A Case Study of Profile Driven Scheduling for a Heterogeneous Cluster of Workstations

Leonidas Kontothanassis, David Goddeau
Cambridge Research Laboratory
HP Laboratories Cambridge
HPL-2004-192
October 27, 2004*

E-mail: {leonidas.kontothanassis, david.goddeau}@hp.com

Clusters of commodity servers are increasingly the platform of choice for running computationally intensive jobs in a variety of industries. Computations such as wind-tunnel simulations, gene and protein analysis, drug discovery, and CGA rendering are run on commodity computers with very successful results. At the same time most cluster environments employ a highly heterogeneous set of machines due to the natural cycle of upgrades and purchases followed by most organizations. In such environments it is likely that some applications may be better suited for one type of machine over another and that the placement of jobs on machines can have a large impact on throughput and job completion time. This paper argues that using profile information to guide scheduling decisions can yield substantial performance improvements over a simple FCFS scheduling policy. In particular we look at a real-world job mix of genomic analysis applications and examine a number of scheduling algorithms that take profile information into account. Our results indicate that the benefits of profile-driven scheduling vary significantly based on the degree of heterogeneity available in the cluster and the variance in execution times of different job types across cluster machines. In addition, we have discovered that affinity effects can also play an important role in improving performance, but plain affinity scheduling is not sufficient in achieving those benefits.

* Internal Accession Date Only
Approved for External Publication

A Case Study of Profile Driven Scheduling for a Heterogeneous Cluster of Workstations

Leonidas Kontothanassis and David Goddeau
{leonidas.kontothanassis,david.goddeau}@hp.com
Hewlett-Packard Laboratories
One Cambridge Center
Cambridge, MA, 02139

Abstract

Clusters of commodity servers are increasingly the platform of choice for running computationally intensive jobs in a variety of industries. Computations such as wind-tunnel simulations, gene and protein analysis, drug discovery, and CGA rendering are run on commodity computers with very successful results. At the same time most cluster environments employ a highly heterogeneous set of machines due to the natural cycle of upgrades and purchases followed by most organizations. In such environments it is likely that some applications may be better suited for one type of machine over another and that the placement of jobs on machines can have a large impact on throughput and job completion time. This paper argues that using profile information to guide scheduling decisions can yield substantial performance improvements over a simple FCFS scheduling policy. In particular we look at a real-world job mix of genomic analysis applications and examine a number of scheduling algorithms that take profile information into account. Our results indicate that the benefits of profile-driven scheduling vary significantly based on the degree of heterogeneity available in the cluster and the variance in execution times of different job types across cluster machines. In addition, we have discovered that affinity effects can also play an important role in improving performance, but plain affinity scheduling is not sufficient in achieving those benefits.

1 Introduction

Clusters of commodity computers are a very successful platform for a large number of enterprises with extensive computational needs. Furthermore, they form the natural building blocks for grid systems [1, 2] which are rapidly gaining acceptance as the preferred way to organize an institution's computing resources. Such environments make it possible to consolidate computing resources into a small number of data centers, share the infrastructure costs across many departments, and reduce the IT management and maintenance overheads.

Computations submitted to these kinds of environments range widely from wind-tunnel simulations, to gene and protein analysis, to CGA rendering and many others. While the spectrum of potential applications is large, it is quite common for a particular cluster to see the same application submitted multiple times with slightly different inputs for every run. This behavior results from the nature of the computations that users want to see done, which include parametric studies, multiple simulations with different initial conditions, or searching for matches of multiple query strings against a large database. Such user needs imply that the same application will be run multiple times with very similar inputs. In some extreme cases the system administrator for a cluster may actually restrict the executables that can be run on the cluster to a small set offering a specific service (i.e. the Blast service from the National Center for Biotechnology Information [3]).

While the use of clusters as compute resources continues to expand, market forces dictate that most cluster environments will employ a highly heterogeneous set of machines. Most companies upgrade their IT infrastructure piecemeal thus making it commonplace for machines with widely different computational capabilities to coexist within a single cluster environment. Even when a company decides to standardize on a particular architecture, machines from different generations of that architecture will almost certainly coexist.

The combination of a small number of repeatable jobs and a heterogeneous cluster environment leads naturally to the realization that different jobs may be better suited to different machine types. Previous work [4, 5] has examined the impact of scheduling heuristics on heterogeneous distributed systems and has found them to provide substantial benefits for artificial workloads. Our work extends these findings by looking at a real-world work-

Machine Type	Cpu Speed	Memory Size	Disk Type
Type 1	733Mhz	256Mbytes	IDE (40MB/s)
Type 2	2.8Ghz	1.5Gbytes	IDE (40MB/s)
Type 3	2.8Ghz	256Mbytes	IDE (40MB/s)
Type 4	2.8Ghz	1.5Gbytes	SCSI (60MB/s)
Type 5	2.8Ghz	256Mbytes	SCSI (60MB/s)
Type 6	525Mhz	8Gbytes	SCSI (60MB/s)

Table 1: Machine types available for experiments.

load and also examining the interplay of affinity effects on such a heterogeneous environment.

In particular this paper analyzes a workload from the *National Center for Biotechnology Information* that is based on the popular Blast application suite [6] and that captures the complete set of jobs run on their cluster over the course of a day. We then examine the execution profiles of the applications on a variety of machine types and discover that indeed different jobs exhibit varying amounts of sensitivity to machine characteristics. Certain jobs are CPU bound and thus depend primarily on CPU speed, while they see little or no benefit from larger memories and better disk subsystems. Other jobs are more I/O bound and are thus less sensitive to CPU speed and more so to the capabilities of the disk subsystem. Finally, we discover that affinity scheduling can play an important role for certain job types but only when the working dataset can fit in a machine's file buffer cache. Given these findings, we proceed to examine a variety of scheduling algorithms that try to address different aspects of the problems. Our simplest strawman algorithm is plain first-come first-served (FCFS) scheduling. We also present an algorithm that attempts to maximize affinity, one that attempts to optimize minimum completion time [4], and two variations on this last algorithm that attempt to minimize completion while taking affinity effects into account. The main difference between these last two variations is that one assumes a priori knowledge of job runtimes on each machine type, while the other goes through a discovery phase to learn the relevant information.

Our results indicate that focusing on affinity alone in a heterogeneous environment results in a deterioration of performance when compared to FCFS scheduling. The algorithm that optimizes minimum completion time provides a modest reduction of up to 10% in average job completion time over FCFS, while the last two algorithms that attempt to minimize completion time while taking affinity effects into account perform the best, with average completion times improving by up to 20% over FCFS scheduling. We also find that the discovery phase has no material impact on average completion times, with the performance between the last two variations of the minimum completion time algorithm being practically indistinguishable.

It it not the intention of this paper to come up with an optimal algorithm for scheduling based on profile data. Rather, we want to advocate the principle of collecting profile information as jobs run, provide a simple method for collecting such profile information for an important application class, and then show how to use this information to guide future scheduling decisions. We also want to demonstrate that affinity effects need to be considered when scheduling for heterogeneous environments but they should not be the primary focus of the scheduler as that can be detrimental. We have therefore settled for simple, easy to explain scheduling algorithms that demonstrate the principles and still provide significant performance advantages over typical schedulers that do not take profile information into account.

The rest of this paper is organized as follows. Section 2 presents the different flavors of Blast, our data collection methodology, and the simulation environment used to evaluate the different scheduling policies. Section 3 analyzes the properties of the *NCBI* Blast workload and its constituent applications, while section 4 presents the scheduling algorithms we evaluated against this workload. We discuss the performance ramifications of the various scheduling algorithms in section 5. The final two sections (6 and 7) discuss related work, summarize our conclusions, and outline future research directions in this area.

2 Methodology

In order to evaluate our premise we have started with a workload from the National Center for Biotechnology Information. For each different application in this workload we collected profile data of multiple runs on six different types of machines. We then used this collected runtime data to drive a simulation engine that evaluated the performance of different scheduling policies on our workload on a variety of cluster configurations.

Our workload consists of the popular Blast application suite used by biologists to perform searches of nucleotide

and protein databases. There are five applications in the Blast suite, although it is constantly being expanded and refined to add new functionality. The five applications found in our workload are:

- **blastn:** A tool for comparing a nucleotide query sequence against a nucleotide sequence database.
- **blastp:** A tool for comparing an aminoacid query sequence against a protein sequence database.
- **blastx:** A tool that compares the six-frame conceptual translation products of a nucleotide query sequence (both strands) against a protein sequence database.
- **tblastn:** A tool that compares a protein query sequence against a nucleotide sequence database dynamically translated in all six reading frames (both strands).
- **tblastx:** A tool that compares the six-frame translations of a nucleotide query sequence against the six-frame translations of a nucleotide sequence database. Typically used when trying to deduce connections between nucleotide sequences of different organisms by comparing the proteins that those sequences produce.

A complete description of the algorithmic properties for all five Blast applications can be found in [6]. From a systems design perspective those applications exhibit large variations on the amount of computation they perform relative to the amount of I/O they incur. They also show significant variation on their sensitivity to the amount of memory available on the machine. This makes them good candidates for profile-driven scheduling since different applications require different types of resources from a machine.

Of equal significance to the executable is the database against which the comparison is run. Clearly, larger databases will results in higher I/O demands from the applications and longer execution times. As it turns out the database space is dominated by two databases *nr* and *nt*, which are the non-redundant, all-inclusive databases of all known protein (*nr*) and nucleotide (*nt*) sequences. The (*nr*) database is used by the blastp and blastx executables, while blastn, tblastn, and tblastx searches are run against the *nt* database. Those two databases account for well over 80% of all searches. The remaining databases have significantly lower frequencies of access and significantly lower runtimes since by definition they have to be smaller than the all inclusive ones. For the purpose of clarity, we have removed the small number of searches against those smaller databases from our trace.

Once we had determined the workload for evaluating our algorithms we had to decide the cluster environment on which to do the evaluation. Rather then picking a single environment, we decided to use simulation and evaluate a number of possible environments with differing degrees of heterogeneity. To that extent we developed a trace-driven, discrete event simulator. The simulator reads in the profile information for every job type and machine type combination, the cluster configuration, and the arrival times of every job in our trace. It then schedules these jobs on machines of our simulated cluster based on a user-defined scheduling policy. We evaluated four different scheduling policies although many more are possible. Our schedulers include:

- *FCSC*: a standard first-come first serve scheduler,
- *Affinity*: a competitive scheduler that attempts to maximize database affinity (i.e. schedule a job accessing a particular database on a machine where that database was last accessed),
- *Greedy-1*: a greedy scheduler that attempts to place jobs on the best possible machine for each job based on profile data,
- *Greedy-2*: a variation on the greedy scheduler that also takes affinity effects into account, and
- *Greedy-3*: a further variation to the Greedy-2 scheduler that attempts to learn the profile information based on the outcome of its scheduling decisions.

The last three schedulers are very similar in nature and are all based on the minimum cost scheduler of [4, 7]. Their differences are explained in detail in section 4.

One of the most important problems that we had to solve was collecting profile information for every job and machine type combination. One approach would be to try and get a job profile that is machine independent and then extrapolate how the job would run on different machine types based on a projection of the job profile and certain salient machine characteristics. This is akin to application-specific benchmarking [8]. A separate approach, which we have chosen instead, is to come up with a small number of salient characteristics that can be used to classify machines into machine types. We can then simply run each job type on every machine type and collect information on job execution times as the jobs complete. It is important to choose a relatively small number of machine characteristics that identify a machine type in order to avoid having to collect an extremely large amount of profile data. It is equally important to identify the arguments to an executable that have an impact on runtime in order to come up with a minimal distinct number of job types. Once both machine types

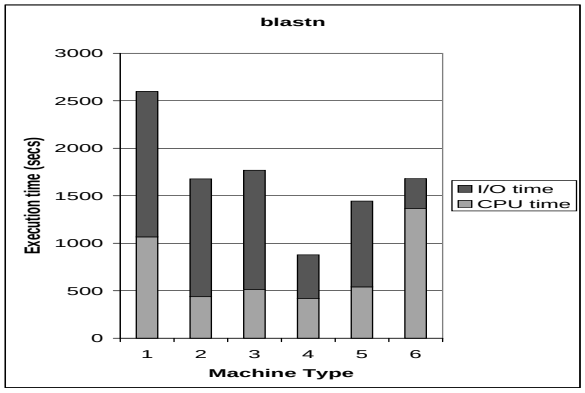


Figure 1: CPU and I/O components of blastn on different classes of machines.

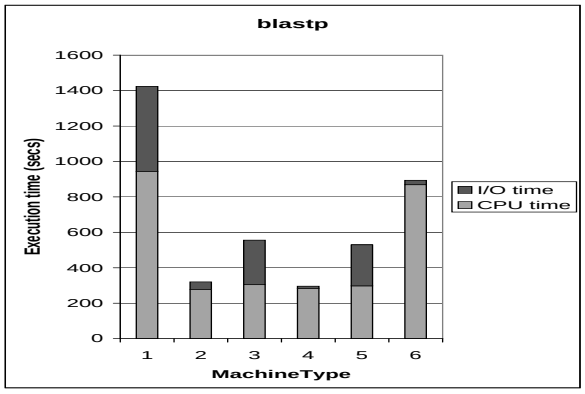


Figure 2: CPU and I/O components of blastp on different classes of machines.

and job types have been correctly identified we can then build a lookup table that the scheduler can consult before making scheduling decisions. For the learning version of our scheduling algorithms this table can be populated based on the outcome of previous scheduling decisions. This technique has been used successfully in other scheduling-problem settings like disk scheduling [9].

For our study we have decided to classify machines based on the CPU speed, the amount of memory available, and the type of disk present in the machine. These characteristics are sufficient for differentiating one machine from another for our application suite. Additional characteristics may need to be taken into account for other applications/workloads. Examples of such additional characteristics may include processor cache size, floating point rating of the processor, and network card characteristics. On the application side the only relevant arguments are the executable name and the database being used. Given that the vast majority of our searches involve a single database for any given executable, the dimensionality of the application space is reduced to a single dimension.

Based on the simple classification scheme described above, we collected profile information for six different types of machines with varying CPU, memory, and disk characteristics. Our machines had CPU speeds between 525Mhz and 2.8Ghz, memory sizes between 256Mbytes and 8Gbytes, and disk subsystems capable of transfer rates between 40Mbytes/sec and 60MBytes/sec. The complete list of available machine types is listed in Table 1.

3 Application and Workload Properties

3.1 Application Profiles

We have collected profile information for each of the applications in our workload against all six types of machines that were available to us. Our method for collecting profile information has been to run each job on every machine type multiple times and verify that the variance between successive runs is small. This is indeed the case for all job and machine type combinations in our workload. We then look at the statistics aggregated by the kernel under `/proc(8)`, in order to determine total execution time, time spent in user and kernel space (also known as CPU time), and time spent performing I/O operations. Obviously the sum of CPU and I/O time needs to be equal to the total execution time for the application. Additional statistics (i.e. the number of I/O operation and whether they hit in the buffer cache or not) are available but we did not use them for the purposes of this study.

Figure 1 shows the breakdown in execution time between CPU and I/O components for blastn. Blastn has a significant I/O component which can range anywhere between 19% and 74% of total execution time depending on the relative speeds of the CPU and I/O subsystems. I/O is least significant for the machines with the slowest processor and fast SCSI disk subsystems, while it has the greatest impact on the fast processor machines with IDE disk drives. The amount of memory available to the application has a smaller, but still significant impact on performance, with larger memory sizes resulting in better runtimes. The reason for the relatively large I/O component is that the application has to find a match of a relatively simple query string against a very large database. The raw database file is a little over 11Gbytes while the various index files used to speed up the search occupy almost 4Gbytes of space. This implies scanning large amounts of on-disk data while performing relatively little computation for each piece of data being fetched from disk.

Blastp ranks second with respect to I/O usage in our application suite. Once again the impact of I/O on to-

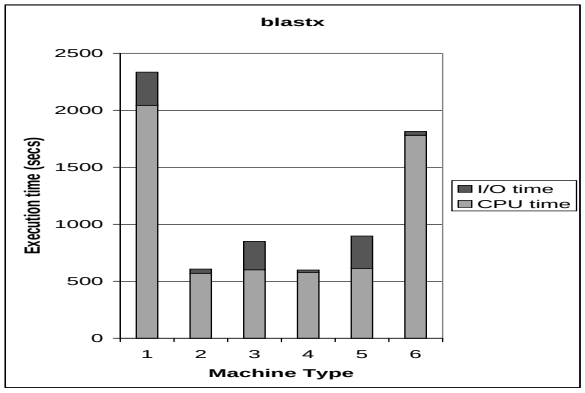


Figure 3: CPU and I/O components of blastx on different classes of machines.

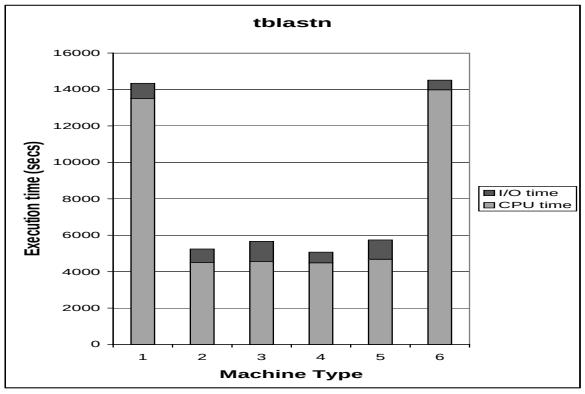


Figure 4: CPU and I/O components of tblastn on different classes of machines.

tal execution time depends on the relative capabilities of the CPU and I/O subsystems. I/O contribution to total execution time ranges from as little as 3% for machine type 6 (see table 1) to as high as 45% for machine type 3. Interestingly enough the amount of time spend in I/O is heavily influenced by the amount of memory available on the machine. This can be explained by looking at the on-disk size of the protein database being searched. The all-inclusive protein database used in the vast majority of searches occupies 1Gbytes of space, while the indices used to speed up the search account for 1.3Gbytes of disk space. The larger memory machines have enough memory to hold almost all of this data in memory and a lot of disk accesses are being replaced by hits in the file buffer cache. The runtimes and their breakdown in CPU and I/O components for blastp are summarized in figure 2.

Blastx is similar to blastp in that it operates on the same protein database. Therefore we expect to see a similar behavior across machines with memory sizes having a significant impact on runtimes. There exist substantial

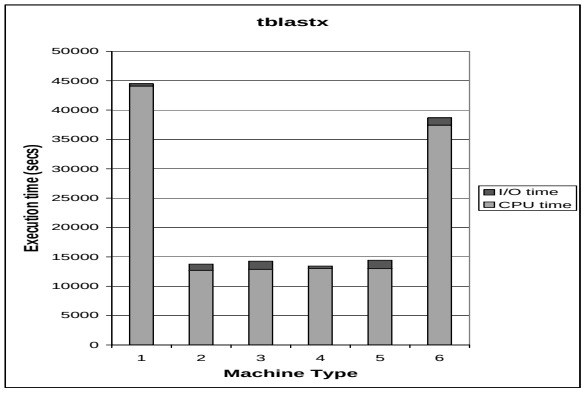


Figure 5: CPU and I/O components of tblastx on different classes of machines.

differences however, with the most important one being that I/O times are never more than 30% of total execution time. This is explained by the more expensive computation that the application performs which includes the translation of the nucleotide query string into multiple peptide strings which are then compared against the database. As a consequence blastx has to perform multiple comparisons against each incoming piece of data, and this reduces the relative importance of I/O when compared to blastp. The runtimes and their breakdown in CPU and I/O categories for this application are summarized in figure 3.

The final two application (see figures 4 and 5) are significantly more CPU-intensive and their execution time is dominated by the CPU-speed characteristic of the machine. This can be seen by the small variations in execution time for the machines that have the same CPU speed but differ in memory and disk configurations (machine types 2,3,4, and 5). The explanation for this behavior lies in the computationally expensive operations associated with translating the nucleotide database to the multiple peptide strings that are used in comparisons. Furthermore those translated strings cannot take advantage of indices (those indices cannot precomputed as is the case for the first three application) and thus result in more exhaustive searches. As a consequence tblastn and tblastx take an order of magnitude longer to complete than their counterparts. The reason they do not completely dominate our workload is that they are used infrequently relative to the first three applications.

3.2 The Impact of Affinity Scheduling

In addition to understanding how particular applications behave on certain classes of machines, another factor that affects performance is whether the application has built data affinity on a particular machine or not. Affin-

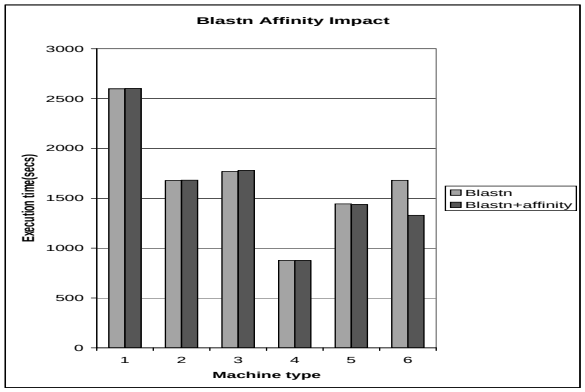


Figure 6: Runtimes for blastn in the presence and absence of affinity scheduling.

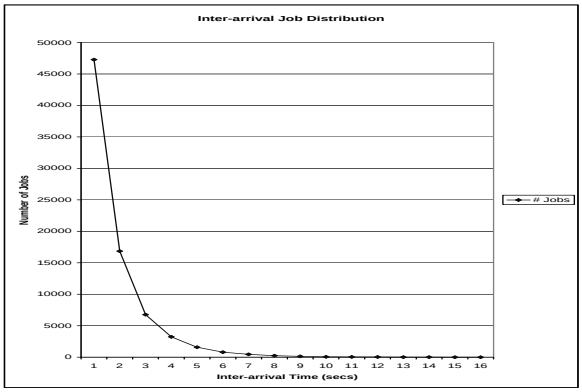


Figure 8: Distribution of jobs according to their inter-arrival intervals.

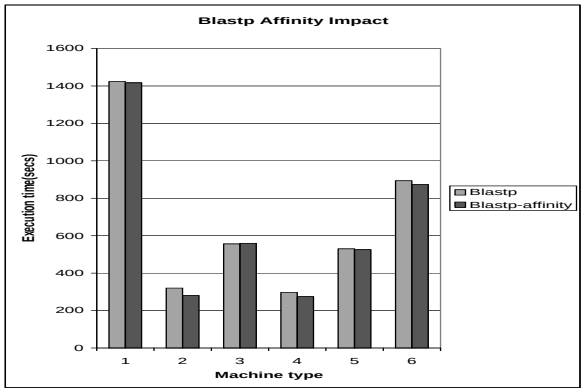


Figure 7: Runtimes for blastp in the presence and absence of affinity scheduling.

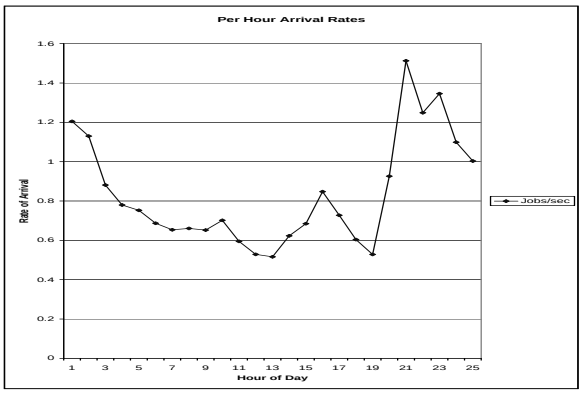


Figure 9: Average job arrival rate for every hour of our trace.

ity scheduling can have a large impact on an application's runtime, but only if a number of constraints are satisfied. First and foremost the application needs to have a significant I/O component that can be reduced by caching the data it needs to access in the machine's file buffer cache. This implies that in our application suite, blastn and blastp are the most likely candidates for affinity-derived benefits. Second, the machine's memory and consequently the file buffer cache needs to be large enough to hold the data being accessed across runs. We devised an experiment in order to determine if any of our applications would benefit from affinity scheduling on any of the machine types available to us. The experiment consists of wiping the buffer cache on the machine clean, by `mmapping(2)` a large enough file into memory. Then we run the targeted application twice in succession and look at the difference in runtime between the first and second run. Any reduction in runtime is due to affinity effects. Each such experiment is repeated multiple times to ensure that results are deterministic and repeatable.

Figures 6 and 7 show the impact of affinity scheduling for our two most I/O intensive applications on all classes of machines. Unfortunately for our machine classes and applications, affinity scheduling seems to only have a limited impact. For most of our machine types the available amount of memory is not sufficient to effectively cache the working set of the applications. There are two exceptions to this observation. One is that machine type 6 has enough memory to cache the necessary data for blastn with performance improving by 21% when the file buffer cache is warm when the application is run. The other is machine type 2 with blastp. In this case the machine has enough memory to cache the data and its I/O system is bad enough that caching provides substantial benefits. The performance improvement due to affinity effects is 12.5%. For the remaining applications and machine types either the I/O component was not significant

Application	Percentage
blastn	55.8%
blastp	18.8%
blastx	21.2%
tblastn	1.4%
tblastx	2.8%

Table 2: Percentages of each job type in the NCBI trace.

enough, or the machine configuration was such that affinity effects were not noticeable. We have also looked at other, smaller genomic databases that fit in the memory of machines with fast CPUs and slow I/O systems and have indeed verified that in those cases, affinity is an important factor that the cluster scheduler should take into account.

3.3 Workload Description

The workload used for the majority of our simulations was derived from a 25 hour trace of blast jobs submitted to NCBI. The trace starts at 1:00pm of a day and ends at 2:00pm the following day. As was noted earlier, jobs accessing minor databases (UniVec, Mito, etc) were removed from the trace for simplicity as the *nt* and *nr* database jobs dominated both the job percentages and the total compute time. Table 2 shows the percentage of each type of job in the trace.

As would be expected of a job mix arriving from many sources to a single service, the jobs are independent and their inter-arrival times are approximately exponentially distributed with a job arriving, on average, every 1.14 secs. Figure 8 shows the distribution of inter-arrival times over the entire twenty four hours of the trace. As can be seen most jobs have an inter-arrival time of one second or less, but the actual distribution has a relatively long tail with some jobs having inter-arrival gaps of as long as fifteen seconds. The shape of the distribution of inter-arrival times within each hour in the twenty four hour period is very similar. What changes across hours is the mean of the inter-arrival times. As a consequence, a relatively precise model of the job mix can be a Poisson process with time-varying inter-arrival times. Figure 9 shows the variation of the inter-arrival times over the whole 24 hours.

While our trace represents a real world job mix, we wanted to ensure that our results were stable across minor variations in the order and rate of job arrival. In order to achieve this we also generated ten synthetic job traces with approximately the same length and characteristics of the NCBI trace. Our trace generator simply used the job percentages and hourly inter-arrival times of the NCBI trace to drive a simulated Poisson process with different initial random seeds. We then verified that our results hold true across these artificially generated traces, even though we only report results from the real NCBI trace.

4 Scheduling Algorithms

In this section we describe the heuristic algorithms we have tried in the context of profile driven scheduling. All of our algorithms assume that the times of job arrivals are unknown to the algorithm and all scheduling decisions have to be made online as jobs enter and leave the system. Three of those algorithms assume that profile information for the arriving jobs is known a priori. For lack of better term we will call these algorithms offline-profile algorithms. The fourth algorithm is an online variation of one of the offline-profile versions, which makes no assumption about runtimes but learns the profile information as jobs arrive and are scheduled into the system.

Scheduling based on profile information can be shown to be NP-complete even for the complete offline case where arrival times and profile information is known ahead of time, by reducing it to one of the many NP-complete scheduling problems [10]. Therefore an exact algorithm is unlikely to be time effective and heuristics have to be employed. Online algorithms where job arrival times are unknown a priori have similar complexity properties to their offline brethren [11]. In addition any online algorithm attempting to learn the profile information, requires the scheduler to come up with a schedule for the learning phase which can be a challenging task on its own. However, if the number of machine types is small and the number of jobs per job type is large this last part can be finessed through exhaustive search. By assigning each job type to each machine type once we can acquire the relevant profile information in a short amount of time relative to the length of the whole execution schedule. Fortunately, even the most heterogeneous clusters rarely have more than a small number of machine types, thus making the exhaustive search approach for learning profile data a reasonable choice.

We have experimented with four different heuristic algorithmics with promising results. The first of those algorithms (*greedy-1*) is depicted in figure 10 and is following a greedy technique of assigning jobs to machines in the cluster.

The basic rational of the greedy algorithm is to try and assign each job to the machine which would run it the fastest, as indicated by the profile data. This will naturally lead to conflicts as more than one job will prefer the same machine, especially since some machines will be more powerful than others. In order to break those conflicts the algorithm decides to assign a job to a machine based on the benefit it would see over its second

```
foreach job in Queue {
    foreach mach in free Machines {
        Cost[job][mach].cost = getProfileData(job, mach);
        Cost[job][mach].mach = mach;
    }
    sort Cost[job] on cost;
}
foreach job in Queue {
    Prefs[0].job = job;
    Prefs[0].mach = Cost[job][0].mach;
    Prefs[0].benefit = Cost[job][0].cost - Cost[job][1].cost;
    numConflicts = 0;
    foreach job1!=job in Queue {
        if (Cost[job1][0].mach == Cost[job][0].mach) {
            numConflicts++;
            Prefs[numConflicts].job = job;
            Prefs[numConflicts].mach = Cost[job][0].mach;
            Prefs[numConflicts].benefit =
                   Cost[job][0].cost - Cost[job][1].cost;
        }
    }
    prefJob = max(Prefs) according to benefit;
    mach = prefJob.mach;
    assign(prefJob.job, mach);
    remove(prefJob.job, Queue);
}
```

Figure 10: A greedy profile-driven scheduling algorithm

```
foreach job in Queue {
    foreach free machine in Cluster {
        if (machine.hitsCache(job) {
           assign(job, machine);
   remove(job, Queue);
        }
    }
}
// Remaining jobs in Queue did not hit in machine caches
if (machinesAvailable()) { // If there are still free machines
    foreach job in Queue {
        foreach free machine in Cluster {
            penalty = cachePenalty(job, machine);
            if (penalty <= curTime - job.submissionTime)
                assign(job, machine);
                remove(job, Queue);
            }
        }
    }
}
```

Figure 11: An affinity-based profile-driven scheduling algorithm

best choice. The benefit is computed as an absolute difference in runtime, rather than a percentage difference. This would allocate long running jobs favorably to fast machines since even small percentage improvements can result in substantial absolute runtime reductions. The downside of this algorithm is that it does a relatively poor job of taking affinity issues into account.

An alternate algorithm we considered (known with the term *affinity scheduler*), is one that seeks to optimize affinity scheduling only. We did not expect this algorithm to provide much benefit for our default workload and we mainly wanted to demonstrate that taking affinity issues into account is not sufficient for heterogeneous environments. The algorithm is presented in pseudocode in figure 11.

The basic rationale behind the affinity algorithm is to assign a job to a machine if that machine had previously run a job of a similar type and thus would benefit from affinity effects. When no such machine is available then the algorithm determines a penalty associated with the assignment. This penalty is expressed by the formula $0.5 * (curJob_{Cold} - curJob_{Hot}) + (prevJob_{Cold} - prevJob_{Hot})$ which tries to balance the benefit the current job would accrue from acquiring affinity vs. the benefit that the previously run job had accrued. This penalty is then compared with current waiting of the job in the queue. If the waiting time exceeds the penalty then the job is considered to have waited long enough and is assigned to the available machine. Otherwise the job remains in the queue until either a machine with appropriate cache contents becomes available, or until the job has aged the right amount of time. It can be shown that for a simple case with only two job types and one machine type this algorithm has a competitive ratio of two relative to an optimal algorithm. Since our interest is in examining the behavior of this algorithm on a realistic workload, rather than coming up with worst case scenarios we will not examine its theoretical properties any further.

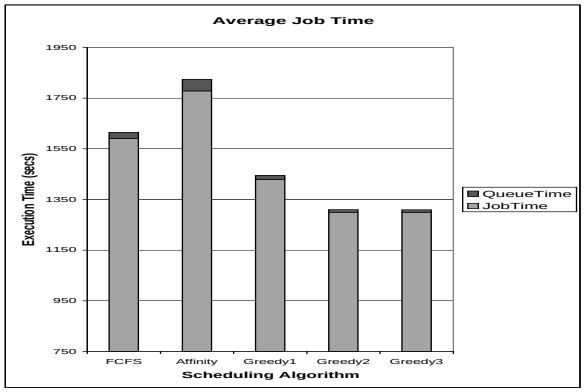
We also examined a variation to *greedy-1* which behaves in a similar fashion but augments the number of machine types by taking into account affinity information. Under this algorithm the job profile information table gets an additional dimension which is based on the type of job that was last run on a machine. This way the algorithm takes into account not only the machine characteristics of CPU, memory size, and disk type, but also the kind of job and database that was last accessed on that machine. The algorithm does not attempt to maximize affinity but will take advantage of it when it is available and it results in reduced job completion times. We term this algorithm *greedy-2*.

Finally, we looked into converting the *greedy-2* algorithm that takes affinity information into account, into one that learns the profile information online. The conversion is relatively straightforward and requires minimal changes to the version with a priori knowledge. Looking at the algorithm in figure 10 we can see that it consists of two phases; the first phase where the cost of running a job on each machine is computed, and the second phase where the job to machine assignment is calculated based on these costs. All we need to do to convert it to an online-profile algorithm is assign a low cost to job/machine combinations in phase one of the algorithm when the true cost is unknown. Then when a job completes, we can update the profile information table with the newly collected information. This approach will force the algorithm to quickly explore the machine space since unknown machines to each job type will be seen as preferred due to their artificially low initial cost. Once the exploration phase is over the algorithm simply behaves like the offline-profile greedy algorithm. We call this last algorithm *greedy-3*.

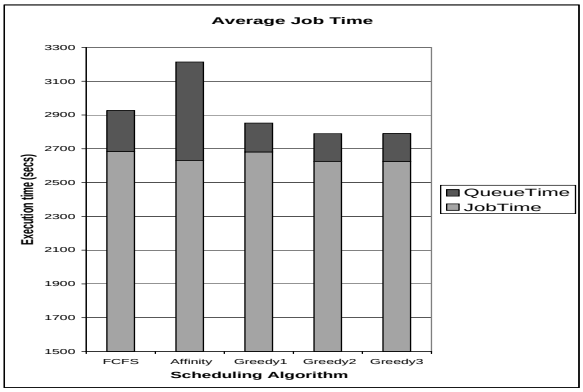
5 Performance Results

In this section we present performance results for our schedulers and a strawman first-come first-serve scheduler, that simply assigns an incoming job to the first available machine in the cluster. We have experimented with multiple different cluster configurations and job traces in order to understand the interaction of our schedulers in the presence of differing amounts of heterogeneity, job arrival rates, and variety in the types of jobs coming into the system. We present results for two types of clusters and two different jobs traces for a total of four possible combinations.

Our first cluster configuration consists of an equal number of machines from every machine type presented in table 1, while the second configuration is a completely homogeneous cluster with "type 6" machines. The reason that we have included a homogeneous cluster in our results is to demonstrate that our algorithms do not impose any significant burden in the absence of heterogeneity. Our first job trace consists of the dominant executable/database pairs from the *NCBI* job trace. We have omitted a small percentage (approximately 15% of the total) of the original trace since it consisted of small jobs whose execution times were less than a few seconds. Such jobs would have no impact on the behavior of our schedulers and would simply make presenting the results cumbersome. The second job trace selected only the jobs associated with the blastn and blastp executables out of the full trace. The reasons behind this selection was that these jobs exhibited the greatest benefits from affinity scheduling and we were interested in seeing how well an affinity scheduler would perform when compared to our greedy schedulers in such a job mix. For this same reason our homogeneous cluster consists of the large-memory



Heterogeneous Cluster



Homogeneous Cluster

Figure 12: Average Job Completion Time for the full NCBI trace, on heterogeneous and homogeneous cluster configurations using different schedulers

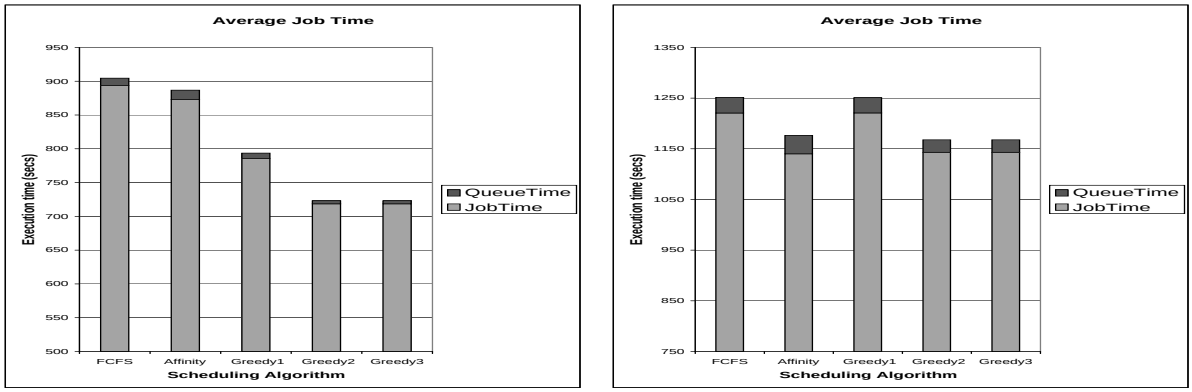
machines since this is the one machine type for which affinity effects are most pronounced.

In order to determine the appropriate size for our cluster type/job trace combinations we run several simulation experiments with different cluster sizes using the FCFS scheduler. Our goal was to discover the cluster size at which FCFS scheduling could keep up with the rate of job arrival in each job trace. Once that size was determined we simply run the additional schedulers using the same cluster size and compared the schedulers based on the average job completion time. This is only one of the possible ways for comparing the performance of different schedulers. An attractive alternative is to determine the minimum cluster size that can keep up with a particular job trace for each scheduling algorithm. We only run a few experiments for this alternative in order to confirm our intuition that shorter job completion times also imply that you can keep up with a particular job trace using a smaller cluster size. This is indeed the case, although the actual connection between average job completion time and needed cluster size is not linear. We will not focus on the "cluster size" metric for the rest of the section. We simply mention it here, since one of the attractions of our proposed scheduling policies is reduced cost to the cluster operator for the same job throughput.

Figure 12 shows the average job completion time for our different schedulers on the full NCBI trace for both a heterogeneous and a homogeneous cluster. Completion time is broken down to time spend running and time spent waiting in the cluster queue to be assigned to a machine. Since we selected our clusters to be of an *appropriate* size for the FCFS scheduler we expect the queue time to be small which is born out by the data. Our homogeneous cluster shows relatively small differences between the different schedulers with the Affinity scheduler being 9.8% worse on average than FCFS. The *greedy-1* scheduler that does not take affinity information into account is almost indistinguishable to FCFS, while the remaining two greedy schedulers improve performance by 4.6%. The *greedy-3* scheduler tracks the performance of the offline *greedy-2* version extremely closely and is practically indistinguishable from it.

Almost all the deterioration of the Affinity Scheduler stems from its policy of waiting for an affinity-appropriate machine for a certain amount of time before assigning a job to any available machine. As a matter of fact when examined closely, the actual runtime of the jobs decreases but the increase in queue waiting time overwhelms any benefits from the decreased runtime. The affinity-conscious greedy algorithms benefit from a reduction in both execution and queueing times with approximately two thirds of the total improvement coming from a reduction in queue time. Looking at the individual jobs most of the improvements come from blastn which experiences a 6% reduction in average execution time, while the remaining programs' execution times improve anywhere between 1% and 3%.

Moving to a heterogeneous cluster amplified the differences between the different schedulers dramatically. The competitive scheduler becomes 13% worse on average from FCFS, the affinity-oblivious greedy scheduler improves performance by 10.5%, while the performance advantage of the affinity-conscious greedy algorithms over FCFS increases to 19%. This is somewhat expected. First and foremost the heterogeneous cluster offers less opportunities for affinity scheduling than the homogeneous cluster did. Many of the machine/application pairs offer no advantage for affinity scheduling and as such are expected to see little benefit from trying to achieve it. This ends up hurting the affinity scheduler since it



Heterogeneous Cluster

Homogeneous Cluster

Figure 13: Average Job Completion Time for the reduced trace on heterogeneous and homogeneous cluster configurations using different schedulers

prioritizes affinity scheduling over everything else. On the other hand as we saw in section 3, our application suite shows remarkable variability of execution time on the different machine types. The greedy schedulers exploit this variability well by assigning jobs to their best-matched machines. As it turns out there is some additional benefit to be had by paying attention to affinity, as is shown by the further improvement in execution times achieved by the affinity-conscious greedy schedulers over their affinity-oblivious counterpart.

Looking into the detailed job statistics for the affinity-conscious greedy schedulers we see that individual job-type performance improvements range from as little as 8.5% for tblastn to as high as 24.5% for blastp. The remaining three applications see performance improvements between 18.5% and 19%. The percentage improvement correlates well with the variance in execution time seen by each job type across the different machine types. Blastp has the highest variance, while tblastn has the lowest. The remaining applications fall somewhere in between. Intuitively, higher variance in execution time implies more opportunities for a profile driven scheduler to outperform one which does not take expected runtimes into account.

We were also interested to see how our various profile-driven schedulers would perform on a job mix that could derive benefits from affinity scheduling. In particular we were interested in figuring out whether the affinity-conscious greedy scheduler would discover affinity effects and how well it would take advantage of such effects when compared to the Affinity scheduler that targets affinity effects explicitly. To that extent we modified our job trace to only include the jobs associated with the blastn and blastp executables. Those two executables see the greatest benefits from affinity scheduling and a job mix containing only those would be expected to favor our affinity scheduler. Our results are summarized in figure 13.

On the heterogeneous cluster we see that the affinity scheduler improves performance over FCFS scheduling by a small amount (approximately 2%), but continues to underperform the greedy schedulers which improve performance by 12%, 20%, and 20% for *greedy1*, *greedy2*, and *greedy3* respectively. The main reason behing this behavior is that the majority of machines provide no benefits for affinity scheduling and thus the waiting time in the queue associated with the affinity scheduler overwhelms the benefits derived by affinity. Similar to the full trace we see that the affinity-conscious greedy schedulers derive an additional performance improvement over their affinity-oblivious counterpart, and the cost of discovering profile information incurred by *greedy3* has no impact on performance.

On the homogeneous cluster we finally have a configuration that favors the affinity scheduler. Indeed, the affinity scheduler improves performance over FCFS by 6%. Interestingly enough the exact same performance improvement is achieved by the affinity-conscious greedy schedulers. However the affinity-oblivious greedy scheduler fails to achieve any performance improvements and simply tracks the performance of FCFS.

We have also run additional simulations on job traces that are modeled against the real job trace obtained from NCBI. The main reason for these additional runs was to establish that our results were not sensitive to small variations in job interarrival times, order of job arrival, and the exact proportions of each job type in the mix. We have verified that our results hold true across ten different traces that were generated to have similar properties to the original trace, although the actual order, frequency

of job arrival and job mix composition vary from trace to trace.

6 Related Work

Scheduling for clusters of computers has received a considerable amount of attention in the literature for both the offline and online [11] versions of the problem, including the case where runtimes need to be learned through system observation [12]. Our goals in this paper are quite different from this work, since we are more interested in understanding the behavior of certain simple algorithms for a real-world workload rather than improving on well known theoretical bounds.

In addition to the theoretical work, at least three systems for cluster scheduling have been commercially developed; the Platfrom Load Sharing Facility [13], the Portable Batch System [14], and the Condor Project [15, 16]. Most of those systems are concerned with discovery of computing resources, ensuring fairness of scheduling, restarting jobs upon failures, and managing user access, quota, and other issues of a cluster environment. Those are incredibly important issues and every cluster scheduling environment needs to address them. Our work seeks to build on top of those environments and to provide scheduling policies based on profile information, that can enhance the throughput of cluster systems.

Scheduling for networks or clusters of workstations has primarily focused on the issue of load balancing and load sharing [17, 18, 19, 20], or on the issue of fairness and proportional resource allocation [21]. We are willing to tolerate some umount of unfairness in our work, in order to maximize overall job throughput. However, our work can easily be set in the context of schedulers that pay attention to load balancing and fairness. In that context our profile driven job placement would be circumscribed by the constraints of the higher level scheduler that resolved the load balancing and fairness constraints.

Affinity scheduling for cluster systems has also received attention in the past [22, 23, 24] with proposals that trade off between affinity-based and load balancing-based scheduling. While affinity is a dimension that our profile driven scheduler takes into account it is not the only one. If a job can build affinity on a cluster node, then our profile data would indicate this and our scheduler would take it into account. However, many jobs do not benefit from affinity scheduling as our profile data indicates and enforcing affinity scheduling for those jobs would prove counterproductive. Our scheduling policies have no built in assumptions about what constitutes a good job placement decision. They simply try to learn based on observations and affinity is just another dimension to which they pay attention.

Scheduling for heterogeneous systems [4, 5, 7] is an extremely active field with many researchers trying to find the best way to use disparate pieces of computing infrastructure. Our work builds on the ideas presented in this community by showing what kind of performance improvements one can expect to get over a realistic workload. Furthermore, we examine the impact of affinity scheduling on heterogeneous systems and show that at least for our workload, taking affinity effects into account can provide additional benefit but only if it is considered as a secondary effect in the context of heterogeneous scheduling.

Profiling applications in order to infer their performance on a particular machine type has received less attention than the problem of scheduling itself. Sodhi and Subhlok [25] discuss the use of skeletons in order to infer the performance of an application on a particular machine. This approach could greatly reduce the length of the data collection phase of our algorithm (especially if the number of different node types is large) and lead to more efficient online algorithms. Similarly Seltzer *et al* [8] discuss how performance of a particular computer can only be determined with respect to the application one is interested in running on it and present a methodology for collecting performance profiles of applications on different computers. Mutka and Livny [26] discuss the use of profiling on workstations. However their goal is not to deduce how appropriate a particular machine is for a certain job, but rather to discover which machines in a multi-use environment are available for batch job execution at any point in time.

7 Conclusions and Future Work

In this paper we have examined the performance impact of a number of scheduling algorithms on a real-world workload in the context of a heterogeneous machine cluster. We have found that simple greedy heuristics can offer tangible performance benefits over simple first-come first-serve scheduling. Furthermore, we have discovered that augmenting those heuristics to take affinity effects into account can further reduce average job completion times, even though pure affinity-based schedulers actually hurt rather than help performance in most cases. We have also shown that our heuristics need not know job profile information ahead of time. This information can be learned as part of the regular scheduling process. It is important however, to reduce the learning space by carefully deciding the machine characteristics and job arguments that uniquely identify distinct machine and job types.

The problem we have examined is not unique to Blast or the NCBI cluster. We are aware of other environments that run different application suites that have very sim-

ilar properties to the application suite examined in this paper. Such environments include proteomics discovery facilities which run an entirely different application suite on a variety of databases containing protein spectra, and computer animation enterprises that need to render scenes against a variety of lighting backdrops, obstacles and other artifacts, using a small number of different rendering algorithms. One important difference between those workloads and the one we examined is the presence of dependencies between jobs. Our workload contained purely independent jobs that could be run in any order, while the other workloads we are aware of, impose constraints on the order that jobs can be run. We also expect to see workloads where some jobs may need to be co-scheduled in order to achieve good performance. For example, we expect co-scheduling issues to arise for some of the latest version of Blast that splits databases in smaller overlapping chunks that can be searched in parallel. The partial results can then be stitched together in a final aggregation phase.

We are interested in further exploring this space in the context of real workloads, in order to better understand the impact of inter-job dependencies and co-scheduling constraints to heterogeneous scheduling. We are also interested in understanding how profile information can help guide cluster upgrade decisions with respect to the balance of CPU, memory, and I/O subsystem characteristics that new machines need to have in order achieve a good balance for a particular workload. Finally, we would very much like to see if we can design a system that can take into account all possible machine characteristics and all job arguments in defining its machine and job types, but can then use a small number of exeriments to quickly collapse the search space by determining which machine characteristics and which job arguments are actually relevant to application performance.

References

[1] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: Enabling scalable virtual organizations," *International Journal of Supercomputer Applications*, vol. 15, no. 3, 2001.

[2] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, "The physiology of the grid: An open grid services architecture for distributed systems integration," *The Open Grid Service Infrastructure Workgroup, Global Grid Forum*, June 2002.

[3] National Institute of Health, *National Center for Biotechnology Information*, 2004, http://www.ncbi.nlm.nih.gov/.

[4] M. Maheswaran, S. Ali, H. J. Siegel, D. A. Hensgen, and R. F. Freund, "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems," in *Heterogeneous Computing Workshop*, 1999, pp. 30–44.

[5] R. Freund, T. Kidd, D. Hensgen, and L. Moore, "Smartnet: a scheduling framework for heterogeneous computing," in *Second International Symposium in Parallel Architectures, Algorithms, and Networks*, June 1996, pp. 514–521.

[6] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "A basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, pp. 403–410, 1990.

[7] R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel, "Scheduling resources in multi-user, heterogeneous, computing environments with smartnet," in *Heterogeneous Computing Workshop*, 1998, pp. 184–199.

[8] M. Seltzer, D. Krinsky, K. Smith, and X. Zhang, "The case for application-specific benchmarking," in *The 1999 Workshop on Hot Topics on Operating Systems (HotOS VII)*, Rio Rico, AZ, March 1999.

[9] F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Robust, portable i/o scheduling with the disk mimic," in *Proceedings of the 2003 Usenix Annual Technical Conference*, San Antonio, TX, June 2003.

[10] J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker, "Complexity of machine scheduling problems," *Annals of Discrete Mathematics*, vol. 1, pp. 343–362.

[11] Jiri Sgall, "On-line scheduling," in *Online Algorithms, The State of the Art (the book grow out of a Dagstuhl Seminar, June 1996)*. 1998, pp. 196–231, Springer-Verlag.

[12] D. Shmoys, J. Wein, and D. Williamson, "Scheduling parallel machines on-line," *SIAM Journal of Computing*, vol. 24, pp. 1313–1331, 1995.

[13] Platform Inc., *Platform LSF*, 2004, http://www.platform.com/products/LSF/.

[14] Altair Grid Technologies, *The Portable Batch System*, 2004, http://www.openpbs.org/.

[15] The Condor Team, *The Condor Project*, 2004, http://www.cs.wisc.edu/condor/.

[16] Douglas Thain, Todd Tannenbaum, and Miron Livny, "Distributed computing in practice: The condor experience," *Concurrency and Computation: Practice and Experience*, 2004.

[17] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adpative load sharing in homogeneous distributed systems," *IEEE Transactions in Software Engineering*, vol. 12, no. 5, pp. 662–675, May 1986.

[18] Keith Ross and David Yao, "Optimal load balancing and scheduling in a distributed computer system," *Journal of the ACM*, vol. 38, no. 3, pp. 676–689, July 1991.

[19] H. Karatza, "A comparison of load sharing and job scheduling in an network of workstations," *Internation Journal of Simulation*, vol. 4, no. 3, 2003.

[20] Rajesh Raman, Miron Livny, and Marvin Solomon, "Matchmaking: Distributed resource management for high throughput computing," in *Proc. of the 7th IEEE International Symposium on High Performance Distributed Computing*, Chicago, IL, Jul 1998.

[21] David E. Culler Andrea C. Arpaci-Dusseau, "Extending proportional-share scheduling to a network of workstations," in *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, Las Vegas, NV, June 1997.

[22] S. Haldar and D. K. Subramanian, "An affinity-based dynamic load balancing protocol for distributed transaction processing systems," *Performance Evaluation*, vol. 17, no. 1, pp. 53–71, January 1993.

[23] H. Lee and T. Park, "Allocating data and workload among multiple servers in a local-area network," *Information Systems*, vol. 20, no. 3, pp. 261–269, May 1995.

[24] S. T. March and S. Rho, "Allocating data and operations to nodes in distributed database design," *IEEE Transactions on Knowledge and Data Engineering*, vol. 7, no. 2, pp. 305–317, April 1995.

[25] S. Sodhi and J. Subhlok, "Skeleton based performance prediction on shared networks," in *Proc. of the 2004 Grids and Advanced Networks Workshop (GAN04)*, Chicago, IL, April 2004.

[26] Matt Mutka and Miron Livny, "Profiling workstations' available capacity for remote execution," in *Performance '87, 12th International Federation Information Processing Working Group 7.3*, Brussels, December 1987, pp. 529–544.