ACADEMIC PRESS

Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

J. Parallel Distrib. Comput. 63 (2003) 649–668

Journal of Parallel and Distributed Computing

http://www.elsevier.com/locate/jpdc

Integrated scheduling: the best of both worlds

Jon B. Weissman,* Lakshman Rao Abburi, and Darin England

Department of Computer Science and Engineering, University of Minnesota, Twin Cities, Minneapolis, MN 55455, USA

Received 31 January 2002; revised 14 October 2002

Abstract

This paper presents a new paradigm for parallel job scheduling called integrated scheduling or iScheduling. The iScheduler is an application-aware job scheduler as opposed to a general-purpose system scheduler. It dynamically controls resource allocation among a set of competing applications, but unlike a traditional job scheduler, it can interact directly with an application during execution to optimize resource allocation. An iScheduler may add or remove resources from a running application to improve the performance of other applications. Such fluid resource management can support both improved application and system performance. We propose a framework for building iSchedulers and evaluate the concept on several workload traces obtained both from supercomputer centers and from a set of real parallel jobs. The results indicate that iScheduling can improve both waiting time and overall turnaround time substantially for these workload classes, outperforming standard policies such as backfilling and moldable job scheduling.

Keywords: Scheduling; Cluster computing; Distributed computing; Parallel processing

1. Introduction

Efficiently executing resource-intensive parallel scientific applications in dynamic parallel and distributed production environments is a challenging problem. Performance depends on resource scheduling at several levels: *job schedulers* which multiplex different applications across limited system resources and *application schedulers* which request resources for their applications. Applications can experience large slowdowns in time-shared systems, or high wait times in space-shared systems, due to contention for these limited valuable resources. Part of the problem is that a limited set of shared resources can rapidly become oversubscribed as the number of applications to execute increases. This is confirmed by several supercomputer workload studies [9,10,18]. However, we believe that inefficient resource allocation by job schedulers and inefficient resource utilization within the application further exacerbates this problem.

The problem is that job schedulers for parallel machines are often built to optimize metrics that may not deliver the best possible application performance. For example, many job scheduling policies are geared to high resource utilization because large parallel machines are expensive, or fairness (e.g. first-come-first-serve), or high throughput. In addition, job scheduling is static: job schedulers for parallel applications generally do not reconsider prior scheduling decisions. Another problem is that job scheduling may be minimally supported in distributed network environments. Without coordination, different application schedulers will make independent scheduling decisions that may lead to reduced application and system performance. Uncoordinated application schedulers may also encounter race conditions on shared resources. In addition, application schedulers typically select resources without regard for the negative impact this may have upon other applications. In some cases, application schedulers may even over-allocate resources with limited benefit to their application. This is particularly true when the end-user is performing application scheduling since it can be hard to manually determine the appropriate number of resources for applications with a high degree of variance or data-dependent behavior.

*Corresponding author. Fax: +612-625-0572.
E-mail address: jon@cs.umn.edu (J.B. Weissman).
URL: http://www.cs.umn.edu/~jon.

0743-7315/03/$ - see front matter
doi:10.1016/S0743-7315(03)00013-3

We believe that the lack of performance delivered to applications is in part due to the functional gap between application scheduling and job scheduling. Application schedulers make resource allocation decisions solely on behalf of their application [3,28,29]. They exploit specific knowledge of the application to select resources that are predicted to yield performance that meets user objectives such as minimal response or completion time. In contrast, job scheduling takes a more global view and makes resource allocation decisions across a set of jobs [4,8,12,14,19]. Job-scheduling approaches span space- to time-sharing policies to combined approaches [31]. Each parallel application is assumed to be a black-box with little or no accompanying information.

In this paper, we propose a novel integrated scheduling framework (*iScheduler*) that bridges the gap between application and job scheduling for parallel scientific applications. We assume that the job or application has already been decomposed into a parallel computation and that it has a special front-end (called an adaptive API) that will be described shortly. The most common model for iScheduler jobs is data parallel in which the parallel components are identical, though this is not a strict requirement. The *iScheduler* is an *application-aware* job scheduler as opposed to a general-purpose system scheduler. Application-aware means that the job scheduler can interact with the jobs as they are running. It dynamically controls resource allocation among a set of competing applications, but unlike a traditional job scheduler, it can interact directly with an application during execution to optimize resource allocation. In some cases, it can exploit application-specific knowledge, e.g. it can use estimated execution time, and in others it makes more generic decisions. Application-awareness is achieved via the adaptive API. The iScheduler can be applied in two ways. A *user-centric* iScheduler can more effectively manage a pool of resources (e.g. a partition of a supercomputer or a cluster) for a set of related applications submitted by a single user. For example, many problems require the repeated execution of the same or similar applications such as parameter studies, Monte-Carlo models, and gene sequence comparison, in which results from each run are typically averaged for a final result, or input into subsequent runs, or simply collected as parts of the overall solution. Workload traces from supercomputer logs reveal a high degree of application repetition [27] and this has been confirmed by several informal surveys [7] and anonymized workload traces [10]. A *system-centric* iScheduler operates across a set of common applications submitted by different users. For example, domain-specific problem-solving environments (PSE) [2,5,17,20,24] can use the iScheduler to manage the resource allocation associated with each concurrent request (e.g. to solve a system of equations) submitted to a PSE.

We have developed an iScheduler framework and applied it both to a supercomputer workload derived from trace data at several centers and to a stream of real parallel jobs that we have implemented. The results indicate that integrated scheduling improved both waiting time and the end-to-end finishing time of parallel applications when compared to several baseline scheduling policies including backfilling and moldable job scheduling, but over an order of magnitude improvement when compared to the default scheduling policy.

2. The iScheduler

2.1. Fluid resource management

The iScheduler is an application-aware job scheduler that is designed to exploit the advantages of application- and job-level scheduling. Traditional job scheduling is static: job schedulers for parallel applications generally do not reconsider prior scheduling decisions. The degree of interaction between a job scheduler and the application (or its scheduler) is minimal and usually consists of an initial resource request (e.g. the job wants P processors). In addition, if the initial resource request cannot be met, then the application must wait because there is often no mechanism to allow it to acquire resources later on, even if it could start with fewer resources. Finally, if some of the resources held by an application are needed by a higher priority application, then the job scheduler has no choice but to suspend the entire application. The iScheduler performs fluid resource management for an application (much like an application-level scheduler), but will perform resource management across a job stream (like a job-level scheduler). Such integration does not come for "free". It requires an API that provides application information (*sensing*) and the ability to control the application's behavior (*actuating*). Sensing provides dynamic performance information such as per iteration execution cost, while actuating is used to take action on the application, e.g. adding or removing processors to it.

Integrated scheduling requires application modification to support these sensing and actuating functions. The iScheduler makes scheduling decisions for a potentially dynamic collection of application instances across a resource pool, e.g. a cluster, parallel supercomputer, network of clusters, or computational Grid. It works in concert with application schedulers to make effective resource allocation decisions both statically and dynamically (Fig. 1). In this paper, the resource on which we focus is processors. Unlike traditional scheduling approaches, resource management in the iScheduler is a *fluid* process in which resources may be allocated and deallocated throughout the lifetime of an

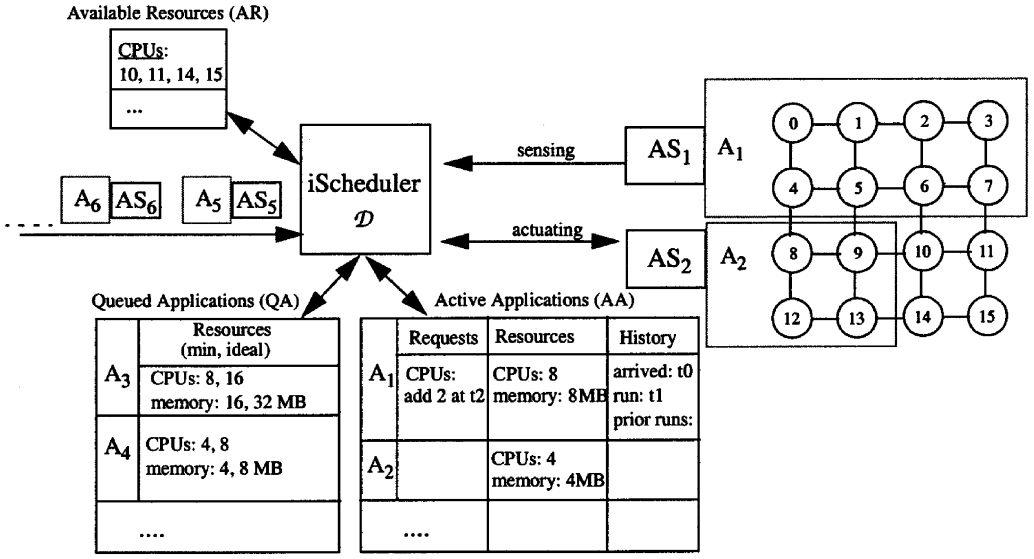


Fig. 1. iScheduler architecture. iScheduler manages incoming application stream (shown as an application A_i and its scheduler AS_i) and allocates resources across 16 CPUs. We have shown the application scheduler as decoupled from the application; however in some cases, the application scheduler may be embedded within the application. Application resource requirements (if known) and constraints may be provided—*min* is the required minimum amount of resource and *ideal* is the desired amount. There are two active applications (A_1, A_2) and two queued applications (A_3, A_4). Application A_1 has made a dynamic request for an additional 2 CPUs. iScheduler makes allocation decisions using information from all depicted sources.

application. iScheduler resource management is based on three core ideas:

- cost prediction
- adaptivity
- admission control

Cost prediction models allow the iScheduler to evaluate and assign candidate resources to its collective set of applications in a manner that meets performance objectives. It allows the iScheduler to assess the cost/benefit of resource allocation and deallocation decisions. Adaptivity allows the application and the iScheduler to adjust their behavior in response to dynamic run-time events. It provides the iScheduler greater flexibility in making collective resource allocation decisions. Admission control is a decision that is made when a new application arrives to system. The iScheduler must decide whether to queue the application or to run it by admitting it into the system. This is particularly important when system resources are oversubscribed with respect to the application stream. When admitting an application, the iScheduler must ensure that there are sufficient resources for the application. This may require the iScheduler to take away resources from running applications. For example, suppose a parallel system contains 128 space-shared processors and a parallel application is running on 120 processors. Using traditional job schedulers, a newly arriving application requiring 16 processors would be queued (unless the new application had higher priority). Using a moldable job scheduling, the new job could be given the eight free processors which would likely result in poor performance. In contrast, the iScheduler could draw eight additional processors from the first application perhaps not affecting its run-time greatly, which would allow the new application to run. Also observe how this improves machine utilization relative to traditional job scheduling since the eight processors are no longer idle. In many instances, there is a point of diminishing returns with respect to the amount of resources for many application classes, so the penalty of removing resources may not be substantial (see Fig. 2 for some examples).

The iScheduler makes resource management decisions as a function of predictive cost functions C, active applications AA (including their priority, performance history, age, current resource allocations and constraints, and pending resource requests if any), queued applications QA (including their priority, wait time, resource requirements and constraints), and available resources AR. The space of iScheduling decisions can be characterized by a time-dependent policy or decision function $\mathcal{D}$ (t, C, AA, QA, AR). The possible decisions include resource allocation, resource deallocation, application admission, and application suspension. An important issue is when will $\mathcal{D}$ be invoked? Some possibilities include: application arrival, completion, active application resource request, and when application queue time or application age exceeds a threshold. A complete answer depends on the performance objectives of the iScheduler. The iScheduler decision function can be configured to optimize a variety of

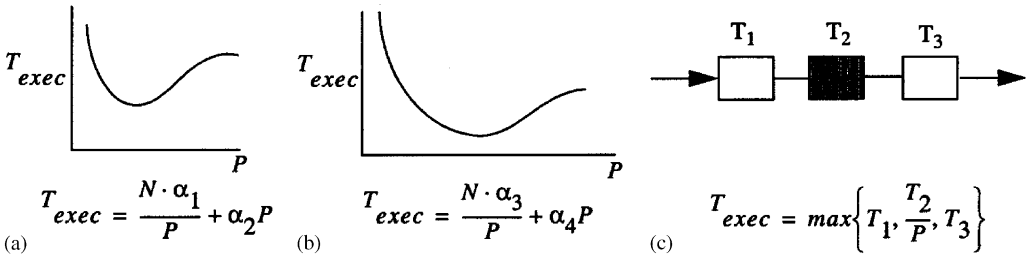


Fig. 2. Point of diminishing returns for (a) data parallel, (b) master-slave bag-of-tasks, and (c) pipeline applications (N is the problem size, P the # of processors). In (a) and (b), a parabolic shape is commonly seen. In (a) the curve turns upwards as communication begins to offset any gains from parallelism. The constants depend on problem and network parameters. Similarly for (b), the curve turns up when the overhead experienced by the master begins to dominate any gains. Lastly for (c), suppose each stage can run in T_i time serially, and the middle stage is parallel. Once T_1 or T_3 exceeds T_2/P there is no benefit to adding more processors to the middle stage.

metrics: completion time, response time, wait time, utilization, and throughput.

2.2. iScheduler policies

We have developed several policies for user- and system-centric iSchedulers that space-share processing resources with the goal of reduced completion time. These iSchedulers we have developed make resource allocation decisions when applications arrive and complete. The resource that is being managed is the number of processors. iScheduler fluid resource management policies will adjust resource consumption of both arriving and running applications. These policies come into play when there are insufficient resources to give all applications concurrent access to their ideal amount of resources. When an application arrives to the system, the iScheduler must decide whether to admit it into the system. If it cannot get its ideal resources an iScheduler can choose to "harvest" resources from running applications to enable it to run. Harvesting itself consists of several issues: when to harvest? from whom to harvest (the victims)? and how much to harvest? In this paper, we set the harvest parameter to the minimum resource amount (*min*) needed by the application. If this resource amount cannot be collected, then the application must be queued. An application cannot be harvested below its minimum resource requirements. When an application finishes, the iScheduler must decide how to distribute the available resources to both queued and possibly running applications. If minimizing wait time is desired, then preference should be given to queued applications increasing the degree of parallel job multiprogramming.

iScheduler resource management policies have four components: *control_policy*, *admission_policy*, *harvest_policy*, and *distribution_policy*. The *control_policy* refers to global properties that all policies must respect. For example, for stability we may want to limit how many times an application can be harvested, or protect newly arriving applications from harvesting until they reach a certain age, or to insure that the benefit of an adaptive action outweighs its cost, etc. The *harvest_policy* determines the victim set and the amount that each application must relinquish. The *distribution_policy* determines what to do with the resources freed when an application finishes. In this paper, all distribution policies first ensure that the *admission_policy* is satisfied. In the large space of possible policies, we are exploring this set of options:

2.2.1. Admission control policies

achieve_mp (M): this policy will admit applications into the system FCFS to achieve a degree of multiprogramming at least equal to M. An application will be queued if M has been reached and insufficient free resources are available.

2.2.2. Harvesting policies

never_harvest: do not harvest (but redistribution to running applications is allowed)

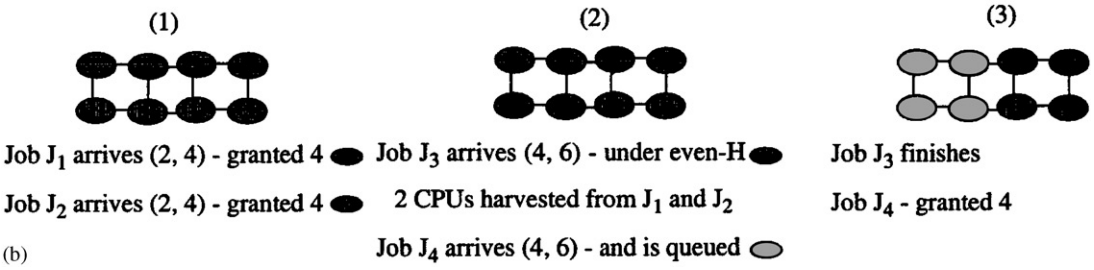
even_harvest: this policy will try to harvest resources evenly from all running applications at each harvest request, irrespective of prior harvesting events. Processors are then taken from each application (one-at-a-time) in a round-robin fashion until the harvested allocation is satisfied. This policy only attempts to be fair within a single harvest event.

fair_harvest: this policy is an extension of *even_harvest* and tries to extend the harvest policy across multiple harvest events. This policy tries to keep the number of harvest events experienced by all running applications the same. It does this by first computing the average number of harvest events suffered by the running applications. Only those applications below this average are candidates for release.

long_harvest: this policy orders applications by their current run length (longest first). Processors are then taken from each application (one-at-a-time) in a round-robin fashion until the harvested allocation is satisfied. Only those applications with an age $> T$ time units are candidate victims. T can be defined as the average age of all running applications.

```
1. determine available resource pool, avail
2. process job arrival, J (min, ideal)
   a) if min <= avail <= max, allocate avail to J
   b) else if deg_of_mult >= M, queue J
   c) else if can harvest* (min - avail) processors, allocate them to J
   d) else queue J in FCFS order
   if a|c, decrement avail
3. process job termination, J releases k processors, avail += k
   a) allocate up to k processors to running jobs (if FR)*
      if any left over, allocate to queued jobs
   b) or allocate up to k processors to queued jobs (if FQ)*
      if any left over, allocate to running jobs
   c) hold any processors as available if not allocated by a or b
```

(a)



(b)

Fig. 3. (a) iScheduler logic. Steps 2 and 3 occur asynchronously. *Depends on harvest/distribution policy selected. The control_policy could be checked at each point a resource management is decision is made (not shown for brevity). (b) Example of *Even-H-FQ* with 8 CPU resources and parallel job stream arriving dynamically. Job is indicated by shaded circle and has (min, ideal) processor requirements. Jobs J_1 and J_2 arrive first to the idle machine and are allocated their ideal resources each (1). Later in (2), J_3 arrives with a min requirement of 4, yet no resources are available. Under *Even-H*, two resources are harvested from J_1 and J_2 allowing J_3 to run. After this, J_4 also arrives with a min of 4 but must be queued as it is not possible to harvest four resources without suspending the other jobs. Finally in (3), job J_3 finishes freeing four resources. Using *FQ*, these resources are given to the queued job J_4.

short_harvest:[1] this policy orders applications by the amount of run time left (largest first). Processors are then taken from each application (one-at-a-time) in a round-robin fashion until the harvested allocation is satisfied. Only those applications with an amount of time left $> T$ time units are candidate victims. T can be defined as the average time left of all running applications.

low_impact_harvest: this policy will try to harvest from the applications that are predicted to be least impacted. Impact is determined by the % of resources that an application will have relative to its desired ideal amount. For example, an application that has two resources allocated with its ideal resources = 4 will be much greater impacted if a resource was taken away than an application with eight resources allocated with its ideal resources = 9 (1/4 vs. 7/9). The applications are ordered by the impact metric and resources are selected using this ordering.

[1] This policy may not be feasible for all applications since the amount of time left may not be known. However, for some parallel applications (e.g. direct solvers) this information can be known. In other cases, historical run-time information can be possibly used. This becomes feasible for iSchedulers that are scheduling parallel job streams of the same type and can collect and exploit this information on-line.

2.2.3. Distribution policies

favor_running: this policy will distribute resources to running applications using the inverse of the harvest policy. For, example, if *low_impact_harvest* was selected, this policy will redistribute to applications that would be the most favorably impacted.

favor_queued: this policy will distribute resources to queued applications to admit them into the system first.

In this paper, the iScheduler policy options we have examined are *even_harvest* and *low_impact_harvest* with *favor_running* and *favor_queued* options for different values of *M*. We call these policies, *Even-H-FR*, *Even-H-FQ*, *Low-Imp-FR*, and *Low-Imp-FQ*. The general iScheduler logic and an example of the *Even-H-FQ* heuristic is shown (Fig. 3). We compared these iScheduler policies against three baselines. The baselines do not allow any modification to the amount of resources held by running applications:

Policy *IDEAL-ONLY* (no adaptation on running applications or arriving applications): arriving application is given its ideal resources; if ideal is not available, application is queued using FCFS.

Policy *MOLDABLE* (no adaptation on running applications): arriving application is given *max* (*min*, *ideal*) resources; if *min* is not available, application is queued using FCFS.

Policy *IDEAL-BACKFILL*: Backfilling will be used to relax FCFS if queue jumping does not delay applications. Each application is granted its ideal resources.

3. Application model and implementation

iScheduler-aware applications must support an adaptive API to enable iSchedulers to implement fluid resource management policies. Otherwise, there are no other special requirements for applications: they can use threads, message passing, etc. Applications are presumed to have a run length that makes iScheduling profitable (minutes or hours as opposed to a few seconds). Consequently, they are assumed to be iterative, with adaptation events initiated and cost information collected, at iteration boundaries. The iScheduler will evaluate the cost and benefit of adaptive actions as part of its scheduling decisions. Clearly, the provision of adaptation within the application implies that it must be *malleable* with respect to system resources. Not all applications are capable of this requirement and thus would not be suitable for iScheduling. However, many parallel scientific applications, can be written or modified to work in this manner. In a recent survey, it was reported that at least 70% of supercomputer jobs could be malleable [6]. An important question is: why would an application writer spend the effort to implement adaptivity, particularly the removal of resources? The first point to stress is that the provision of resource release may actually benefit (or at least not harm) the performance of the application. There are at least four reasons why resource release may be beneficial: (1) resource release may prevent application suspension due to the presence of higher priority jobs or a resource owner reclaiming resources, (2) resource release may not harm performance at the knee of the performance curve (Fig. 2), (3) better resources may become available dynamically and weaker resources could be released, and (4) trading resources between applications owned by the same user or PSE may better meet their overall performance objectives. Even if there is a potential benefit to having adaptivity, the programmer effort can be substantial, though the effort involved depends on the application at hand. Many application and system designers have concluded that individual application adaptivity is worth the effort [15,16,25,26,30]. For threaded parallel applications [6,16] and master–slave applications [15] adaptivity is not hard to implement. Even for parallel message-passing programs, we have built applications that support adaptivity and showed it can be done with modest effort [30]. Another argument to make is that if the programmer is willing to expend the effort to have resources dynamically added to their application (a clear advantage in many cases), then the same adaptation code can be easily transformed to allow for resource release because issues relating to data redistribution, communication, load rebalancing, are the same in both cases.

We have divided the adaptive API into two components, *sensing* and *actuating*. Sensing functions provide performance information to the iScheduler and actuating functions give the iScheduler the ability to control the application's behavior in support of fluid resource management. The sensing functions may provide two kinds of information—a prediction of future performance and measured values of past performance. Prediction is useful for making an initial static decision (e.g. to determine the ideal number of resources), but the measured values provide dynamic run-time information that may be needed to tune the decision in the following situations: (1) application performance is hard to characterize statically (e.g. the application consists of several distinct phases with vastly difference resource requirements), (2) resource availability may fluctuate due to external users, and (3) the iScheduler wishes to share resources with other applications and may dynamically add or remove resources from the application. The sensing functions are as follows:

exec_time (&future_iters, &past_iters): returns predicted average per iteration execution time (includes computation, communication, I/O, etc.) for future_iters; and/or returns measured average per iteration execution time over last past_iters.
exec_time_comp (&future_iters, &past_iters): returns predicted average per iteration computation time for future_iters; and/or returns measured average per iteration computation time over last past_iters.
exec_time_comm (&future_iters, &past_iters): returns predicted average per iteration communication time for future_iters; and/or returns measured average per iteration communication time over last past_iters.
resources: returns the *min* and *ideal* value for the amount of resources in several categories (number of CPUs, amount of memory, amount of local disk).

The *resources* sensing function provides information that can help the iScheduler make resource allocation decisions. The minimum value is particularly important as it serves as a constraint on resource allocation: the application must have at least the minimum amount to run. The ideal amount is a hint and is not required, though some application schedulers may be available to provide accurate values for it. For example, it can be determined from the sensing functions for regular data parallel applications as will be shown.

Actuating functions are used to adjust the resource consumption of applications in support of fluid resource management. Sensing provides useful information to the iScheduler that can be used in the actuating process. For

example, the iScheduler can use dynamic sensing information to adjust or tune scheduling decisions. The actuating functions are as follows:

add_processor (num_to_add)
remove_processor (num_to_remove)
migrate_processor (old_proc, new_proc)
actuating_cost_benefit (adaptation_method, num_procs, &cost, &benefit)

The iScheduler invokes these functions within the application to control its resource usage. For example, it may seek to remove processors from an application in order to give them to another, perhaps newly arriving application. It may also allow an application to gain additional processors when they become available such as when an application finishes. These functions represent basic mechanisms for resource adaptation within the application. How these functions are used is a policy decision made by the iScheduler. Predicting the cost and benefit of adaptation to the application is an important ingredient in building efficient iSchedulers. The *actuating_cost_benefit* function exports this information to the iScheduler.

Performance prediction is fundamental to the application model, the iScheduler, and the adaptive API. It answers the questions: how will the application perform on a given set of resources and what is the cost–benefit of adaptation in support of fluid resource management? In prior research, we have had success predicting performance for three models common to parallel scientific applications that use message passing: *SPMD data parallel*, *pipelines*, and *master–slave bag-of-tasks* using the Prophet system [28]. Prophet schedules parallel applications with the objective of reduced completion time. It has been successfully applied to a wide range of applications including parallel gene sequence comparison, electromagnetic scattering using finite elements, and image processing pipelines. Prophet predicts the optimal number of processors for a given parallel application by constructing cost functions that predict application performance. It explores a sequence of processor configurations to find the knee of the curve (Fig. 2). Prophet was modified to export internal cost information to the iScheduler. This cost information includes application performance on a candidate set of resources and the minimum number of required resources. In addition, Prophet also contains models that accurately predict the cost and benefit of adding, removing, migrating, and dynamically load balancing processors [30]. The prediction model achieved high accuracy for both the cost and benefit of adaptation: within 10% for a large variety of applications. The latter two adaptation methods (migration and dynamic load balancing) will be particularly useful when the environment contains non-iScheduler-aware applications, a topic of future investigation.

comm_topology **: 1-D, tree, ring, broadcast**
comm_complexity **: avg message size in bytes**
numDEs **: number of data elements (DE) in the problem**
comp_complexity **: # of executed instructions per DE**
arch_cost**: arch-specific execution cost per instruction for a data element (usec/instruction)**
iters**: number of iterations if known**
min_requirements**: minimum resources required for execution**

Fig. 4. Callbacks. (a) lists the callback functions. The *comm_topology* is used to select from a set of topology-specific communication functions determined by off-line benchmarking. The *arch_cost* is also determined by off-line benchmarking and returns values for each machine type in the network.

In Prophet, the cost models it constructs rely on information provided by the application in the form of *callback* functions. The callbacks provide information for each computation and communication phase of a parallel program. A simple example is shown for a Jacobi iterative solver (Fig. 4). Because they are functions, the callbacks can capture simple data-dependent behavior (e.g. based on problem size). From these callbacks, cost functions that predict the average per-iteration execution time ($\overline{T_{exec}}$), computation time ($\overline{T_{comp}}$) and communication time ($\overline{T_{comm}}$)[2] can be defined. An example of the Prophet cost functions for a message-passing data parallel application is shown (p is the number of candidate processors, *lat* and *bw* are latency and bandwidth communication constants that are determined off-line):

$$(\overline{T_{comm}}) = lat + (bw \cdot p \cdot comm\_complexity)/(\overline{bw\_avail})$$

$$(\overline{T_{comp}}) = (comp\_complexity \cdot \overline{arch\_cost} \cdot numDEs)/(p \cdot \overline{cpu\_avail})$$

$$(\overline{T_{exec}}) = \overline{T_{comm}} + \overline{T_{comp}}$$

For other applications such as threaded-parallel, the formulation is different. The accuracy of Prophet's cost models is generally high even for complex multi-phase applications. This callback information is used to support the adaptive API. For example, $\overline{T_{exec}}$ is used to implement the *exec_time* function as part of the sensing API. This information is also used to help determine the cost and benefit of adaptation. Continuing with the data parallel example, the benefit of adding a processor can be determined by plugging $p+1$ into the equations above. The cost of adding a processor is more complex, but is determined, in part, by the cost of data communication. This can be computed using a

[2] Communication function shown is for a shared 10 base-T ethernet network in which bandwidth is shared among the p communicating processors (hence the dependence on p). In a shared environment, communication and computation cost depends on the amount of available bandwidth the CPU cycles, respectively.

communication cost function (e.g. $\overline{T_{comm}}$ but with *comm_complexity* replaced by the amount of data to be redistributed). The adaptation cost model relies on other parameters not shown here and is omitted for brevity, details may be found in [30].

We have used Prophet callbacks to implement part of the sensing functionality in support of the adaptive API. In addition, the resource requirements of the application are also provided by callbacks. For example, *min_requirements* reports the *min* resources needed and the *ideal* resources can be determined internally within Prophet via its scheduling algorithms. Both of these are returned by the *resources* sensing function to the iScheduler. This is treated as a hint to the iScheduler because all application schedulers will not be able to accurately provide this information. The Prophet integration demonstrated that existing schedulers can be incorporated into the iScheduler framework with minimal difficulty.

4. Results

To evaluate the potential of the iScheduler paradigm, we have built an iScheduler framework and used it to construct several user-centric and system-centric iSchedulers which both were evaluated by simulation. The framework is currently being used as part of a "live" scheduler for parallel Prophet jobs on cluster networks. Here we report results on the simulation experiments. The user-centric iSchedulers were applied to schedule a stream of parallel jobs submitted from a single user and use the Prophet scheduling system internally to generate cost information needed for the trace-driven simulation. These jobs were initially run on our small cluster (an ATM cluster of 8 Sparc 20 nodes) to generate real cost information. Simulation allows us to "run on" larger clusters (e.g. at least 64 nodes) with larger problem sizes to produce more meaningful results than can be obtained on a small cluster. The system-centric iScheduler was applied to supercomputer workloads in a trace-driven simulation. For both user- and system-centric iSchedulers, we have evaluated the *IDEAL-ONLY*, *MOLDABLE*, *IDEAL-BACKFILL*, *Even-H-FR*, *Even-H-FQ*, *Low-Imp-FR*, and *Low-Imp-FQ* scheduling policies. The performance metrics that we used for comparison were average waiting time, average execution time, and total completion time.

4.1. User-centric workload model

We built user-centric iSchedulers for three data parallel applications, an iterative jacobi solver (STEN), Gaussian elimination with partial pivoting (GE), and parallel gene sequence comparison (CL), the details of these applications are described elsewhere [28]. These applications were modified to operate with the iScheduler (i.e. they implement the adaptive API) and represent a useful initial test of the applicability of the paradigm: the applications were not master–slave, but message-passing programs. CL, in particular, is a real application. The workload consisted of instances of each application with differing problem sizes and run lengths, as would be expected in a pool of submitted jobs. This scenario would be appropriate for a related set of parallel jobs submitted by a user or the workload requests to a PSE that offered fixed parallel services [5,20]. In the supercomputer workload traces we examined, jobs submitted from the same user generally fell into three categories: batch or simultaneous submission, one-job-at-a-time submission, and overlapping submission. We modelled each of these cases by adjusting the interarrival rate of the application instances (P is the total number of processors, $\overline{RT_{ideal}}$ is the average run-time for all application instances given their ideal number of processors, $\overline{P_{ideal}}$ is the average ideal number of processors for all application instances, and β is a constant):

$$inter\_arrival\_rate = \frac{\overline{RT_{ideal}}}{P/\overline{P_{ideal}}} \cdot \beta, \quad \text{where } \beta \text{ is } [0,1].$$

The values for $\overline{P_{ideal}}$ and $\overline{RT_{ideal}}$ were determined by Prophet. Larger values of β reflect a smaller load or wider gap between application arrival. For example, if β is 1 then applications should be able to run (approximately) without delay. This is used to represent one-job-at-a-time submission. For smaller values of β, we can emulate overlapping submission. We selected values of 0.25, 0.5, 0.75, and 1.0 for our study. We generate several traces based on four instances each of STEN, GE, and CL (Table 1). Execution information was obtained from our Sparc Cluster using the predictive cost models of Prophet. Four trace templates were used to generate traces: STEN-ONLY, GE-ONLY, CL-ONLY, and MIXED. Each problem instance was generated with equal likelihood, so, e.g. a STEN-ONLY trace would contain an approximately equal number of P1, P2, P3, and P4 instances. For a MIXED trace, each application instance from STEN, GE, and CL are equally likely. On our Sparc cluster running 100 Mb ethernet, we also determined the cost of shipping M bytes of data to be approximately: $1 + 0.00009\,M$. For an adaptive event (either add or remove), data is first sent to a master processor and then redistributed. The amount of data (M) moved is the following:

STEN: $2N^{2*}8/P$, GE: $2N^{2*}8/P$, and CL: 2NumSourceSeqs*SeqBlockSize/P.

For example, the overhead of removing a processor for STEN with $N = 128$ at $P = 4$ would be approximately 7 s as compared with 1289 s of runtime. The cost of adding is a little higher since process creation is needed, but this is a few hundred milliseconds at worst.

Table 1
Problem instances

Application	$P1$	$P2$	$P3$	$P4$
STEN	$N=128$	$N=256$	$N=512$	$N=1024$
	$P_{ideal}=4$	$P_{ideal}=8$	$P_{ideal}=10$	$P_{ideal}=14$
	$RT_{ideal}=1289$ s	$RT_{ideal}=6359$ s	$RT_{ideal}=24\,758$ s	$RT_{ideal}=113\,824$ s
GE	$N=128$	$N=256$	$N=512$	$N=1024$
	$P_{ideal}=6$	$P_{ideal}=8$	$P_{ideal}=12$	$P_{ideal}=18$
	$RT_{ideal}=177$ s	$RT_{ideal}=768$ s	$RT_{ideal}=3857$ s	$RT_{ideal}=3857$ s
CL	NumSourceSeqs = 25	NumSourceSeqs = 50	NumSourceSeqs = 100	NumSourceSeqs = 200
	NumTargetSeqs = 100	NumTargetSeqs = 100	NumTargetSeqs = 100	NumTargetSeqs = 100
	$P_{ideal}=10$	$P_{ideal}=16$	$P_{ideal}=24$	$P_{ideal}=32$
	$RT_{ideal}=4815$ s	$RT_{ideal}=6666$ s	$RT_{ideal}=23\,622$ s	$RT_{ideal}=244\,800$ s

An issue for future investigation is the issue of how iScheduling will scale to large, possibly multi-phase applications that may be dynamic or irregular in structure. There is nothing within the paradigm in terms of mechanism that precludes extension to larger applications, but it is likely that new iScheduling policies will be needed. For example, it is possible that different iScheduling policies might be required for distinct phases of the application. The ability to add and release resources will be particularly useful for applications with different resource requirements in different phases. For irregular data-dependent applications, the minimum and ideal number of resources may also be data-dependent. Since these values are provided by the *resources* sensing function, this function can be programmed to be return values based on the input data.

4.2. System-centric workload model

We built a system-centric iScheduler and used supercomputer center traces from the SDSC IBM SP2 (128 nodes), LANL SGI Origin 2000 (2048 nodes), and CTC IBM SP2 (512 nodes) to generate workloads [10]. Our model generates synthetic job traces by modelling the statistics of these published traces in terms of job interarrival times, job size, and job run times. We observed that for interarrival, more jobs arrive to the system during the day than during the night (Fig. 5). In order to capture the differences in the arrival rates, we used a non-stationary Poisson process to model the job interarrival times in which the arrival rate λ is a function of time. The interarrival times are generated from an exponential distribution, the mean of which, $1/\lambda$, varies according to the (simulated) time of day (Fig. 6). The x-axis represents the bins into which the interarrival times are placed. Each bin is 500 s in duration. Most of the interarrival times fall into the first bin, which indicates that most jobs arrive within 500 s of the preceding job. While the fit is not perfect, it captures the initial spike and tail-off effect. For our workload model, the size of a job is characterized by the number of processors it requests. In the actual workload traces, there are a few

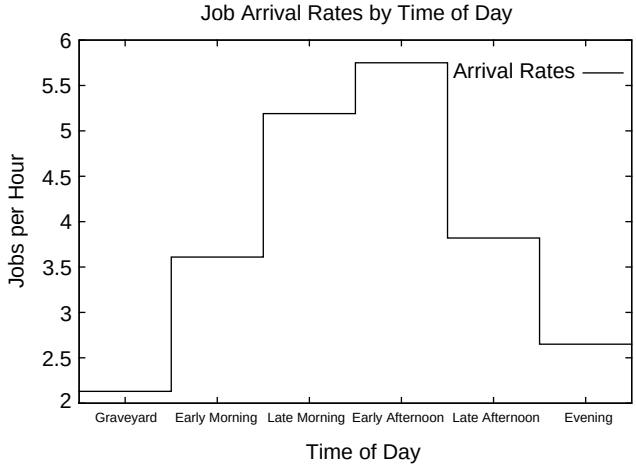


Fig. 5. Job arrival rates by time of day.

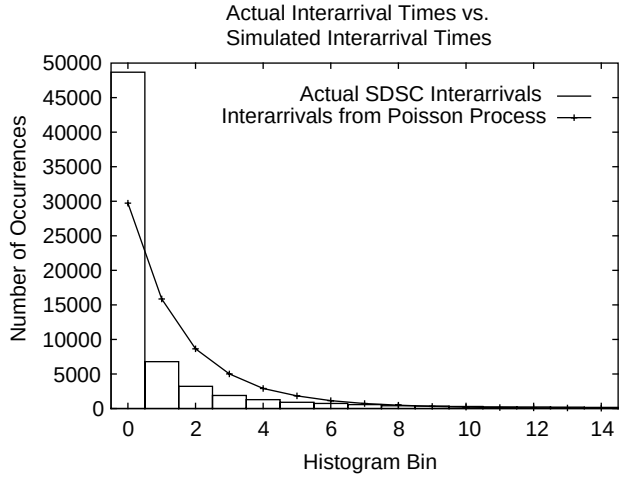


Fig. 6. Interarrival times: actual vs. model for SDSC.

job sizes that are very common. The common job sizes are 1, and the *power-of-two* sizes 2, 4, 8, 16, 32, and 64. All other job sizes occur infrequently. For our simulated workloads, we use a discrete probability distribution that reflects the job sizes in an actual workload (Fig. 7). Job run times have been modelled by a Weibull distribution (Fig. 8). The figure shows a histogram of all job run times from the SDSC workload plotted

against the Weibull distribution. Like the histograms for the interarrival times, the bin sizes are 500 s. Most jobs have short run times, while a few jobs run for very long periods of time. We use a single Weibull distribution with estimated shape parameter α and scale parameter β to model job run times for our simulated workloads.

Each generated job in the trace has an arrival time (Fig. 6), an ideal number of processors (Fig. 7), and an execution time given its ideal number of processors (Fig. 8).

4.3. Adaptation model

For each application in a trace, we know how long it will take provided it is given its ideal or desired resources. We assume these are the values available in the trace via Fig. 8 or Table 1. However, for several scheduling policies including *MOLDABLE* and iScheduling policies, the execution time must be adjusted if an amount of resources less than the ideal is granted. Furthermore, the cost of harvesting must also be included since the application is suspended while resource re-allocation is being performed. For the experimental applications, an accurate measure of adaptation overhead has been developed [30] and was incorporated into simulation of the user-centric iSchedulers. Adaptation overhead is highly application-dependent. It depends upon the amount of data transferred to achieve load balance which in turn depends on the problem size and number of processors. It includes the cost of data redistribution on the cluster network and process creation (for adding processors to the application). However, for the supercomputer workload traces, the jobs are essentially black boxes so this overhead cannot be known. However, the average run length of these applications is in the 1000s of seconds and we have determined that adaptation overhead is typically on the order of a few seconds at worst [30]. In addition, our results indicate that the number of adaptive events is fairly small (1–3 per application). The small jobs are typically single CPU jobs (ideal = min = 1) which means they do not suffer adaptations, but can benefit from them. Therefore, the overhead is ignored for the system-centric iScheduler simulation, but is included for the user-centric iScheduler simulation. In addition, the minimum amount of CPU resources is also not available from the traces. Instead, we set this as a parameter, *min_%_of_ideal*, which sets the minimum number of required resources as a percentage of its ideal. For example, if *min_%_of_ideal* is 10%, and *ideal* was 100, then *min* would be 10. We vary this parameter in generating the workload traces.

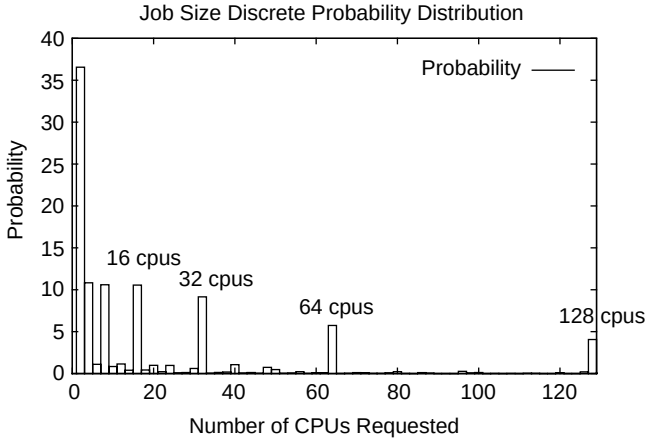


Fig. 7. Discrete probability distribution for job sizes.

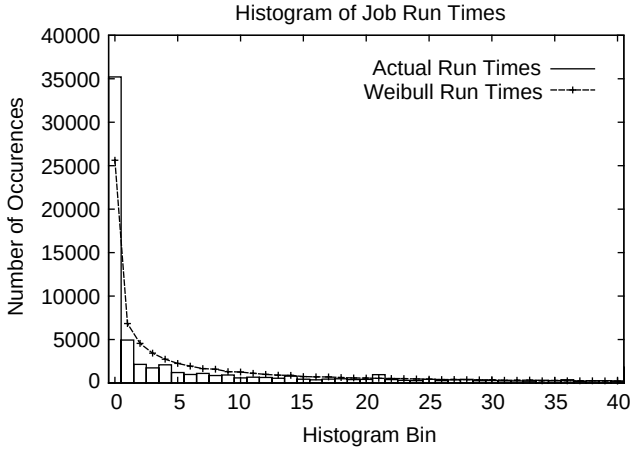


Fig. 8. Actual job run times vs. Weibull distribution.

Adjusting application run length as a function of the number of resources is also an issue for the supercomputer jobs since they are black boxes. To address this issue, we have implemented two common execution models within the simulation, (i) linear and (ii) parabolic. The linear model does not consider communication and assumes that execution time is linear in P. The parabolic model assumes there is communication and that the ideal number of processors (P_{ideal}) is the true minimum execution time. This leads to the classic parabolic shape of execution time vs. processors (Fig. 2). The real parallel applications are parabolic, so this model is used for the user-centric iScheduler. The execution time for both models is as follows (P is the number of processors):

$$exec\_time\_linear = \frac{P_{ideal} \times RT_{ideal}}{P},$$

$$exec\_time\_parabolic = \frac{a}{P} + bP, \quad \text{where } a = \left(\frac{P_{ideal} \times RT_{ideal}}{2}\right),$$

$$b = \frac{RP_{ideal}}{2P_{ideal}}.$$

If P changes during the execution of the application (if iScheduling is enabled), then these functions are

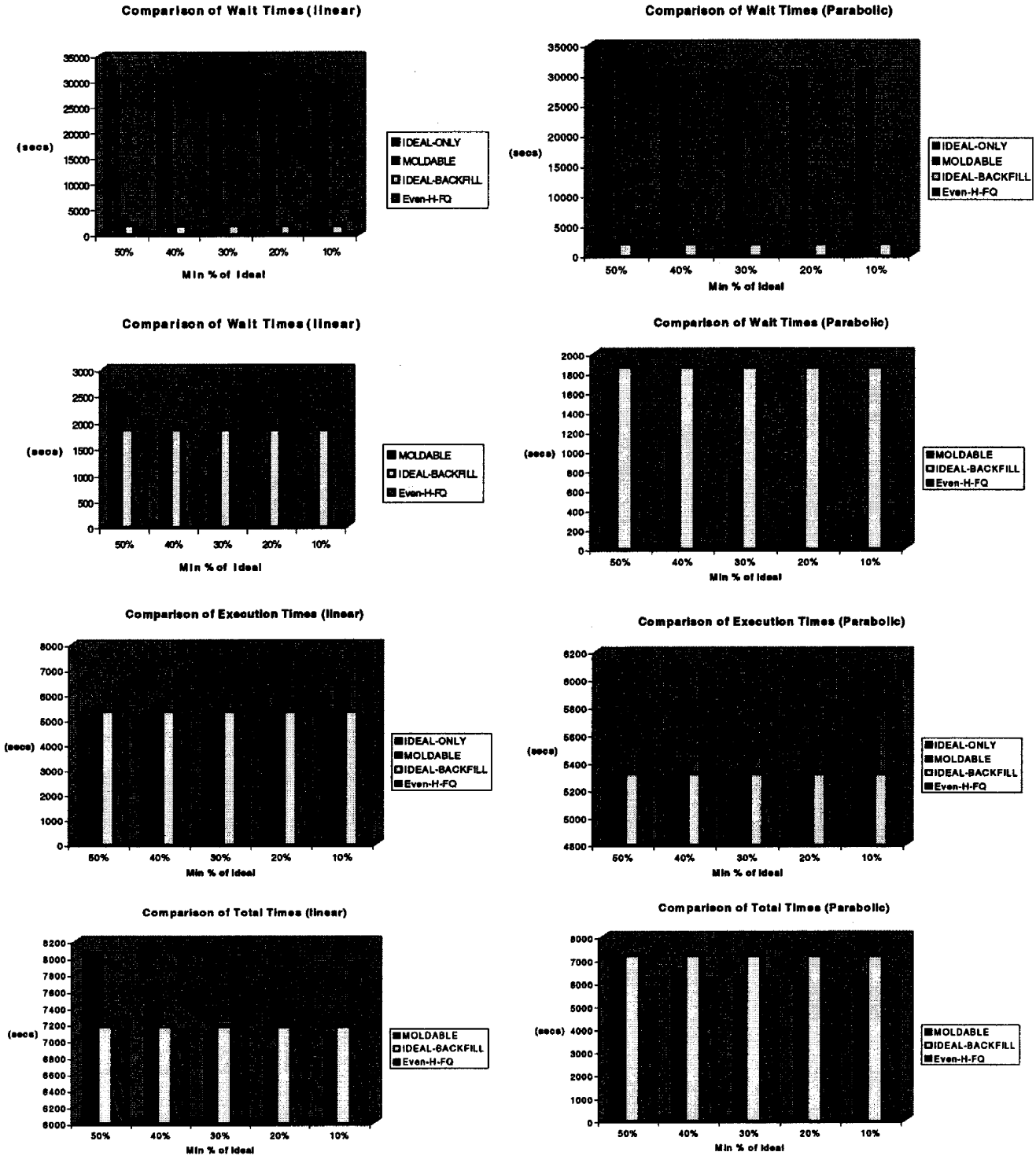


Fig. 9. Comparative performance of iScheduling vs. baselines.

applied for each value of P over the appropriate duration.

4.4. Simulation results

We examine three basic questions in this study. First, does the iScheduling paradigm provide a performance benefit over baseline policies? Second, what is the most effective iScheduling policy given the large space of options. Third, what is the sensitivity of iScheduler performance to workload and system characteristics. We determined the performance sensitivity to M, the degree of multiprogramming, and to *min_%_of_ideal*, the minimum allowable % of ideal resources. In general,

M will be varied from 3,6,9,12, and *no_limit*, and *min_%_of_ideal* from 50%, 40%, 30%, 20%, and 10% of ideal. We generated 100 traces each containing 10 000 jobs for each simulation experiment. The values presented are the average of the $100 \times 10\,000$ jobs.

4.4.1. System-centric iScheduling

In the initial comparison, the baselines, *IDEAL-ONLY*, *IDEAL-BACKFILL*, and *MOLDABLE*, are compared against the simplest harvest-based policy, *Even-H-FQ* to see whether harvest-based policies are competitive with standard policies (Fig. 9). The simulated machine contains 128 processors. We show results for both the linear and parabolic execution models. To enable a fair comparison with the baselines, M is set to be *no_limit* for *Even-H-FQ*, as this is the default settings for the baselines. It is evident immediately that *IDEAL-ONLY* performs quite badly with respect to wait time since it requires waiting for the ideal number of resources in a FCFS manner. Since the waiting time of *IDEAL-ONLY* dominates the total time (wait time + execution time), we omit it from the remaining graphs in Fig. 9 as it is significantly worse than the other policies (we will return to this policy at the end of this section). The results show that *Even-H-FQ* achieved significantly lower waiting time than all of the baselines, but that execution time goes up because applications may be run with fewer resources under iScheduling. However, total completion time is better for iScheduling for certain values of *min_%_of_ideal* (larger values), but not for all values. This establishes that iScheduling (at least, *Even-H-FQ*) performs favorably for certain values of M and *min_%_of_ideal*. The final results will show that an iScheduling policy can be defined that will outperform ALL baselines for all values of M and *min_%_of_ideal*.

We now compare several harvest-based policies head-to-head. The comparative performance of the scheduling policies did not differ significantly between the linear and parabolic application execution models. Therefore in the remaining graphs for the system-centric iScheduler, we present results for the linear model only. We show waiting time and total time (waiting time + execution time) as the key metrics of interest. In the first set of results, we compare the waiting time under the four scheduling policies and their sensitivity to the degree of multiprogramming M (Fig. 10). The results indicate that when *min_%_of_ideal* is 50%, *-*FQ* outperforms *-*FR* for $M = 3$ and 6. When it is 40%, *FQ* outperforms *FR* for $M = 3$, 6, and 9. When it is 30%, *-*FQ* outperforms *FR* for $M = 3$, 6, 9, and 12. The trend is clear: as *min_%_of_ideal* decreases, *-*FQ* outperforms

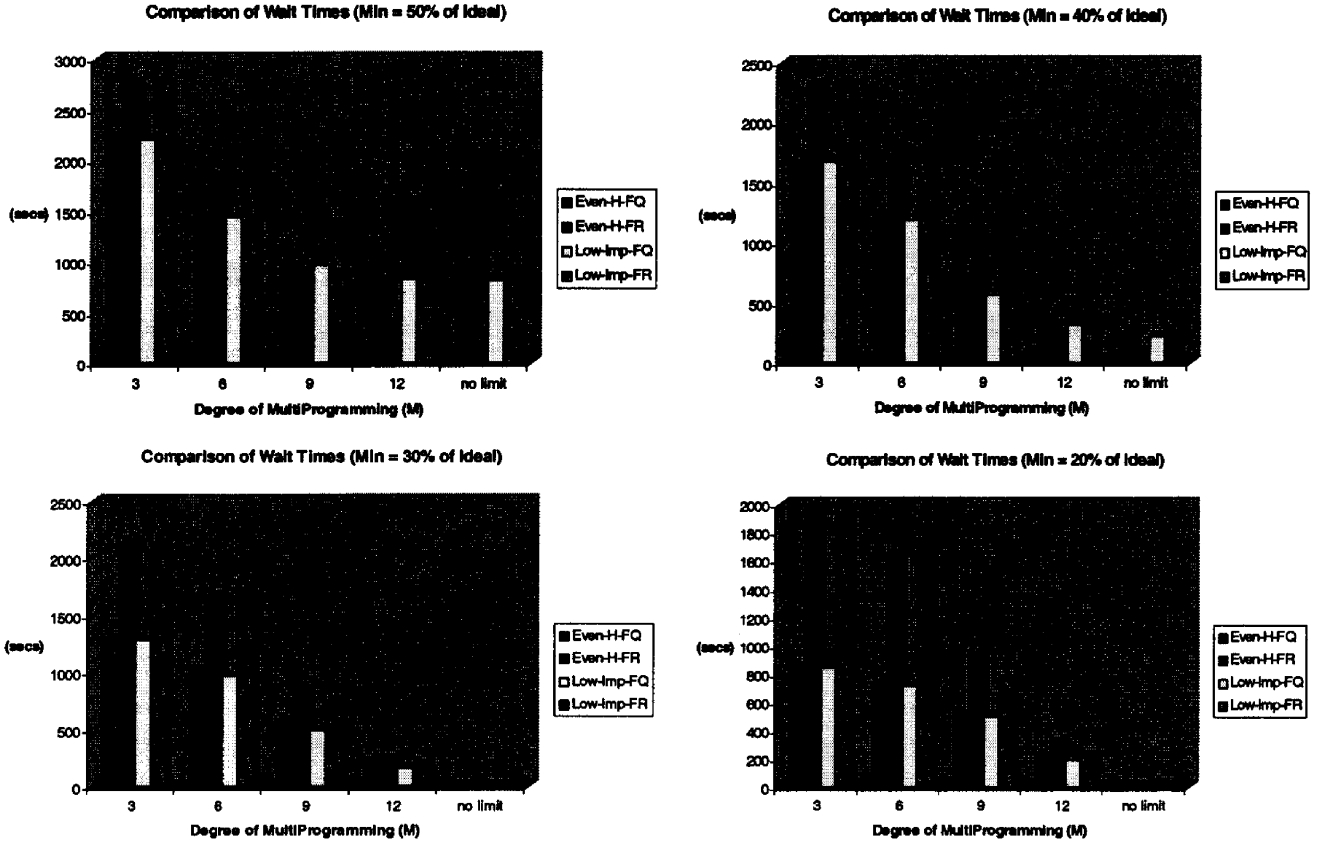


Fig. 10. Wait time comparison—sensitivity to M.

*-*FR* for a wider range of *M*. The reason is that when *min_%_of_ideal* decreases, more jobs are likely to be eligible to enter the system, hence *-*FQ* is able to exploit this opportunity more easily.

In general, as *M* increases for all policies, the waiting times decrease since the system is more likely to admit jobs onto the machine. When *M* increases to the point of *no_limit*, then the waiting times all converge to the same small value. The reason is that when a job finishes, the scheduling policies first try to satisfy the constraint on *M*, whether it is *-*FR* or *-*FQ*. Once the *M* constraint has been met, then either running or additional queued jobs are considered. So a value of *no_limit* is essentially the same as *-*FQ*. The performance of *Low-Imp-** is slightly better than *Even-H-** which is not surprising considering that it considers the impact on the harvested application. However, the performance is similar enough to suggest that simple policies such as *Even-H-** are effective. However, it is clear that the *FQ* option reduces wait time.

The next metric to observe is the total completion time dependence on *M* (Fig. 11). The *-*FR* policies achieve better overall performance when compared with *-*FQ* as *M* increases. As *M* increases, the *-*FR* performance also decreases. However, for the *-*FQ* policies performance generally decreases as *M* increases to a point, and then increases. Where the minimum occurs, depends on *min_%_of_ideal*. At low values of *M*, *-*FQ* is superior to *-*FR* policies, but as *M* increases the *-*FR* policies are better. In general, if low waiting time is the desired metric, then either scheduling policy with the *FQ* option should be used. However, if reduced total time is desired, then either scheduling policy with the *FR* option will generally perform better than *FQ* at high degrees of multiprogramming *M*. The results suggest we should be running at high *M* since this yields reduced total time. *F*R is superior to *FQ* at high *M* since a high value of *M* admits more jobs into the system which in turn means that many jobs will be running below their ideal number of processors. Therefore, a distribution strategy that will favor these running jobs is better than giving additional greater priority to queued jobs (a large *M* already favors queued jobs to some extent).

The next question we investigated was the comparative performance of the scheduling policies as a function of *min_%_of_ideal.* First, we show the waiting time as *min_%_of_ideal* is varied for fixed values of *M* (Fig. 12). The results show that in general as *min_%_of_ideal*

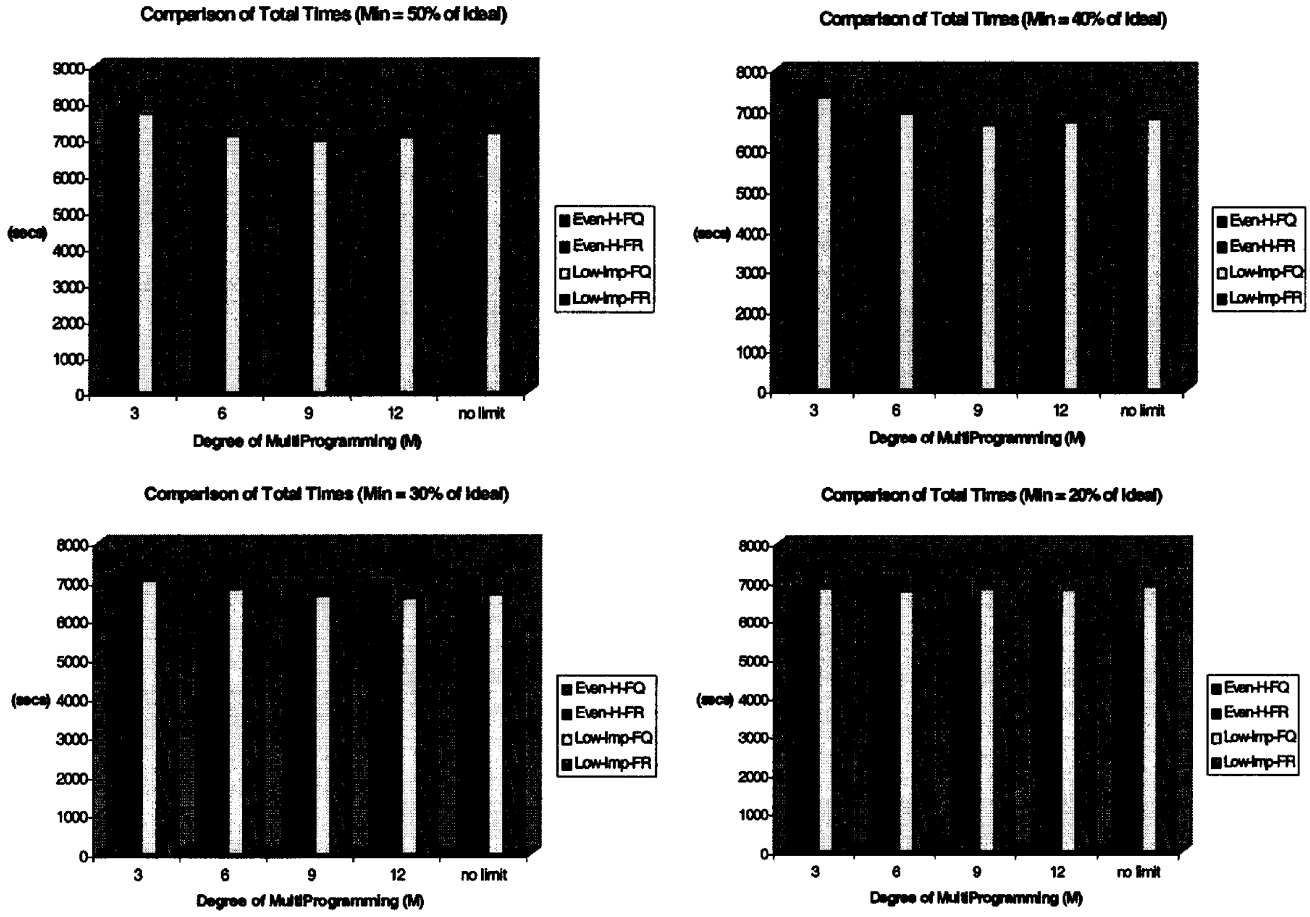


Fig. 11. Total time comparison—sensitivity to *M*.

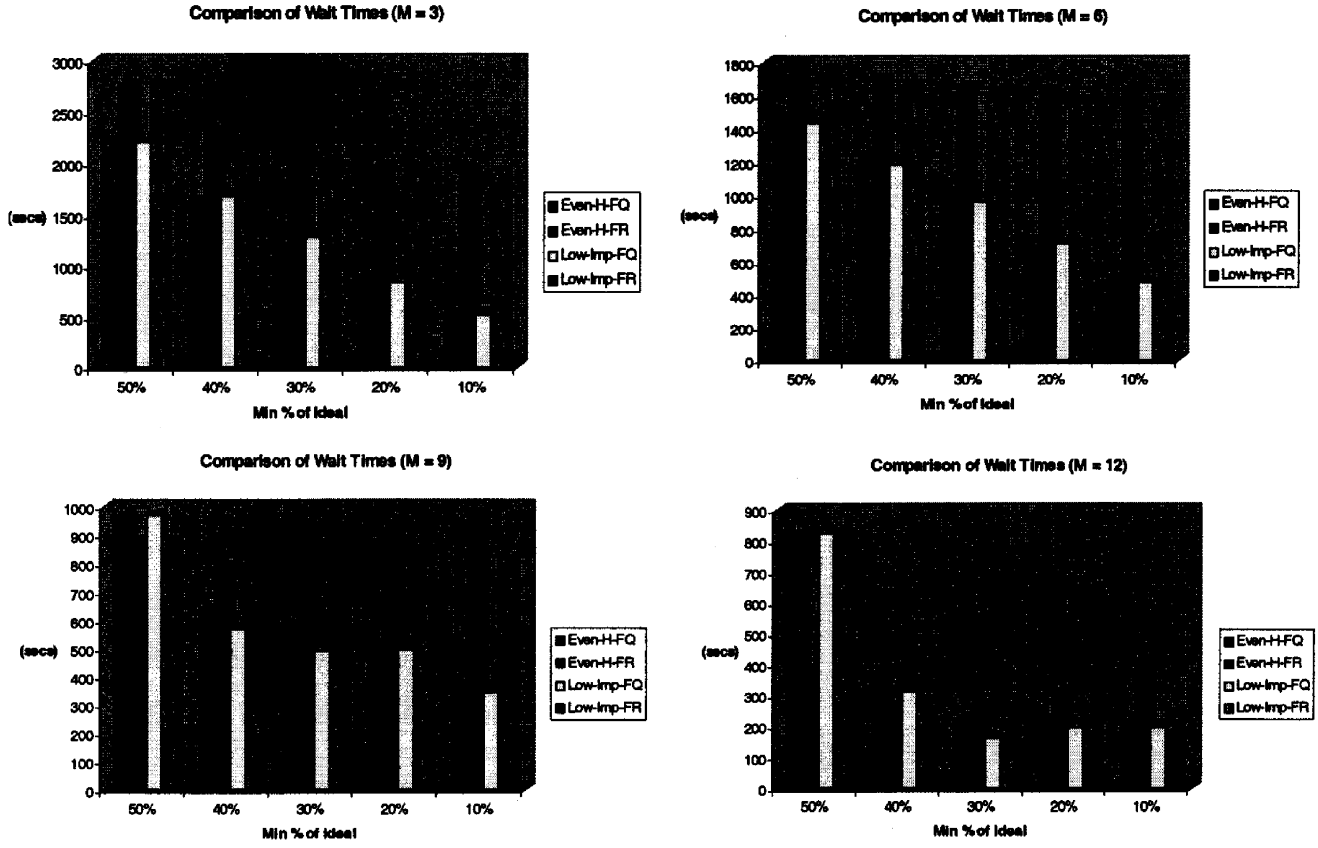


Fig. 12. Wait time comparison—sensitivity to *min_%_of_ideal*.

decreases, the waiting times for the *-*FQ* policies decrease. This trend is also observed for *-*FR* under low degrees of multiprogramming, but as the degree of multiprogramming increases for *-*FR* policies, a decrease in *min_%_of_ideal* leads to higher waiting times. The reason is that as *min_%_of_ideal* decreases there are more running jobs that are eligible to receive additional resources particularly for larger M. Hence, once M is satisfied, *-*FR* is more likely to distribute resources to running jobs, increasing the wait times of jobs on the queue.

The next metric to observe is the total completion time dependence on *min_%_of_ideal* (Fig. 13). It is clear from the graphs that *Low-Imp-FR* for $M >= 9$ is best irrespective of *min_%_of_ideal*. However for smaller values of M, *Low-Imp-FQ* is better for larger values of *min_%_of_ideal*. For $M = 3$, *Low-Imp-FQ* is clearly better. For $M = 6$, *Low-Imp-FQ* is better until *min_%_of_ideal* hits 10%. Returning to the baselines, we now plot the baselines vs. the best overall harvest-based policy *Low-Imp-FR* for $M = no\_limit$ (Fig. 14). From the results, we see that *Low-Imp-FR* improved waiting time by over a factor of 70 relative to the *IDEAL-ONLY* baseline (the default scheduling policy) and a factor of 5 or more relative to *IDEAL-BACKFILL* and *MOLDABLE*. It improved total completion time by a factor of 7 relative to *IDEAL-ONLY* and approximately 20% as compared to both *IDEAL-BACKFILL* and *MOLDABLE*. Finally, we present some summary statistics that show why the harvest-based policies are generally superior (Table 2). The results shown are an average over all values of M and *min_%_of_ideal*. The results indicate that harvesting was successful the vast majority of the time (98% or better), but that most jobs did not suffer more than a few harvest events (1–3 on average). This contributes to the success of this paradigm relative to the baseline policies.

4.4.2. User-centric iScheduling

The user-centric iScheduler experiments agreed with the system-centric results in that *Low-Imp-FR* was also the overall best and so we only show the comparison from this policy to the baselines. Since we are comparing to the baselines, M is *no_limit*. The simulated cluster machine contained 64 processors, and we show the performance dependence of the interarrival rate parameter β and *min_%_of_ideal*. The application execution model is parabolic since these applications all had parabolic execution times. In this section, we show GE and the MIXED workload only (the pattern for STEN and CL is similar) for brevity. As with the system-centric iScheduler, user-centric iScheduling can also

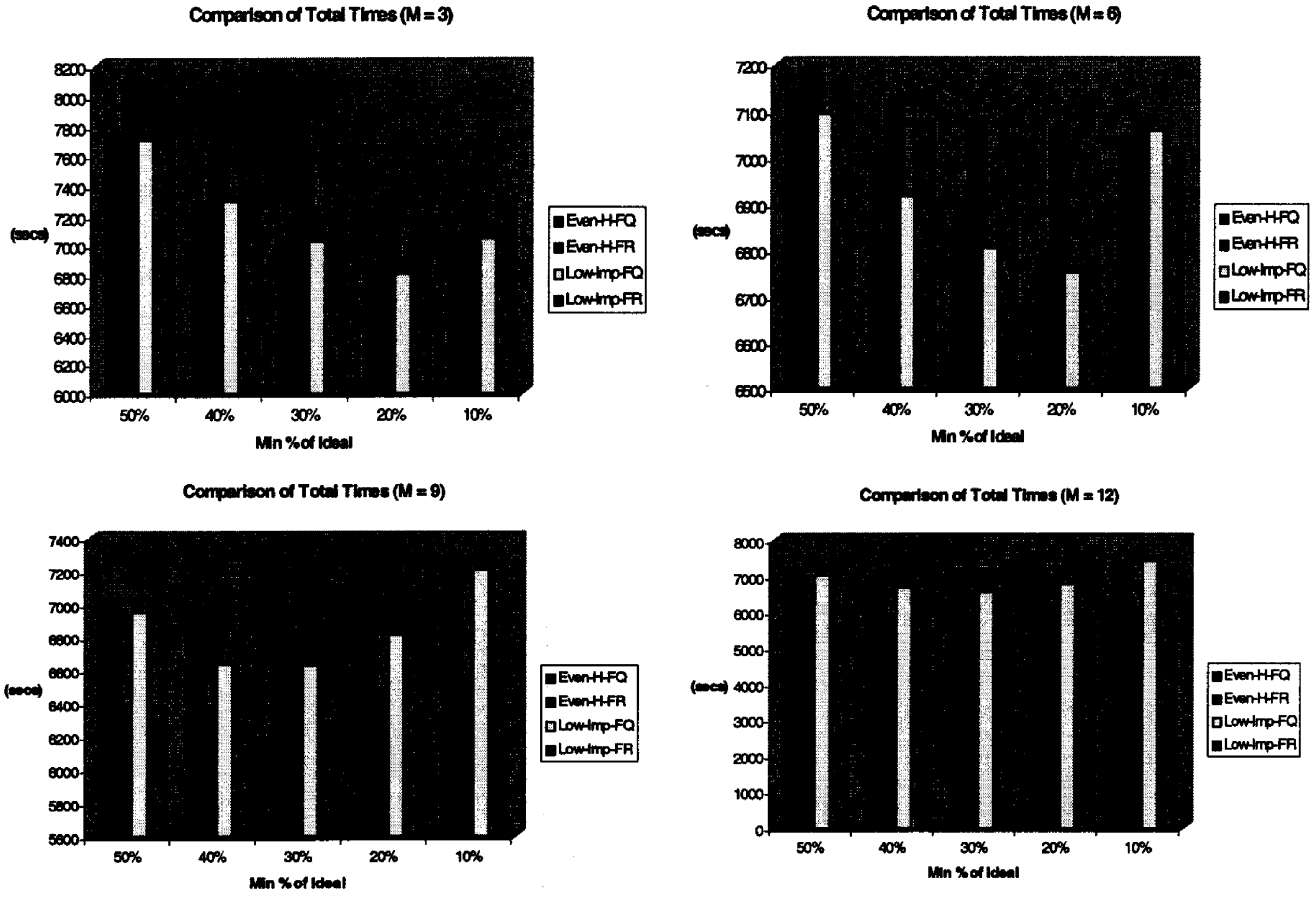


Fig. 13. Total time comparison—sensitivity to *min_%_of_ideal*.

substantially reduce waiting time (Fig. 15). As β decreases, the effective interarrival rate increases, putting more pressure on the available resources. As β approaches 1, the jobs are arriving with enough spacing to increase the probability that they can get their ideal resources without adapting. iScheduling policies improved waiting time for all values of β but the improvement increases as β increases. Performance improvement ranges from a factor of 2 to 20 with respect to *IDEAL-ONLY*, and from 10% to a factor of 5 as compared with the other baselines. For all applications, iScheduling improved the total time relative to all other policies (Fig. 16). Performance improvement ranges from a factor of 2 to 10 with respect to *IDEAL-ONLY*, and from 10% to a factor of 3 as compared with the other baselines. As β decreases, iScheduling is able to achieve better performance in the face of increased resource contention. Finally, we present throughput results for a representative application GE as a function of β and *min_%_of_ideal* (Table 3) that includes the adaptation overhead. The throughput results confirm the iScheduling is generally superior to all methods. As the *min_%_of_ideal* decreases, iScheduling is able to exploit this, allowing more jobs to enter the system and more jobs to finish per unit time. Moldable is also able to exploit this, but when *min_%_of_ideal* becomes too small (10%), moldable suffers because it has no way to replenish running jobs with resources. iScheduling is more sensitive to β increasing (more spacing between jobs) because it is able to keep up with the job stream at higher arrival rates. More spacing injects extra delay.

Collectively, the system- and user-centric workload models are varied enough to suggest that the iScheduling paradigm appears to have great promise as a new scheduling technique for improving the performance of parallel applications in space-shared parallel machines and clusters.

5. Related work

The concept of integrated scheduling is complementary to the large body of dynamic scheduling research for parallel applications. Application-level scheduling approaches such as [3,28,29] each propose policies and mechanisms to make resource allocation decisions that are deemed best for the application. They are focused

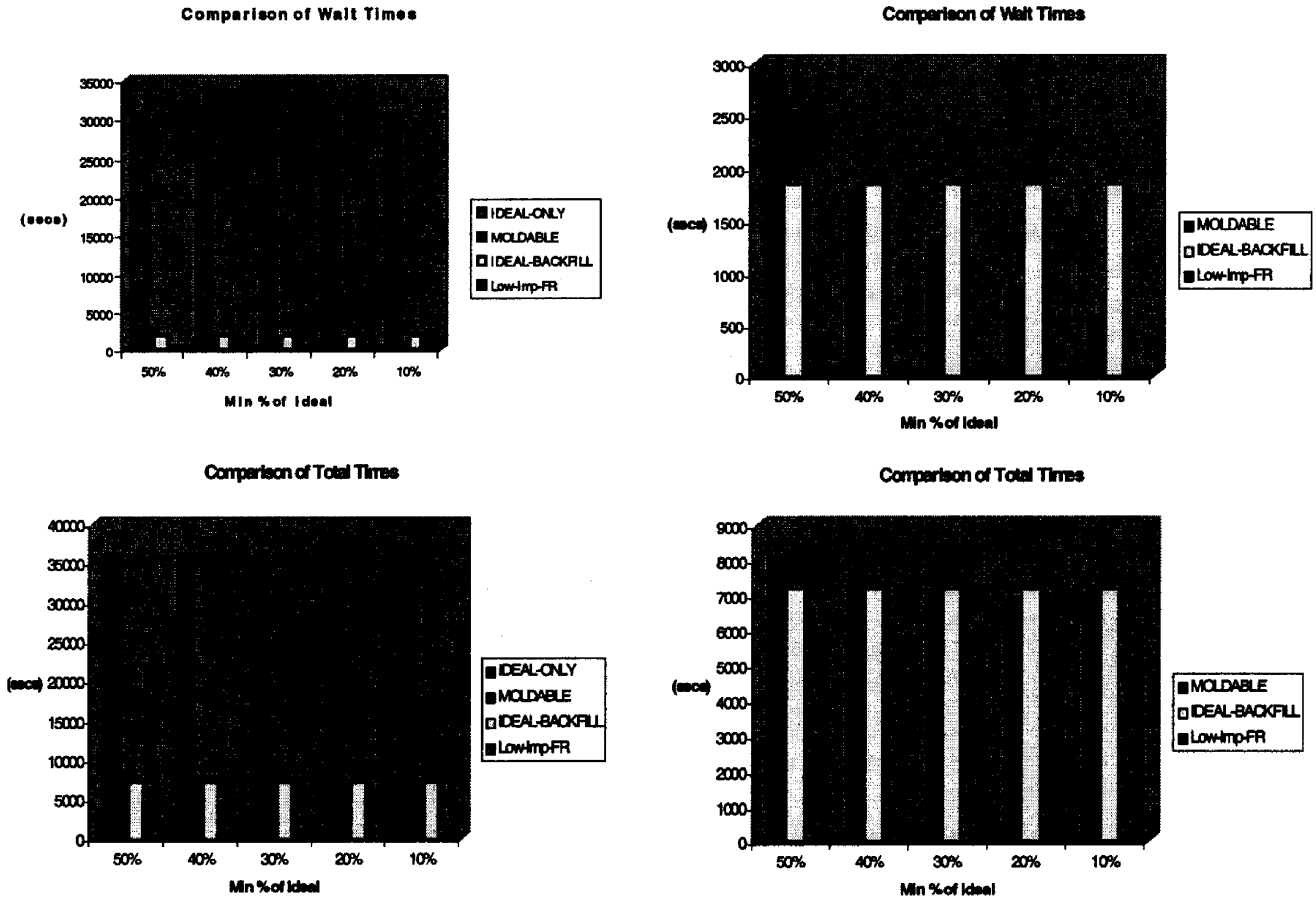


Fig. 14. Final comparison: best vs. baselines.

Table 2
Harvest statistics

Scheduling method	Linear			Parabolic		
	Harvest success rate	# Harvest events/job	% of jobs that perform = or better under harvesting	Harvest success rate	# Harvest events/job	% of jobs that perform = or better under harvesting
Even-H-FQ	96.7	1.6	89.3	98.4	1.5	92.8
Even-H-FR	99.5	3.2	86.4	99.4	3.1	89.0
Low-Imp-FQ	96.7	1.6	89.2	98.3	1.5	92.7
Low-Imp-FR	99.0	3.6	86.7	99.4	3.4	89.2

solely on a single application instance. For example, AppLeS has defined the necessary components to perform application-centric scheduling in distributed environments [3]. In contrast, iSchedulers are designed to support the scheduling of multiple applications. However, iSchedulers could inter-operate with multiple application schedulers (e.g. an AppLeS scheduler or a Prophet scheduler) provided each application scheduler supports its interface. Job-level scheduling approaches make fixed resource allocation decisions for a group of jobs that are treated as "black-boxes" usually with the goal of high throughput, fairness, and high utilization [4,8,11,12,14,19,31]. Job schedulers require that up-front resource requirements be expressed to the system to enable resource allocation when the application is scheduled. However, they do not allow for adaptive scheduling to further optimize scheduling in response to dynamic changes in application resource demands, changes in system state, or subsequent job arrival. iSchedulers can leverage the large-body of job scheduling algorithms, e.g. backfilling, but further improve performance by allowing for more fluid resource management.

The concept of malleable (or adaptive) jobs has been proposed by others in terms of mechanisms [15,26], or theoretical benefits [11]. In this paper, we have presented

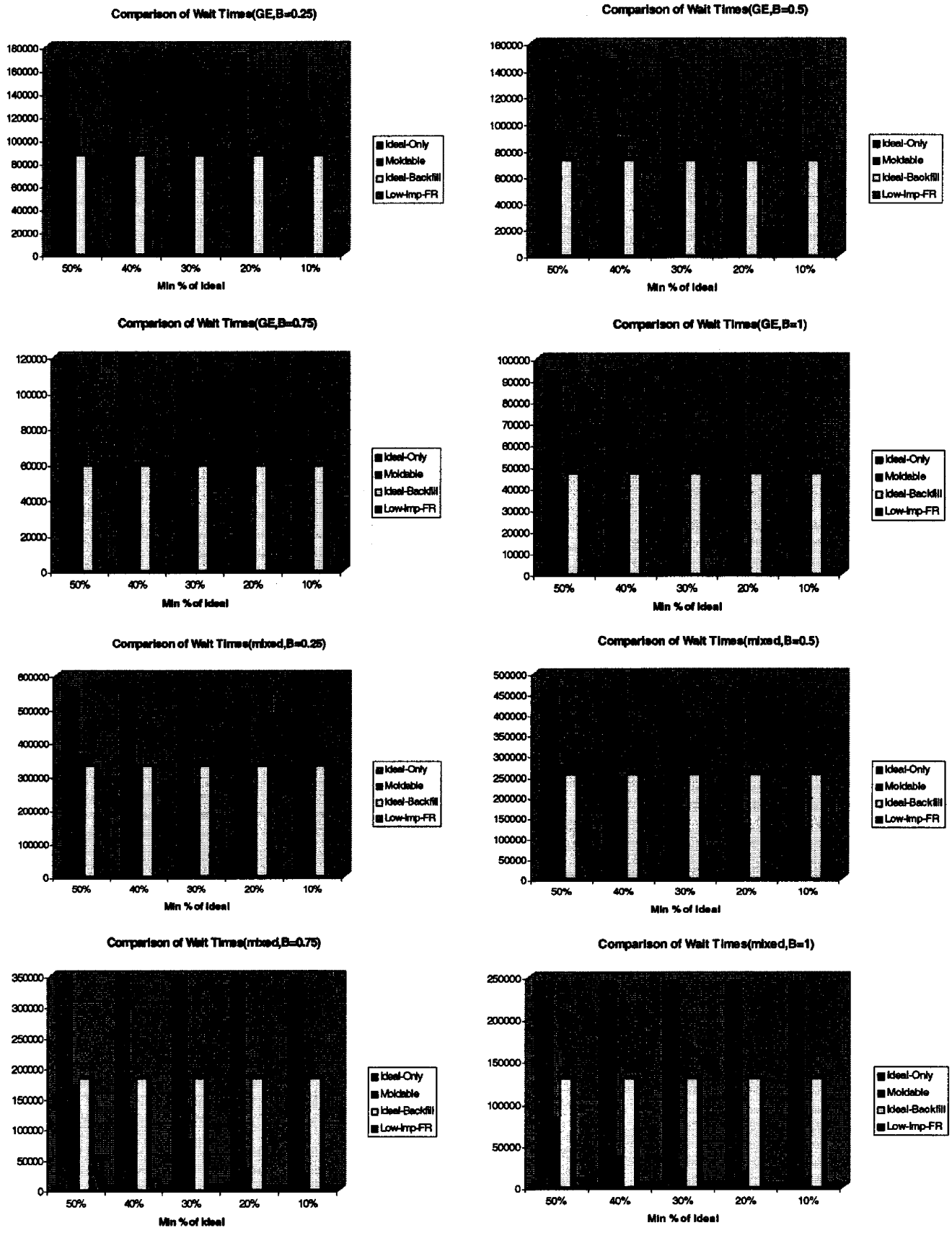


Fig. 15. Wait time comparison—sensitivity to interarrival rate β.

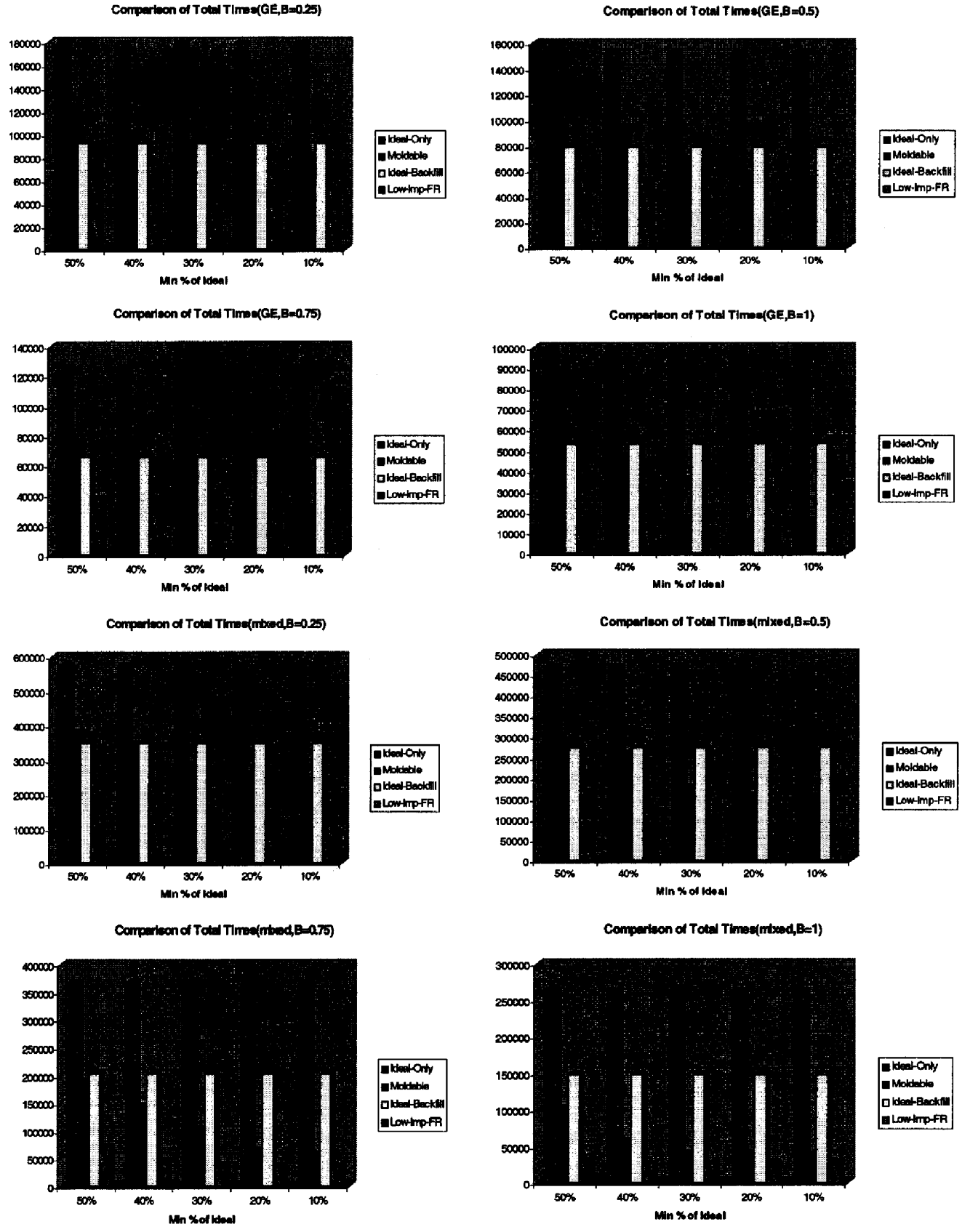


Fig. 16. Total time comparison—sensitivity to interarrival rate β.

Table 3
GE throughput (jobs/h)

β	*min_%_of_ideal*=50%				*min_%_of_ideal*=30%				*min_%_of_ideal*=10%			
	IO	M	IB	LIFR	IO	M	IB	LIFR	IO	M	IB	LIFR
0.25	1.78	2.45	1.79	3.16	1.78	2.95	1.79	3.73	1.78	2.75	1.79	4.04
0.5	1.78	2.35	1.79	3.15	1.78	2.96	1.79	3.71	1.78	2.68	1.79	3.93
0.75	1.77	2.39	1.78	3.12	1.77	2.87	1.78	3.67	1.77	2.63	1.78	3.68
1.0	1.77	2.40	1.78	2.94	1.77	2.85	1.78	3.01	1.77	2.70	1.78	2.99

IO = ideal-only, M = moldable, IB = ideal-backfill, LIFR = low-impact-FR.

policy alternatives and quantified the benefits for such jobs relative to alternative approaches such as backfilling or moldable job scheduling. We have explored performance sensitivity to a range of factors in both supercomputer workloads and in live experiments. Finally, our work should be contrasted to the large body of important static scheduling research including [21,22], and automatic parallelization and scheduling [1,23]. In contrast, our work assumes the job has already been parallelized, and scheduling is performed dynamically at run-time in response to job arrivals and job completion.

6. Summary

In this paper we introduced a new scheduling paradigm, integrated scheduling or iScheduling, that performs application-aware job scheduling. This paradigm allows for much more fluid resource allocation during the life of an application than traditional scheduling approaches. This gives the scheduler much more flexibility in making resource allocation decisions to the benefit of both individual applications and the system as a whole. We showed that several harvest-based scheduling policies can dramatically outperform standard scheduling techniques such as fixed resource allocation (*IDEAL-ONLY*) on both supercomputer workload traces (system-centric iScheduling) and real parallel job traces (user-centric iScheduling). Both waiting time and total finishing time were significantly reduced. These policies also outperformed other optimized policies (*MOLDABLE* and *IDEAL-BACKFILL*) in both waiting and finishing time. These results suggest that iScheduling holds great promise for improving the performance of parallel applications in space-shared systems. However, to reap the benefits of iScheduling in real systems, the applications must be iScheduler-aware and support an adaptation interface for sensing and actuating. Our future work consists of building tools and libraries that facilitate the use of iScheduling for new applications and environments. We are working on adaptive libraries that make it easier to "insert" adaptivity within applications in support of the iScheduling interface and a run-time library that encapsulates harvest-based scheduling decisions. The latter is currently being developed as part of a project to support resource management of high-end network services. Other work includes a more thorough investigation of different harvest-based policies. Another avenue of future work is applying the concept to environments in which not all applications are iScheduler-aware such as Grids [13]. In this environment, the iScheduler will only control a subset of the applications and resources, and have to adapt to resource sharing due to applications outside its control.

Acknowledgments

This work was sponsored in part by the Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative Agreement Number DAAD19-01-2-0014.

References

[1] I. Ahmad, Yu-Kwong Kwok, Min-You Wu, Wei Shu, CASCH: a tool for computer aided scheduling, IEEE Concurrency 8(4) (October–December 2000).
[2] R. Armstrong, et al., Towards a common component architecture for high-performance scientific computing, Proceedings of the Eighth International Symposium on High Performance Distributed Computing, Redondo Beach, CA, August 1999.
[3] F. Berman, R. Wolski, Scheduling from the perspective of the application, Proceedings of the Fifth International Symposium on High Performance Distributed Computing, Syracuse, NY, August 1996.
[4] T.L. Casavant, J.G. Kuhl, A taxonomy of scheduling in general-purpose distributed computing systems, IEEE Trans. Software Eng. 14 (February 1988) 141–154.
[5] H. Cassanova, J. Dongarra, Netsolve: a network server for solving computational science problems, Internat. J. Supercomput. Appl. High Performance Comput. 11 (3) (1997) 212–223.
[6] N. Chrisochoides, Multithreaded model for dynamic load balancing parallel adaptive PDE computations, ICASE TR-95-83, Norfolk, VA, December 1995.

[7] W. Crine, F. Berman, A model for moldable supercomputer jobs, Proceedings of the IPDPS 2001—International Parallel and Distributed Processing Symposium, San Francisco, CA, April 2001.
[8] D.L. Eager, E.D. Lazowska, J. Zahorjan, Adaptive load sharing in homogeneous distributed systems, IEEE Trans. Software Eng. 12 (May 1986).
[9] D. Feitelson, B. Nitzberg, Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860, in: D. Feitelson, L. Rudolph (Eds.), Job Scheduling Strategies for Parallel Processors, Lecture Notes in Computer Science, Vol. 949, Springer, Berlin, 1995.
[10] D. Feitelson, Parallel Workload Archive, http://www.cs.huji.ac.il/labs/parallel/workload/logs.htm, 1999.
[11] D. Feitelson, L. Rudolph, Parallel job scheduling: issues and approaches, Job Scheduling Strategies for Parallel Processing Workshop, Santa Barbara, CA, 1995.
[12] D. Feitelson, A. Weil, Utilization and predictability in scheduling the IBM SP2 with backfilling, Proceedings of the 12th International Parallel Processing Symposium, Orlando, FL, April 1998.
[13] I. Foster, C. Kesselman (Eds.), High-performance schedulers, in: The Grid Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers, Los Altos, CA, 1998 (Chapter 12).
[14] H. Franke, P. Pattnaik, L. Rudolph, Gang scheduling for highly efficient multiprocessors, Sixth Symposium on the Frontiers of Massively Parallel Computation, Annapolis, MD, 1996.
[15] D. Gelernter, et al., Adaptive parallelism and piranha, IEEE Comput. 28 (1) (1995) 40–49.
[16] J.K. Hollingsworth, P.J. Keleher, Prediction and adaptation in active harmony, Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing, Chicago, IL, July 1998.
[17] E.N. Houstis, J.R. Rice, E. Gallopoulos, R. Bramley (Eds.), Enabling Technologies for Computational Science: Frameworks, Middleware, and Environments, Kluwer Academic Publishers, Dordrecht, 2000.
[18] S. Hovoty, Workload evolution on the Cornell theory center IBM SP2, in: D. Feitelson, L. Rudoplh (Eds.), Job Scheduling Strategies for Parallel Processors, Lecture Notes in Computer Science, Vol. 1162, Springer, Berlin, 1996.
[19] W. Leinenberger, G. Karypis, V. Kumar, Job scheduling in the presence of multiple resource requirements, Proceedings of SC'99, Portland, OR, November 1999.
[20] N.H. Kapadia, R.J. Figueiredo, J.B. Fortes, PUNCH: web portal for running tools, IEEE Micro 10 (May–June 2000) 38–47.
[21] Y. Kwong Kwok, I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Comput. Surv. 31(4) (December 1999) 406–471.
[22] Y. Kwong Kwok, I. Ahmad, Benchmarking and comparison of the task graph scheduling algorithms, J. Parallel Distributed Comput. 59(3) (December 1999) 381–422.
[23] V.M. Lo, et al., OREGAMI: tools for mapping parallel computations to parallel architectures, Internat. J. Parallel Programming 20(3) (June 1991) 237–270.
[24] H. Nakada, M. Sato, S. Sekiguchi, Design and implementation of Ninf: towards a global computing infrastructure, J. Future Generation Comput. Systems, Metacomput. Issue 10 (1999) 649–658.
[25] T. Nguygen, R. Vaswani, J. Zahorjan, Maximizing speedup through self-tuning of processor allocation, Proceedings of the 10th International Parallel Processing Symposium, Honolulu, HI, April 1996.
[26] S. Setia, et al., Supporting dynamic space-sharing on clusters of non-dedicated workstations, Proceedings of the 17th International Conference on Distributed Computer Systems, Baltimore, MD, 1997.
[27] W. Smith, I. Foster, V. Taylor, Predicting application run times using historical information, in: D. Feitelson, L. Rudolph (Eds.), Job Scheduling Strategies for Parallel Processors, Lecture Notes in Computer Science, Vol. 1459, Springer, Berlin, 1998.
[28] J.B. Weissman, Prophet: automated scheduling of SPMD programs in workstation networks, Concurrency: Practice Experience 11(6) (May 1999) 301–321.
[29] J.B. Weissman, Gallop: the benefits of wide-area computing for parallel processing, J. Parallel Distributed Comput. 54(2) (November 1998) 183–205.
[30] J.B. Weissman, Predicting and cost and benefit of adapting data parallel applications in clusters, J. Parallel Distributed Comput. 62(8) (August 2002).
[31] Y. Zhang, et al., Improving parallel job scheduling by combining gang scheduling and backfilling techniques, Proceedings of the 14th International Parallel & Distributed Processing Symposium, Cancun, Mexico, May 2000.

Lakshman Rao Abburi is currently a graduate student in the Department of Computer Science, University of Minnesota–Twin Cities. He received the bachelor's degree in computer science in 2001 from the University of Madras, India. His research interests include distributed systems, operating systems, computer networks and parallel processing.

Darin England received the Bachelor of Science in industrial engineering from Purdue University and the Master of Science in operations research from the Georgia institute of Technology. He is currently working on his Ph.D. in computer science at the University of Minnesota. Darin's research interests are parallel and distributed computing, heuristic search methods, and mathematical optimization.

Jon B. Weissman received the B.S. degree from Carnegie-Mellon University in 1984, and the M.S. and Ph.D. from the University of Virginia in 1989 and 1995 respectively, all in Computer Science. He has been an assistant Professor of computer science at the University of Minnesota since 1999. He was an active member of the Mentat and Legion research groups while at the University of Virginia. His current research interests are in resource management in parallel and distributed systems and Grid computing. He is the architect of two software systems, Prophet and Gallop, that provide automated scheduling support in Grid systems. His research is supported by NSF and the AHPCRC.