Cluster Scheduling for Explicitly-Speculative Tasks

David Petrou
Electrical & Computer Eng.
Carnegie Mellon University
dpetrou@ece.cmu.edu

Gregory R. Ganger
Electrical & Computer Eng.
Carnegie Mellon University
ganger@ece.cmu.edu

Garth A. Gibson
Comp. Sci. and ECE
Carnegie Mellon University
garth@cs.cmu.edu

ABSTRACT

Large-scale computing often consists of many speculative tasks to test hypotheses, search for insights, and review potentially finished products. E.g., speculative tasks are issued by bioinformaticists comparing DNA sequences and computer graphics artists adjusting scene properties. We promote a way of working that exploits the inherent speculation in application-level search made more common by the cost-effectiveness of grid and cluster computing. Researchers and end-users disclose sets of speculative tasks that search an application space, request specific results as needed, and cancel unfinished tasks if early results suggest no need to continue. Doing so matches natural usage patterns, making users more effective, and also enables a new class of schedulers.

In simulation, we show how *batchactive schedulers* reduce user-observed response times relative to conventional models in which tasks are requested one at a time (interactively) or in batches without specifying which tasks are speculative. Over a range of situations, user-observed response time is about 50% better on average and at least two times better for 20% of our simulations. Moreover, we show how user costs can be reduced under an incentive cost model of charging only for tasks whose results are requested.

Categories and Subject Descriptors: D.4.1 [Operating Systems]: Process Management
General Terms: Algorithms, Design, Performance
Keywords: speculative, optimistic, cluster, grid scheduling

1. INTRODUCTION

Large-scale computing often consists of many speculative tasks to test hypotheses, search for insights, and review potentially finished products. This work addresses how to reduce or eliminate user-observed response time by prioritizing work that the user is waiting on and wasting fewer resources on speculative tasks that might be canceled. This *visible response time* is the time that a user actually waits for a result, which is often less than the time that a speculative task has been in the system. Our target architecture is a computational grid [5] or shared cluster [3, 25]. Our deployment plan is to replace or augment the extensible scheduling policies in clustering software such as Condor, Beowulf, Platform LSF, Globus, and the Sun ONE Grid Engine.

Speculative tasks are pervasive and intelligently scheduling them is increasingly important in supercomputing and grid architectures. A speculative task is one not yet known to be required [12]. A *task set* is a collection of such tasks. The user may want one, all, or some of these tasks. After requesting a needed task, receiving the output, and considering this output, the user decides whether to request the next task in the set. This approach is commonly known as a *speculative search, speculative test,* or *parameter study.*

Imagine a scientist using shared resources to validate a hypothesis. She issues a task set that could keep the system busy for hours or longer. Tasks listed earlier are to answer pressing questions while those later are more speculative. Early results could cause the scientist to reformulate her line of inquiry; she would then reprioritize tasks, cancel later tasks, issue new tasks. Moreover, the scientist is not always waiting for tasks to complete; she spends minutes to hours studying the output of completed tasks.

We promote a model that exploits the inherent speculation in such scenarios. Users exploring search spaces *disclose* sets of speculative tasks, *request* results as needed, and *cancel* unfinished tasks if early results suggest no need to continue. We call this the *batchactive model* in contrast to users that interactively submit needed tasks one at a time and users that submit batches of both needed and speculative tasks without identifying which tasks are which.

In the batchactive model are *batchactive schedulers* that attempt to maximize human productivity and minimize user resource costs by scheduling speculative tasks differently. Our heuristic is to segregate tasks into two queues based on whether a task is speculative and give the non-speculative tasks priority, resulting in better response time at lower resources usage. This organization is shown in Figure 1.

Our approach applies best to domains in which several to a potentially unbounded number of intermediate speculative tasks are submitted and early results are acted on while unfinished tasks remain. Considerable performance improvements are found even when the average depth across users

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'04, June 26–July 1, 2004, Malo, France.
Copyright 2004 ACM 1-58113-839-3/04/0006 ...$5.00

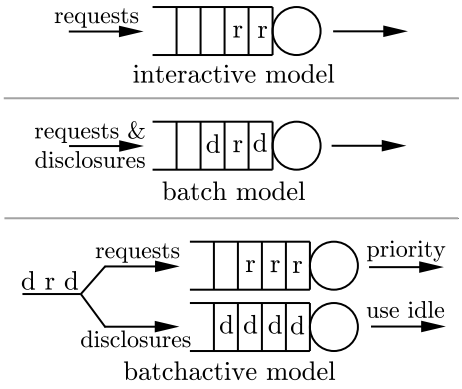

Figure 1: Models of user and scheduler behavior. Users either request needed tasks one at a time (interactive), or request needed and disclose speculative tasks together (batch and batchactive). Our batchactive policies place requested and disclosed tasks in different queues, in which the requested queue has priority.

of disclosed speculative tasks is 3 or 4. The following are several highly-applicable domains:

Bioinformatics Bioinformaticists explore biological hypotheses, searching among DNA fragments with similarity tools like BLAST [2] and FASTA. These scientists share workstation farms — such as the dedicated 30 machines at the Berkeley Phylogenomics Group [17] — and issue series of fast, inaccurate searches (taking from 10 seconds to 10 minutes when matching against human genome sequences) followed by slow, accurate searches to confirm initial findings. A batchactive scheduler would enable scientists to explore ambitious ideas potentially requiring unbounded resources without fear that resources would be wasted on tasks that might be canceled after early results were scrutinized. Some scientists wish to submit 10,000 sequencing tasks because they 'really do not know what [. . .] sequences will work.' [14]

Computer graphics Teams of hundreds of artists creating a computer-animated film, such as at Dreamworks or Pixar, submit shots for rendering, where each shot has roughly 200 frames, to a cluster of hundreds to thousands of processors. Each frame, which consists of independent tasks (for lighting, shading, animation, etc.) can take from minutes to hours to render.[1] The artist submits the entire shot at once. This work is highly speculative: the overwhelming majority of computation never makes it into the final film. [13, 21] The artist requests key frames, those with more action, for example, to decide on the finished product. With a batchactive scheduler, the key frames would run before speculative, remaining frames. If upon seeing these frames the artist changes lighting attributes or moves objects, speculative frames would be canceled and would not have unnecessarily competed against the key frames of other team artists.

[1]Over a nine month production period, Weta digital employed 3,200 processors to create Lord of the Rings: The Return of the King. In this film there were 1,400 special effects shots, each containing at least 240 frames, and the average frame took 2 hours to render. [16]

Simulation Computer scientists routinely use clusters to run simulations exploring high dimensional spaces. Exploratory searches, or parameter studies, for feature extraction, search, or function optimization, can continue indefinitely, homing in on areas for accuracy or randomly sampling points for coverage.[2] In our department, clusters are used for, among other things, trace-driven simulation for studying microarchitecture, computer virus propagation, and storage patterns related to I/O caching and file access relationships. With a batchactive scheduler, such simulations could occur in parallel with the experimenter analyzing completed results and guiding the search in new directions, with the speculative work operating in the background when pressing tasks are requested.

The batchactive model requires rethinking metrics and algorithms. In particular, when the system is aware of speculative tasks, the traditional response time metric should be refined. We introduce *visible response time,* the time between requesting and receiving task output, or the time 'blocked on' output. Visible response time accrues only after a user requests a task that may or may not have been already disclosed. In contrast, and less usefully, response time accrues as soon as a task enters the system.

The batchactive model also suggests refining the cost model. Traditionally, computing centers charge for resource usage irrespective of whether a task was needed [20]. We introduce an incentive cost model which charges for only resources used by tasks whose results are requested. This encourages users to disclose speculative work deeply, and prevents the type of gaming in which users mark all tasks as high-priority, requested tasks. We show that user costs are actually reduced under this cost model.

We show in simulation that discriminating between speculative and conventional tasks improves on time and resource metrics compared to traditional schedulers. Over a broad range of simulated user behavior, we show that visible response time is improved by about 50% on average under a batchactive scheduler, and is at least two times better for 20% of our simulations. Beyond these benefits, the batchactive computing model creates a new challenge for scheduling theory and practice: how does one best schedule speculative, abortable tasks to minimize visible response time?

2. USER AND TASK MODEL

We model a closed loop of a constant *number of users* interacting with the system. Users *disclose* speculative work as *task sets,* which are ordered lists of tasks. When a user needs a result from a task set, the user *requests* the task. After some *think time,* the user may request the next task or *cancel* all unrequested tasks and issue a new task set. The act of cancelation models a user that doesn't need any more results from the task set.

Figure 2 depicts this user interaction with the scheduler, and the scheduler's interaction with the computing system.

[2]The Xfeed agent with the Xgrid clustering software (from Apple Computer, Inc.) explicitly supports this model. One specifies a range of arguments (or a random sampling) to pass to a command. Xfeed generates task specifications for each possible combination of arguments and submits them.

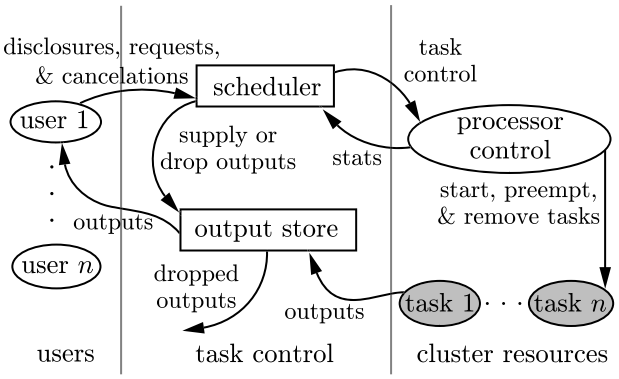

Figure 2: Interaction between users, the scheduler, and the computing center's resources.

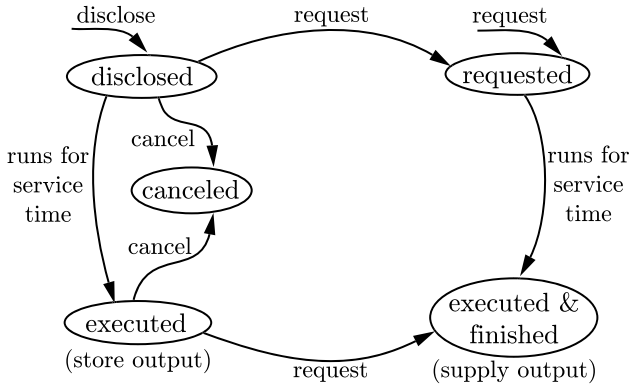

Figure 3: Task state transitions.

Any number of users disclose, request, and cancel any number of tasks. The scheduling policy decides which and when disclosed and requested tasks run. If a disclosed task is canceled, it is no longer a candidate. The scheduler communicates decisions to the conventional clustering software which handles the details of running tasks on the servers.

Figure 3 details the states that a task can be in. Each task has a *resource usage* that increases as the task runs. When a task's resource usage equals its *service time,* the task is considered *executed.* If a task is both executed and requested, then the task is considered *finished* and the task's output is supplied to the requesting user. If a task executes and was disclosed but not requested, then the task's output is stored until requested or canceled.

The probability that a given task's results will cause a user to cancel and issue a new task set is the *task set change probability.* This parameter is modeled by a uniform random variable whose lower bound is always 0 and whose upper bound varies across runs. Each user is assigned a change probability from this distribution for all the user's tasks, so that we simulate users who are more or less certain about whether they will request their disclosed, speculative work.

The number of *tasks per task set* is another simulation parameter and is also drawn from a uniform random variable for each user. Here, the lower bound is always 1, meaning no disclosure. A low upper bound reflects shallow disclosure; a scientist planning up to five or so experiments ahead. A high upper bound, deep disclosure in the thousands, reflects task sets submitted by an automated process for users searching high-dimensional spaces.

Service time and think time for all tasks, irrespective of user, drawn from the exponential distribution.

3. BATCHACTIVE SCHEDULING

People wish to batch their planning and submission of tasks and pipeline the analysis of finished tasks with the execution of remaining tasks. Conventional interactive and batch models are obstacles to this way of working.

In the interactive model, users request one task at a time. In contrast, tasks can execute while task output is consumed in the batchactive model. Endowing servers with knowledge of future work in the form of disclosed tasks gives the servers an early start, rather than being idle. In the batch model, the scheduler does not discriminate between requests and disclosures. In contrast, distinguishing between disclosed and requested tasks in the batchactive model enables the user to disclose deeply, knowing that the scheduler will give precedence to users waiting on requested tasks. In contrast to both common practices, users will observe lower visible response times, making them more productive and less frustrated. The profound effect on scheduling metrics exhibited by batchactive scheduling is detailed by the simulation results in the evaluation section.

Our batchactive cost model charges for only resources used by tasks whose results are requested. This encourages users to disclose speculative work deeply, which enables the scheduler to best reduce visible response time. Note that if a user attempts to gain scheduling priority by requesting instead of disclosing speculative tasks, then the user will be charged for those speculative resources.[3]

3.1 Metrics

Mean *visible response time* is our main metric. A task with service time S is requested at time t_r and is executed (completes) at time t_e. Each requested and executed task has a corresponding visible response time denoted by

$$V_{\text{resp}} \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } t_r > t_e, \\ t_e - t_r & \text{if } t_r \leq t_e. \end{cases}$$

Recent work [15] argues that mean slowdown, which expresses the notion that users are willing to wait longer for larger work, should be minimized. Thus, we also study *visible slowdown* denoted by

$$V_{\text{slow}} \stackrel{\text{def}}{=} \frac{V_{\text{resp}}}{S}.$$

[3]Moreover, a user who discloses a large task set but never requests tasks will lessen the benefit of batchactive scheduling for others because these unrequested tasks would compete with speculative tasks that will probably be requested. This might be unintentional or a denial-of-service. One solution would be a disclosed task scheduler that favors users who have historically been better speculators.

metric	description
visible response time	blocked time
visible slowdown	blocked time over service time
task throughput	number of finished tasks
scaled billed resources	billed over requested resources
load	fraction of server busy time

Table 1: Metrics reported among schedulers.

Note the differences between these two metrics and the traditional response time and slowdown metrics. Because disclosed tasks can run before they are requested, visible response time can be less than service time and visible slowdown can be less than one.

We also measure task throughput: the number of finished tasks (tasks that were requested and executed) over the duration of our simulation. One of our results is that we improve visible response time without hurting task throughput, and in many cases we improve both.

We introduce a metric reflecting the billed resources wasted on disclosed tasks that were never requested. In the interactive and batchactive models, only requested tasks are billed to the user. However, for the batch model, which does not discriminate between requested and disclosed tasks, this waste is significant. *Scaled billed resources* is the ratio of the billed resources to the requested resources. For example, if a scheduler charged for ten seconds of resource time but the user only requested five seconds of resource time, then the scaled billed resources is 2.

For any server, its *load* (also known as device utilization) is the fraction of time that the server was running a task.

Finally, we report an *improvement* factor between a new, batchactive scheduler and some baseline. For example, if a metric is better when lower and if the baseline scheduler simulation gave 50 as the metric and the new scheduler simulation gave 25, then the improvement is 2.

We summarize these metrics in Table 1.

3.2 Scope

We make some simplifying assumptions: A task uses only one resource significantly. For example, a processor-intensive task might access the disk to load a dataset, but those accesses must be an insignificant part of the task's work. Moreover, we assume that preemption costs are low. Our policies have not been tailored to tasks with out-of-core memory requirements, for example, although we believe adapting known techniques would be straightforward. Further, we assume no complex interactions among tasks, like contention for shared locks. Finally, we assume that there is sufficient storage to hold speculative output from disclosed tasks that have not yet been requested or canceled.

3.3 Policies

Any scheduler that behaves differently based on whether a task is requested or disclosed is a batchactive scheduler. Thus, there are many possible batchactive schedulers, such as using any combination of traditional policies separately for requested and disclosed tasks. We notate batchactive schedulers by *requested task policy* × *disclosed task policy* (for example, SRPT × FCFS).

We wish to minimize mean visible response time across all tasks. Since requested work is more important than speculative work, our heuristic gives requested tasks priority. That is, if an idle processor and a pending requested task exist, this task runs before any pending disclosed tasks. Most of the time, this is the right choice.

SRPT (shortest-remaining-processing-time), which optimally minimizes response time in non-speculative contexts [10, ch. 8], is used for requested tasks for most of our batchactive results. We also show that non-size-based policies (FCFS) provide nearly identical improvements. Thus, knowing service time is *not* required for batchactive scheduling.[4]

The disclosed queue from the batchactive scheduler are run according to FCFS (first-come-first-serve). The motivation behind FCFS for this queue is to quickly run tasks that will be requested first. Recall that task sets are ordered lists of disclosed, speculative tasks. Since our simulated users request tasks in this order, then applying FCFS to the task set order is a perfect estimate of request order within one user, and a reasonable estimate across all users.

4. EVALUATION

We demonstrate that the batchactive user model of disclosing speculative tasks combined with batchactive schedulers that segregate requested and disclosed tasks into two queues provides better visible response time, visible slowdown, task throughput, and scaled billed resources.

Evaluation is a simulation fed the synthetic tasks and user behaviors described in Section 2. Time, such as service time, is stated in seconds. We simulate rather than analytically model due to the complexities of our user model which includes probabilistic task cancelation motivated by the pervasive and important scenarios of Section 1.

Our simulations explore a range of user and task behavior. Each run simulates two weeks of time after two warmup days (see Section 4.3) were ignored.

The choice of simulation parameters is key to arguing for the batchactive computing model. We can make batchactive computing look arbitrarily better than common practice by selecting parameters most appropriate to the batchactive scheduler. However, this would not be a convincing argument. Instead, we have chosen parameter ranges that not only include what we believe to be reasonable uses of speculation for our target applications, but also ranges that include little or no speculation. We show that under no speculation, we do no worse than conventional models.

[4]Using service time predictions provides better absolute performance for batchactive and non-batchactive schedulers. Spring, et al. show that the service time of the `complib` biological sequencing library is highly predictable from input size [27]. In computer rendering, once a shot is being processed, the runtimes of individual tasks are 'generally predictable' [13]. Kapadia et al. use regression to predict the resource requirements of grid applications [18].

parameter	range
number of users	4–16
task set change prob.	0.0 to 0.0–0.4 (uni.)
tasks in a task set	1 to 1–19 (uni.)
service time (s)	20–3,620 (exp.)
think time (s)	20–18,020 (exp.)

Table 2: The parameter ranges used in simulating users. The parameters were defined in Section 2. Parentheses indicate random variables (uniform and exponential). Uniform distributions are notated by 'lower bound to upper bound,' where the upper bound is a range.

4.1 Summarizing results

We used the parameter ranges listed in Table 2 for the results in this section unless otherwise noted. For each parameter, we sampled several points in its range. All told, each scheduler was evaluated against 3,168 selections of parameters.

We simulate one server, and restrict to 16 the maximum number of users concurrently issuing task sets to the server. The probability after receiving a requested task's output and thinking about it that the user cancels the rest of the task set ranges from those who always need their speculative work to those who cancel task sets 40% of the time. Tasks per task sets range from no disclosure to about twenty disclosures, reflecting those using domain-specific knowledge to plan small to medium-sized task sets.

Service times, which vary from one third of a minute to about one hour, are modeled after BLAST DNA similarity searches [14] and frame rendering [16, 13, 21]. Think times, which vary from a third of a minute to roughly five hours, reflect a user who can make a quick decision about a task's output to one who needs to graph, ponder, or discuss results with colleagues before deciding to request the next task or cancel and start a new task set.

The following figures are cumulative improvement graphs that show the fraction of runs in which the performance of the batchactive model was at least a certain factor better than the non-batchactive schedulers. The x-axes are factor improvements and the y-axes are the metrics. For example, in Figure 4, the solid line intersection with the x-axis at 3 says that in 10% of all simulation parameters, the ratio of the mean visible response time for interactive SRPT to the mean visible response time for SRPT × FCFS was at least 3.

Figure 4 shows how SRPT × FCFS compares to interactive SRPT and batch SRPT for visible response time. SRPT × FCFS and interactive SRPT perform the same for about 65% of the runs, while SRPT × FCFS does better than batch SRPT for many more situations. However, there are more situations where the batchactive scheduler is between two and four times better than the interactive scheduler. Against both existing models, batchactive performs at least two times better for about 20% of the runs. The mean improvement is over 50%.

It is important to know whether these results hold using non-size-based schedulers since size cannot always be obtained

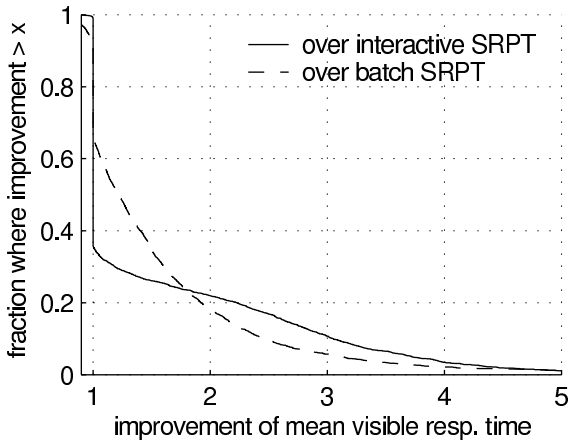

Figure 4: Cumulative factor improvement of SRPT × FCFS over interactive and batch SRPT for visible response time. Against both schedulers, batchactive performs at least twice as better for about 20% of the simulated behaviors. The mean improvement is 1.525 over interactive and 1.537 over batch.

or predicted. We find in Figure 5 that the improvements of batchactive scheduling over interactive and batch using FCFS scheduling are quite similar to those in Figure 4.

We now compare batchactive with batch FCFS[5] in Figure 6 for scaled billed resources (the ratio of the billed resources to the requested resources) and find that the user cost for resources is considerably lower under the batchactive model. Recall that in the batchactive model, disclosed tasks, which run when the server would otherwise be idle, are free unless they eventually are requested. But because the batch model does not discriminate between requested and disclosed tasks, it charges for them all. (The interactive model, not shown, charges the same as the batchactive model because only requested tasks are run.) In sum, batchactive charges at least four times less for about 40% of the runs.

To the resource provider's benefit, the batchactive model can provide more total billed resources over the same time period compared to the interactive model. Both models charge only for requested resources. But since the batchactive model provides better task throughput, there can be more requested tasks completed in the same time.

For a small percentage of the runs, batchactive did slightly worse. We believe this is not an actual performance drop, but error introduced due to variations in the number of finished tasks: different schedulers complete different numbers of tasks during the same virtual time. Thus, metrics among two schedulers running with the same parameters are tallied from a different number of finished tasks. The confidence intervals for a small subset of runs shown in Section 4.3 suggest that for the cases in which batchactive's mean is worse, the 95% confidence intervals between batchactive and baseline schedulers overlap.

[5]We compared against batch FCFS instead of batch SRPT because it provided better competition against batchactive SRPT × FCFS for scaled billed resources.

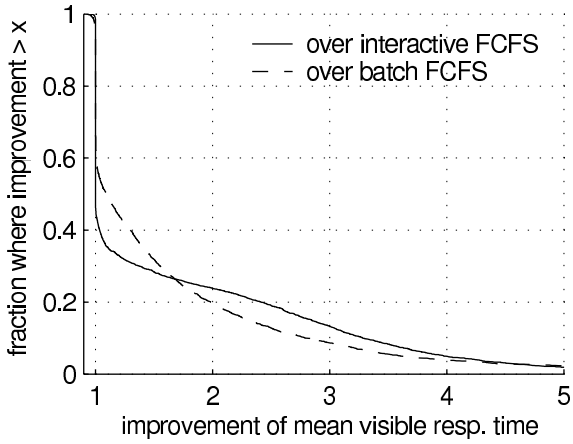

Figure 5: Cumulative factor improvement of FCFS × FCFS over interactive and batch FCFS for visible response time. Against both schedulers, batchactive performs at least twice as better for about 20% of the simulated behaviors. The mean improvement is 1.615 over interactive and 1.595 over batch.

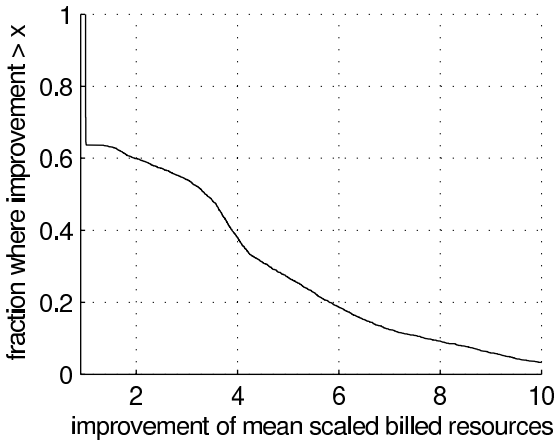

Figure 6: Cumulative factor improvement of SRPT × FCFS over batch FCFS for scaled billed resources. As only requested tasks are charged, batchactive performs much better than batch. Batchactive charges at least four times less for about 40% of the runs. The mean improvement is 3.647.

parameter	setting
number of users	8
task set change prob.	0.0 to 0.2 (uni.)
tasks in a task set	1 to 15 (uni.)
service time (s)	600 (exp.)
think time (s)	6000 (exp.)

Table 3: The fixed parameters used to compare schedulers. For each run, all but one were held constant at these values. Parentheses indicate random variables. Uniform distributions are notated by 'lower to upper bound.'

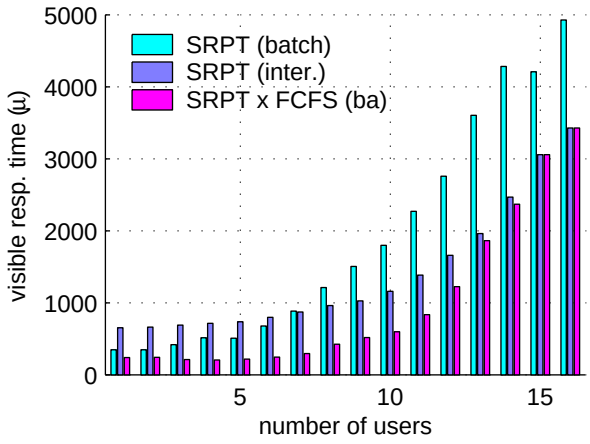

Figure 7: The effect of the number of users on visible response time. With few users, interactive is better than batch; with many, batch is better than interactive. Almost always, batchactive adapts and wins.

4.2 Per-parameter investigations

Now we look closely at what affects performance. We report metrics for different scheduling models while considering slices of the user and task behavior parameter space.

The following graphs have the same format. Each set of three bars corresponds to one selection of parameters. The left bar is the batch model, the middle bar is the interactive model, and the right bar is the batchactive model. Each graph varies one parameter. The fixed parameters are taken from those listed in Table 3 unless otherwise noted.

Figure 7 shows how the number of users affects visible response time. Batch is suited to few users because execution time and think time are pipelined and the load is sufficiently low that one's disclosed but never requested tasks don't overly interfere with other requested tasks. Interactive is suited to many users because the server is always busy with requested tasks.

Nearly always, batchactive performs best, exhibiting adaptability to load. Batchactive is better than batch under many users because requested tasks never wait for disclosed tasks; it is better than interactive under few users because it consumes idle time with disclosed tasks. At the busiest part, interactive and batchactive are equivalent because the requested task queue is never empty.

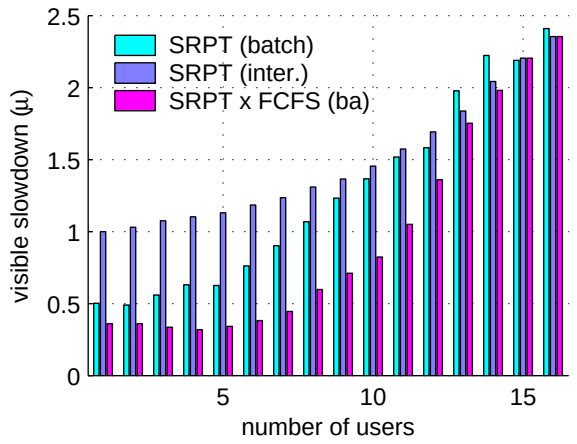

Figure 8: The effect of the number of users on visible slowdown. In nearly all cases, batchactive wins. (Note: visible slowdown can be under one.)

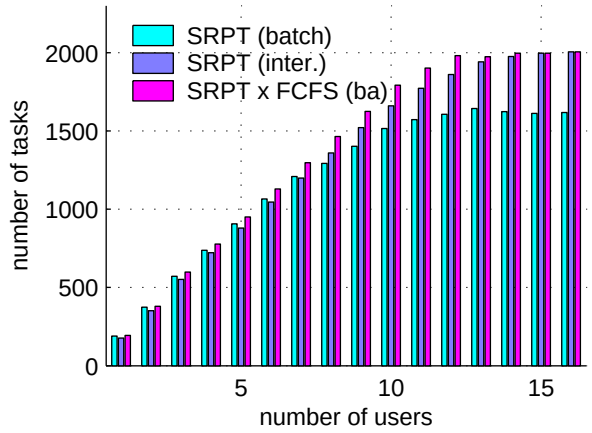

Figure 9: The effect of the number of users on task throughput (number of times requested output was delivered to the user) over the simulation run. Batchactive provides the best task throughput.

Figure 8 repeats the runs of Figure 7 but plots visible slowdown instead. Batchactive still outperforms the alternatives. In contrast to the visible response time results, with many users the visible slowdowns of batch and interactive converge. This occurs because the visible response times of the large tasks affect visible slowdown less.

Figure 9 shows how the number of users affects task throughput (how often a user received requested task output) over two simulated weeks. Batchactive finishes more tasks while providing the better visible response times and slowdowns in Figures 7 and 8. Only toward the highest number of users does interactive throughput converge to batchactive.

Additional insight is found by comparing Figures 7, 8, and 9 with Figure 10, which is the same run except that load is plotted. The load under the batch and batchactive models are similar. As their loads increase, batchactive provides better visible response time than batch because batchactive favors requested tasks. Batchactive provides better visible response time and slowdown than interactive because batchactive pipelines disclosed tasks with think time. Counterintuitively, when batchactive's load is higher than interactive, it still performs better than interactive. What matters isn't merely load, but the fraction of load made up of tasks that have been or will be requested. Only when the batchactive and interactive loads approach one (near 14 users), do their visible response times and slowdowns converge.

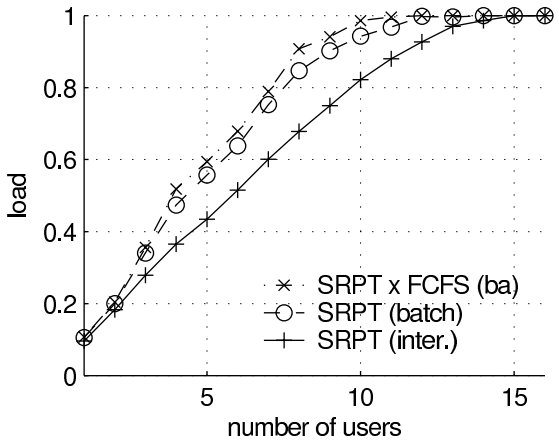

Figure 10: The effect of the number of users on load (utilization). The price batchactive schedulers pay for significantly improving visible response time relative to interactive schedulers is increased load. In comparison to batch schedulers, however, batchactive induces little extra load while delivering significantly better visible response time. (See Figure 7.) This illustrates that conventional load does not fully convey batchactive behavior: what matters is the fraction of load made up of tasks that have been or will be requested.

In Figure 11, we vary the upper bound of the task set change probability from 0.0–0.4 and plot visible response time. This parameter does not affect the interactive model, which does not submit disclosed tasks. We notice a greater dependence on this parameter with the batch model compared to the batchactive model because batchactive avoids speculative tasks when requested work remains.

Now we show large task sets reflecting users searching high-dimensional spaces. We varied the upper bound of the tasks per task set uniform distribution from 1–1024 in multiples of 2. We also set the task set change probability to 0.1 so that task sets are not canceled as often.

Visible slowdown results are in Figure 12. When all task sets have only one task, then all models provide the same visible slowdown. Task sets as small as several tasks — easily realizable by users performing exploratory searches — provide good improvement over interactive. As the task set size increases, both batch and batchactive provide visible slowdowns less than one. This occurs because there is now user think time that can be leveraged to run disclosed tasks. Soon batch performance becomes unusable as its single queue is

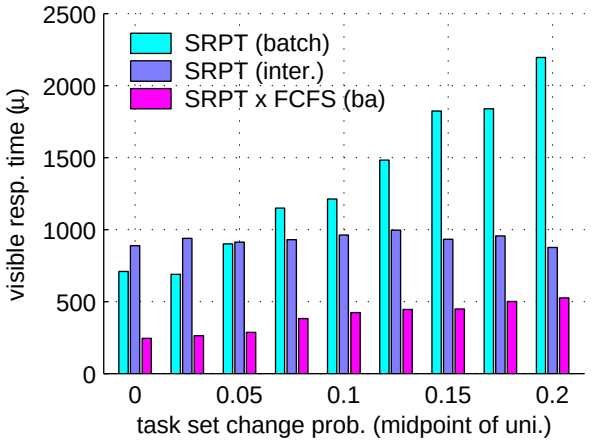

Figure 11: The effect of the task set change probability on visible response time. The probability is a uniform random variable and shown on the x-axis is the average of its lower and upper bounds. The interactive model is unaffected, while the batchactive model is affected less than the batch model and outperforms both.

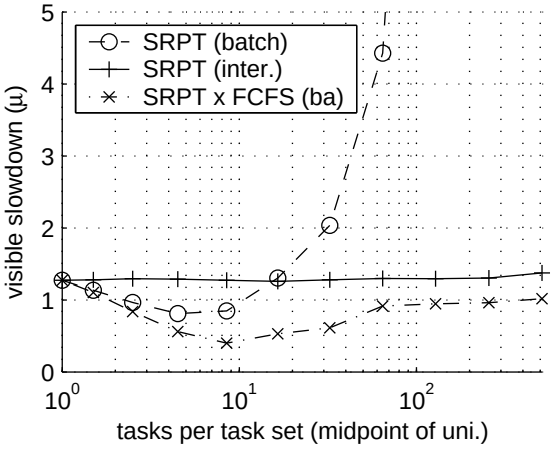

Figure 12: The effect of the tasks per task set on visible slowdown. (This graph is log-linear.) The tasks per task set is a uniform random variable and shown on the x-axis is the average of its lower and upper bounds. Batchactive always wins. Batch is unusable at a task set size with an upper bound of 100. (At 1024, its visible slowdown is over 170.)

overwhelmed with speculative tasks. The interactive model is immune to the task set size because each user will have at most one task (a requested task) in the system. The batchactive model is always best compared to the other models. Compared to itself, its performance gets worse as the task set size increases because there is more competition among speculative tasks, some of which run but are never needed.

Figure 13 shows how service time affects visible response time. As service time increases, the load increases and the performance of interactive and batchactive converge. As a limiting case, when a server is always running requested work, the batchactive model does not help performance.

Think time, the remaining parameter, works inversely to service time: the more think time, the more opportunity for the batchactive scheduler to reduce visible response time. Once the ratio of think time to service time is sufficiently high, which results in a low load, batchactive performs no better than batch while continuing to outperform interactive. We omit results to save space.

4.3 Simulation details

Warmup period The queue length near the start of the simulation is not representative of the behavior of the system over steady-state. At start, all users begin submitting tasks and only after task results are received do users enter their think times. We avoid including warmup time in our reported metrics by conservatively dropping all data in the first two simulated days. Since each run simulates 16 days, our metrics reflect two weeks at steady-state.

Confidence intervals Our parameter studies show that the parameters have a significant effect on scheduling metrics and that batchactive scheduling performs best nearly all of the time (Section 4.2). To reinforce these observations, we took confidence intervals of mean visible response time for a

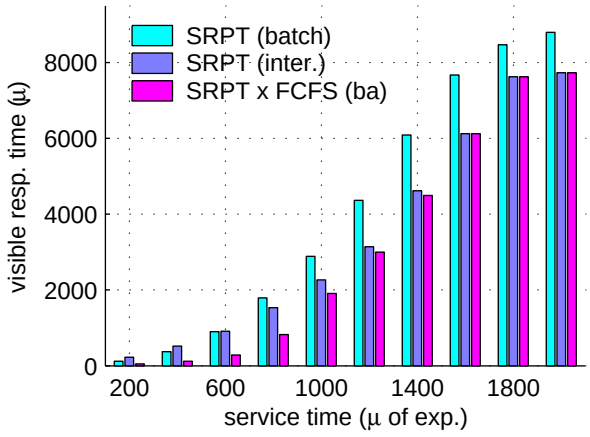

Figure 13: The effect of service time on visible response time. At the low end, there is more opportunity for batchactive to improve performance.

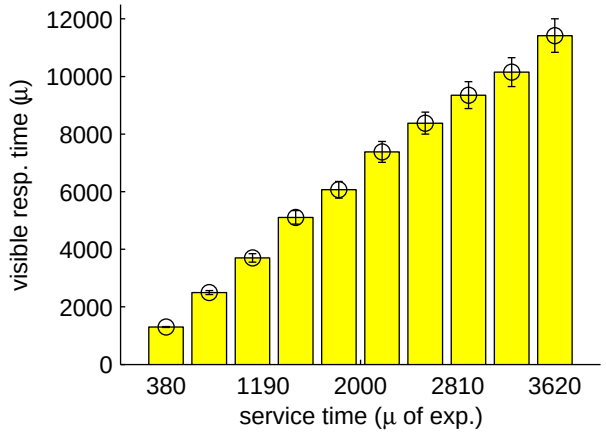

Figure 14: Confidence intervals for a small run suggest that our results are significant. Plotted is the mean visible response time across ten service times. The 95% confidence intervals were from 40 runs using different random seeds at each service time.

small interactive SRPT run in which service time was varied in six minute increments while other parameters were held constant (Figure 14). Each reported visible response time mean and confidence interval was the result of 40 simulations that were started with different random seeds. Out of ten selections of service times, only one pair of 95% confidence intervals overlapped, and all confidence intervals were less than 5% of their respective mean visible response times. A normal probability plot (not shown) of the response times for each service time was sufficiently Gaussian to suggest that the confidence intervals are reliable.

Implementation Our simulator is in C and has modules that simulate user, task, and server behavior. One Perl script executes large numbers of this program to explore the simulation parameter space and stores its results into a MySQL relational database. Other scripts query this database, analyze the results in conjunction with MATLAB, and process them for visualization. Roughly 50,000 runs were performed on a set of shared 2.4 GHz Pentium IV machines each with 512 MB of memory. Most took tens of seconds and less than 100 MB of memory to simulate 16 days of virtual time while some with parameters causing many more tasks to be created took roughly ten minutes to run.[6]

5. RELATED WORK

Speculation is a pervasive concept found at the level of I/O requests, program blocks, and instructions across all areas of computing including architecture, languages, and systems. We apply speculation to tasks using the processor resource. We find great opportunity here for our application domains: there are negligible costs to storing speculative results and no need to isolate or rollback such results if not needed. Here, we focus on the closest related systems work.

Bubenik and Zwaenepoel modeled a cluster of users engaged in software development using a modified `make` tool [6]. At each save of a source code file, their system speculatively runs the compiler. This is possible because build rules are encoded in the `Makefile`. Their work isolates speculative compilations from the rest of the system. This is not needed in our system which stores speculative results until requested. Their simulator models only one task (rebuild) pending per user. Our user model is broader, encompassing users who operate interactively or who submit long batches of speculative work for the kinds of important and pervasive parameter studies described in Section 1. Beyond their reported metrics, we also study resource cost, throughput, and slowdown.

In storage, the TIP system showed how performance increases if the application discloses storage reads in advance of when data is needed [24]. Their work addressed storage and memory questions such as how to balance cache space between prefetches and LRU caches.

In networking, the bandwidth-delay product of current and future grids have spurred speculative approaches to speeding up tightly-coupled applications [9, 19]. Such work examines how to rollback unneeded computation and throttle work so that speculation does not overly consume resources. In contrast, we speculate among multiple independent (non-communicating) tasks. Further, we study how to schedule among task sets from multiple users.

Some work discriminates between speculative and requested network transmissions. Padmanabhan et al. showed a tradeoff in visible response time and fractional increase in network usage when varying the depth of their web prefetcher [23]. In Steere et al.'s system, people manually construct sets of web prefetch candidates and the browser prefetches candidates simultaneously until all are fetched, or until the person initiates new activity [28].

In databases, Polyzotis et al.'s speculator runs queries during the think time when one constructs complex queries. [26]

Some speculative approaches require a user or user agent to disclose speculative work. Others automatically generate speculative work. In our work, Steere's system, and the TIP system, speculative work is generated by the user. In the systems of Padmanabhan et al., Polyzotis et al., and Bubenik and Zwaenepoel, the system queues speculative tasks in addition to scheduling them. An extension to TIP by Chang et al. introduces automatic I/O disclosure generation [8]. Other work transforms a single executable into speculative pieces, yet programmer annotations are needed, and intra-task speculation has not been considered [7, 11]. It does not seem possible to automatically speculate computation in general without domain-specific knowledge or user agents tailored to specific applications.

6. CONCLUSION

Ideally, a grid scheduler would run speculative tasks while users were analyzing completed tasks, minimizing the users' waiting time. The catch is that speculative tasks will take contended resources from users who are waiting for requested tasks unless the two types of tasks can be discriminated.

[6]We desired a batchactive system while exploring our simulator's large parameter space. The service and think times of this research were ideal for batchactive scheduling.

The solution we promote exploits the inherent speculation in application-level search: users disclose speculative task sets, request results as needed, and cancel unfinished tasks if early results suggest no need to continue. We observe that not all tasks are equal — only tasks blocking users matter — leading us to introduce the *visible response time* metric which measures the time between a task being requested and executed, independent of when it was speculatively disclosed. Our *batchactive scheduler* segregates requested and disclosed tasks into two queues, giving priority to the requested queue, toward minimizing mean visible response time.

We simulated a variety of user and task behavior and found that for several important metrics our batchactive model nearly always does better than conventional models where tasks are requested one at a time (interactively) or requested in batches without specifying which are speculative. We have found that visible response time is improved by about 50% on average under a batchactive scheduler, and is at least two times better for 20% of our simulations. These results hold for both size-based (SRPT) and non-size-based (FCFS) policies. Compared to a batch scheduler, for about 40% of the runs, the average user pays for a fourth of the resources.

Batchactive scheduling is adaptive: it is at least as good as interactive or batch as load varies between extremes, and in the middle ranges it is better than both (Figures 7 and 8). Many studies show that idle processing time is abundant [22, 4, 1]. Anecdotally, some report that during 'crunch times,' resources are saturated [13, 21]. While this might occasionally occur now, we believe that once users discriminate between speculative and demand work, this will be even rarer: resources will not be saturated with demand work and batchactive scheduling will provide the benefits shown even during peak usage.

Our cost model charges for only requested tasks to encourage deep speculative disclosure, which enables the scheduler to best reduce visible response time. Certainly it is true that not charging for speculative work means that some computing resources are consumed without being billed. However, our results suggest that disclosed speculation benefits the overall effectiveness of the computing resource. When the system approaches saturation, so that all schemes charge the same amount, the unexposed speculation in traditional batch scheduling delivers fewer results per hour (Figures 9 and 10). And when the system is not saturated, so that resources are going idle anyway, exposed speculation delivers significantly reduced visible response time relative to no speculation at all, so charging only for requested tasks delivers better use of unutilized cycles (Figures 7 and 10).

7. REFERENCES

[1] A. Acharya, G. Edjlali, and J. Saltz. The utility of exploiting idle workstations for parallel computation. *Sigmetrics*, 1997.

[2] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–10, 1990.

[3] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW team. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, Feb. 1995.

[4] R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson. The interaction of parallel and sequential workloads on a Network of Workstations. *Sigmetrics*, 1995.

[5] F. Berman, G. Fox, and T. Hey. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons, 2003.

[6] R. Bubenik and W. Zwaenepoel. Performance of optimistic make. *Sigmetrics*, 1989.

[7] R. Bubenik and W. Zwaenepoel. Semantics of optimistic computation. *ICDCS*, 1990.

[8] F. Chang and G. A. Gibson. Automatic I/O hint generation through speculative execution. *OSDI*, 1999.

[9] N. Chrisochoides, A. Fedorov, B. B. Lowekamp, M. Zangrilli, and C. Lee. A case study of optimistic computing on the grid: Parallel mesh generation. *NGS Workshop at IPDPS*, 2003.

[10] R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory of Scheduling*. Addison-Wesley, 1967.

[11] C. Cowan and H. Lutfiyya. Formal semantics for expressing optimism: The meaning of HOPE. *PODC*, 1995.

[12] D. DeGroot. Throttling and speculating on parallel architectures. *Parbase*, 1990. Abstract of keynote speech.

[13] D. Epps. Personal communication, 2004. R&D director at Tippett Studio since 1992.

[14] M. Giddings and D. Knudson. Xgrid-users email list, Feb. 2004. subjects 'differing resources' and 'What's on your Xgrid' at `http://lists.apple.com/mailman/listinfo/xgrid-users`.

[15] M. Harchol-Balter, K. Sigman, and A. Wierman. Asymptotic convergence of scheduling policies with respect to slowdown. *Performance*, 2002.

[16] J. Hillner. The wall of fame. *Wired Magazine*, 11(12), 2003.

[17] D. Holliman. Personal communication, 2003. Former system administrator for the Berkeley Phylogenomics Group.

[18] N. H. Kapadia, J. A. B. Fortes, and C. E. Brodley. Predictive application-performance modeling in a computational grid environment. *HPDC*, 1999.

[19] C. A. Lee. Optimistic grid computing, 2002. Talk at the Performance Analysis and Distributed Computing Workshop.

[20] The Lemieux Supercomputer. Pittsburgh Supercomputer Center, `http://www.psc.edu/machines/tcs/lemieux.html`, 2003.

[21] T. Lokovic. Personal communication, 2004. Graphics software engineer at Pixar Animation Studios since 1998.

[22] M. W. Mutka and M. Livny. Profiling workstations' available capacity for remote execution. *Performance*, 1987.

[23] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve World Wide Web latency. *Computer Communication Review*, 26(3), 1996.

[24] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. *SOSP*, 1995.

[25] G. F. Pfister. *In Search of Clusters*. Prentice Hall, 1995.

[26] N. Polyzotis and Y. Ioannidis. Speculative query processing. *Conf. on Innovative Data Systems Research*, 2003.

[27] N. Spring and R. Wolski. Application level scheduling of gene sequence comparison on metacomputers. *Supercomputing*, 1998.

[28] D. C. Steere. Exploiting the non-determinism and asynchrony of set iterators to reduce aggregate file I/O latency. *SOSP*, 1997.