit from the coarse grain interprocess fairness ensured by round-robin takeaway.

As for the giveaway experiments, fixed processor giveaway from processor 1 fails to improve performance for all three applications because processor 1 is not the bottleneck processor. Round-robin giveaway improves performance of *Water* and *LU* because time is given to the bottleneck processor and the rate of blocking request is low. On the other hand, round-robin giveaway fails to improve performance of *Barnes* due to the application's high rate of blocking requests.

4.3 Summary

We summarize the results of the experiments with several observations and conclusions.

Synchronization intensive applications reap full benefits from gang scheduling and in fact require it for good performance. Embarrassingly parallel applications and self load-balancing applications, on the other hand, do not benefit from gang scheduling. Note that the volume of communication is an orthogonal issue. *Enum* is an example of a real embarrassingly parallel application, even though it communicates a lot.

Load imbalance is a common application characteristic in practice. Enforcing gang scheduling on applications with this characteristic hurts machine utilization.

Many real applications can tolerate at least a small amount of perturbations in a gang scheduling environment. Load imbalanced applications are tolerant of perturbations if they are not occurring at the bottleneck processors. And all applications that we studied are tolerant to round-robin perturbations. Even the inherently load-imbalanced CRL applications slow down by only 50-60% of the time taken away by the round-robin disturbances. This is because round-robin perturbations do not introduce severe artificial load imbalances.

For applications without self load-balancing, two characteristics primarily determine how they behave under various scheduling conditions. The characteristics are the level of load imbalances and the volume of blocking communication. Total volume of communication is a second order effect.

To avoid scheduler-induced load imbalances, interprocess fairness is a good criteria to keep in mind when a gang scheduler does alternate scheduling. The failure to do so could be a reason why research has found that random alternate scheduling provides little gain to system throughput [17].

To first order, there are two types of blocking communication. One is the barrier-like communication used to check that processes have all reached a checkpoint. The other type is the request/reply-type message, where one process is trying to obtain information which can be readily provided by another.[3]

The two types of communication have different scheduling characteristics. Wait time for barrier is affected by load imbalances. To minimize this wait time, it is more important to minimize the load imbalances than it is to gang schedule. Wait time for request/reply, on the other hand, depends on whether the sender and receiver are scheduled simultaneously. This wait time can only be minimized by gang scheduling the sender/receiver pair.

5 Runtime Identification of Gangedness

Section 3 illustrates the benefits of flexible gang scheduling derived from not being required to gang schedule compute-bound jobs. Section 4 shows that the costs of relaxing gang scheduling varies between applications. Together, they suggest that a scheduler can benefit from identifying the *gangedness* of each application, defined to be the level to which the application benefits from gang scheduling. Jobs with high gangedness indicate both a low tolerance of interruptions and

[3] We can also have blocking communication where the receiver is blocked instead of the sender. A processor blocks waiting for an incoming message containing the information it requires. But there is no fundamental difference between that and request/reply: it's like request/reply done with "polling."

the inability to utilize fragmented cpu resources, while jobs with low gangedness can be scheduled much more flexibly to improve overall system performance.

In this section, we consider how gangedness can be determined from coarse-grain messaging statistics.

5.1 Information Content of Message Arrival Times

To relate messaging statistics to gangedness, we look at the high level information contained in message arrival times, and we try to find a relationship between that information and gangedness.

First, messages contain information about synchronization, because they are the medium through which synchronization is implemented. And synchronization relates directly to gangedness. Unfortunately, messages don't necessarily synchronize, and there is no way of distinguishing synchronizing messages from non-synchronizing ones.

Alternatively, message arrival times contain information about the state of a process. A process can send a message only if it is not blocked, or in other words, if the process is making forward progress. To relate gangedness to progress, a high gangedness means that an application must be gang scheduled in order to make forward progress.

5.2 Experiment

We test our intuition on the relationship between message counts and gangedness with the following experiment. For each of the real applications described in Subsection 4.2, we run it in the fixed-processor takeaway environment described in Section 4. The environment is suitable because it contains both gang scheduled quanta and non-gang scheduled quanta. Any of the four environments in Section 4 could have been used for this purpose, and we do not expect the choice of environment to effect the conclusions we draw.

For every processor-quantum unit, we collect the number of messages received and the amount of progress made. These statistics are used to generate four pairs of values per quantum, one pair for each processor. In the pair of values, we associate the number of messages received by a processor to the amount of progress made by all *other* processors. This association corresponds to the intuition that the number of messages received should correlate with the amount of progress made by the senders of the messages.

Within this scheduling environment, we distinguish between three sets of data values. *Gang-all* is the set of values collected on ganged, undisturbed quanta. *Non-Gang-running* is the set of values collected on the running processors in non-ganged, disturbed quanta. Finally, *Non-Gang-non-running* is the set of values collected on the non-running processors in non-ganged, disturbed quanta.

To collect enough data values for all three sets, we run the experiment five times. Table 3 summarizes the results. Each row gives the average and standard deviation of message count per quantum as well as the average and standard deviation of progress per quantum.

Description	Msg Avg	Msg SD	Prog Avg	Prog SD
Enum				
Gang-all	247.73	47.51	1483678	209519
Non-Gang-running	162.29	60.21	882735	307236
Non-Gang-non-running	242.30	88.57	1324103	459779
Water				
Gang-all	15.41	19.63	1127732	485835
Non-Gang-running	2.54	3.40	479135	387983
Non-Gang-non-running	1.17	0.70	718703	554749
LU				
Gang-all	2.85	3.19	1242051	397927
Non-Gang-running	1.40	1.82	740624	313568
Non-Gang-non-running	0.89	0.73	1110937	421677
Barnes				
Gang-all	28.44	56.99	1436370	282763
Non-Gang-running	2.53	3.00	507318	300674
Non-Gang-non-running	1.80	0.78	760978	389349

Table 3. Aggregate statistics for correlation experiment in non-gang environment. Each row gives the average and standard deviation of message count, and the average and standard deviation of sender progress.

The progress data are consistent with the characteristics of the applications. *Enum* uses infrequent barriers and non-blocking messages, so it can make progress without gang scheduling. As expected, the experiment shows that sender progress for *gang-all* and *non-gang-running* are roughly within 10% of each other (1483678 vs. 1324103). The sender progress for *non-gang-non-running* is lower than that of *gang-all* and *non-gang-running*, but that only reflects the fact it has one less sender than the other data values; normalizing these values with the number of senders would yield the expected nearly identical values.

Water, *LU*, and *Barnes* all have infrequent barriers, but they use blocking messages. The rate of messages then determines how much progress an application can make in a non-gang time quantum. [4] *LU* has the lowest message rate, so low that it in fact runs quite well in non-gang quanta. *Barnes* and *Water*, on the other hand, have message rates which are high enough to cause their performance to degrade in non-gang quanta, with *Barnes*'s higher rate yielding a more severe degradation.

Figure 10 plots the range of number of messages received for each application in both gang quanta and non-gang quanta. To obviate the need to normalize all the values by the number of sender processors, for the non-gang quanta we

[4] As Section 4 shows, the issue of load balance is an important consideration as well. Frequency of barriers and blocking messages determines the level of progress that can be made in non-gang environments. Level of load balance determines whether such progress is ultimately *useful*, *i.e.,* whether it reduces the run time of the application.

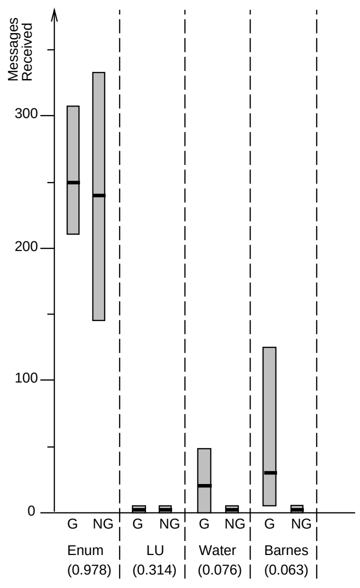

Fig. 10. Number of messages received per quantum per processor for each application in gang quanta (G) and non-gang quanta (NG). Each bar represents the 90% confidence interval; the thick line within the bar marks the average message count. Each number in parenthesis below the application name shows the ratio of the average message counts between the non-gang and the gang quanta.

use data from *non-gang-non-running*, which has the same number of sender processors as *gang-all*.

The results confirm our intuition. *Enum* is the only application which uses non-blocking messages. Therefore, it is the only application which can sustain a high volume of communication in non-gang quanta. As stated in Subsection 5.1, this run-time observation allows one to conclude that the application has low gangedness and does not require gang scheduling. On the other hand, *Water*, *LU*, and *Barnes* all use blocking messages, so their volume of communication during non-gang quanta is low. One cannot, however, draw any conclusion about gangedness from this lack of communication: applications can be making progress without sending any message.

Rather than using message count of non-gang quanta, a more robust way to determine the gangedness of an application is to use its ratio of message counts between non-gang and gang quanta. A high ratio corresponds to low gangedness, while a low ratio corresponds to high gangedness. As shown in Figure 10, ordering the applications by ratio corresponds exactly to reverse-ordering the applications by gangedness. Moreover, the ratios for applications with low gangedness (*Enum* and *LU*) are at least a factor of five larger than the ratios for applications with high gangedness (*Water* and *Barnes*). This sizable difference makes it easy to accurately categorize the applications into a high gangedness class and a low gangedness class simply based on their message ratios.

5.3 Summary

We summarize what we learn about the relationships between message count, progress, and gangedness. Message count relates to progress in the following way. High message count always implies progress. Low message count, on the other hand, can arise because the application is not sending any message, so it does not necessarily imply a lack of progress.

As for the relationship between message count and gangedness, a high message count while an application is not gang scheduled shows that an application can make progress without gang scheduling. It thus indicates a low gangedness. More generally, one can compare the average message counts between non-gang and gang environments to determine the gangedness of the application. A high ratio of non-gang message count to gang message count corresponds to low gangedness, while a low ratio corresponds to high gangedness.

Note that our conclusion is somewhat counterintuitive to conventional thinking. Conventional thinking has the notion that the more an application makes use of communication resources, the greater the need for the application to be gang scheduled. In fact, Sobalvarro [17] bases his dynamic coscheduling scheme directly on this principle, as he achieves coscheduling behavior by taking each incoming message as a cue to schedule the addressed process. We argue that if a processor continues to receive messages for an unscheduled process, the sending processes must be making progress under the status quo, and no scheduler intervention is necessary.

6 Related Work

Multiprocessors can be shared by partitioning in space, in time, or both. Much work has been done to explore and compare the various options [7]. Space-sharing can be very efficient for compute-bound applications and is desirable when permitted by the programming model and application characteristics [3, 11]. Time-sharing remains desirable for flexibility in debugging and in interleaving I/O with computation. These considerations become more important as multiprocessors become more mainstream.

Ousterhout introduced the idea of coscheduling or gang scheduling to improve the performance of parallel applications under timesharing [15]. There are two benefits to gang scheduling. First, from a programmability standpoint, gang scheduling is attractive because it is compatible with conventional programming models, where processes of a parallel application are assumed to be scheduled simultaneously. This feature simplifies reasoning about performance issues as well as correctness issues like deadlock and livelock. Second, from a performance standpoint, gang scheduling is absolutely necessary for applications that synchronize frequently.

Several studies have quantified the benefits of gang scheduling [1, 3, 9]. Feitelson and Rudolph [9] demonstrate that gang scheduling benefits applications that perform fine-grain synchronization. Arpaci *et al* [1] and Mraz [14] observe that the disruption of system daemons in a network of workstation is potentially intolerable without some efforts to synchronize gangs across processors. Our study confirms these observations and draws detailed conclusions about the causes of slowdown for specific applications.

There are costs to gang scheduling as well. Much literature focuses on its cost of implementation [2, 8, 17, 19]. This cost comes about because gang scheduling requires global coordination and centralized scheduling. The implementation of our scheduler uses a two level distributed hierarchical control structure for efficiency similar to Distributed Hierarchical Control [8, 19]. But even in a system where these features come for free, gang scheduling still has costs which make its universal use undesirable. Our study shows the degradation of response time due to a form of priority inversion. Other effects degrade utilization, for instance by losses due to constraints on the packing of jobs into the global schedule and by the inability to recover wasted time in a job with load imbalance. An ideal scheduler would perform a cost-benefit analysis which gives proper weights to all the issues above.

Studies have pointed out that parallel scientific applications may consist of a significant amount of I/O activities due to reading and writing of results [4, 5]. I/O activities may also come from paging activities, and Wang [20] notices that even for programs written with a SPMD programming model, there is little coordination of I/O across processing nodes because of data dependencies. This behavior is consistent with our assumption in the experiments that I/O activities across the processing nodes are independent.

In our work, we assume that the members of the job are known *a priori* and concentrate on the problem of deciding whether to gang schedule based on indirect measurements. A fully dynamic solution to gang scheduling includes the identification of gang members at run-time. Sobalvarro [17] uses individual message arrivals as cues to the identification of a gang, while Feitelson and Rudolph [10] monitor the rate at which shared communication objects are being accessed to determine whether and which processes need to be ganged.

Given the processes that make up each job, our system monitors communication rate between job members to identify those jobs that require coscheduling versus those jobs that can tolerate having their processes individually scheduled. In this respect our scheduler differs from both the Meiko CS-2 [16], and SGI IRIX [2] schedulers.

7 Conclusion

We summarize the results presented in this paper. First, traditional gang scheduling hurts workloads containing I/O. Second, interrupting ganged, compute-bound jobs can benefit workloads. Third, one needs to schedule fragmented cpu resources intelligently, by selecting jobs with low gangedness to run in those spaces, and by preserving interprocess fairness. Finally, message statistics can identify the gangedness of applications.

We envision a scheduling strategy flexible enough to accommodate all jobs. I/O-bound jobs can have either coordinated I/O or uncoordinated I/O. Compute bound jobs may either be perturbation-friendly or perturbation-sensitive. Scheduling would be done in two sets of rounds. Uncoordinated I/O-bound jobs and perturbation-friendly compute-bound jobs are scheduled in rounds with loose coordination. Coordinated I/O-bound jobs and perturbation-sensitive compute-bound jobs can be scheduled in rounds with strict coordination.

At the very high level, we demonstrate that a flexible gang scheduler is both necessary and possible.

References

1. R. Arpaci, A. Dusseau, A. Vahdat, L. Liu, T. Anderson, and D. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of Sigmetrics/Performance '95*, pages 267–278, May 1995.
2. J. M. Barton and N. Bitar. A Scalable Multi-Discipline, Multiple-Processor Scheduling Framework for IRIX. In *Lecture Notes in Computer Science, 949*, pages 45–69, Santa Barbara, 1995. Springer Verlag. Workshop on Parallel Job Scheduling, IPPS '95.
3. M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos. Multiprogramming on Multiprocessors. In *Proceedings of the third IEEE Symposium on Parallel and Distributed Processing*, 1991.
4. R. Cypher, S. Konstantinidou, A. Ho, and P. Messina. A Quantitative Study of Parallel Scientific Applications with Explicit Communication. In *The Journal of Supercomputing*, pages 5–24, January 1996.
5. J. M. del Rosario and A. Choudhary. High Performance I/O for Parallel Computers: Problems and Prospects. In *IEEE Computers, vol. 27, no. 3*, pages 59–68, 1994.
6. A. Dusseau, R. Arpaci, , and D. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of ACM SIGMETRICS 1996*, Philadelphia, May 1996. ACM.
7. D. G. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems. Technical Report IBM/RC 19790(87657), IBM, October 1994.
8. D. G. Feitelson and L. Rudolph. Distributed Hierarchical Control for Parallel Processing. In *Computer*. IEEE, May 1990.
9. D. G. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. In *Journal of Parallel and Distributed Computing*, pages 306–318, December 1992.
10. D. G. Feitelson and L. Rudolph. Coscheduling Based on Runtime Identification of Activity Working Sets. In *International Journal of Parallel Programming*, pages 135–160, April 1995.
11. A. Gupta, A. Tucker, and L. Stevens. The Impact of Operating System Scheduling Policies and Synchronization Methods of the Performance of Parallel Applications. In *Proceedings of 1991 ACM Sigmetrics Conference*, 1991.

12. K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.
13. K. Mackenzie, J. Kubiatowicz, M. Frank, W. Lee, V. Lee, A. Agarwal, and M. F. Kaashoek. UDM: User Direct Messaging for General-Purpose Multiprocessing. Technical Memo MIT/LCS/TM-556, March 1996.
14. R. Mraz. Reducing the Variance of Point-to-Point Transfers for Parallel Real-Time Programs. In *IEEE Parallel & Distributed Technology*, 1994.
15. J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *3rd International Conference on Distributed Computing Systems*, pages 22–30, 1982.
16. K. E. Schauser and C. J. Scheiman. Experience with Active Messages on the Meiko CS-2. In *Proceedings of the 9th International Symposium on Parallel Processing*, 1995.
17. P. G. Sobalvarro and W. E. Weihl. Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors. In *Lecture Notes in Computer Science, 949*, pages 106–126, Santa Barbara, 1995. Springer Verlag. Workshop on Parallel Job Scheduling, IPPS '95.
18. A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles (SOSP-12)*, pages 159–166, December 1989.
19. F. Wang, H. Franke, M. Papaefthymiou, P. Pattnaik, L. Rudolph, and M. S. Squillante. A Gang Scheduling Design for Multiprogrammed Parallel Computing Environments. In *Lecture Notes in Computer Science, 1162*, pages 111–125, Honolulu,Hawaii, 1996. Springer Verlag. Workshop on Parallel Job Scheduling, IPPS '96.
20. K. Y. Wang and D. C. Marinescu. Correlation of the Paging Activity of the Individual Node Programs in the SPMD Execution Mode. In *Proceedings of the Hawaii International Conference on System Sciences*, 1995.

This article was processed using the LaTeX macro package with LLNCS style