Application Scheduling over Supercomputers: A Proposal

Walfredo Cirne and Fran Berman

University of California San Diego
Department of Computer Science and Engineering

1. Introduction

Performance is a very important aspect of computer systems. This has been true since the very birth of modern computers, where they were seen as calculators whose reason d'être was their ability to perform calculations faster than the human being. Performance is important to all areas of Computer Science, from Algorithms to Data Bases, from Operating Systems to Cryptography. Today, performance remains a major concern for both industry and academia.

Among the factors that affect performance, scheduling[1] is of fundamental importance. The execution time of an application strongly depends on the ordering of the pieces of work that form the application, which resources carry out each piece, and when such resources do so.

Scheduling has been traditionally done by the operating system [Bunt 76]. This is because the operating system controls the resources of interest to most applications (processors, memory, disks, etc). One salient characteristic of traditional scheduling is that the operating system receives requests from multiple users, and thus has to arbitrate among such users. This scenario leads to complex schedulers, that implement some arbitration scheme (e.g., real-time priorities, Unix priorities, ticket-based allocation) while trying to optimize for some system-wide performance goal (e.g., utilization, throughput). We call this kind of scheduler (i.e., one that control resources and arbitrate requests to such resources) a *resource scheduler*.

1.1. Metacomputing

High-speed networks enable the use of resources that are geographically distributed and even under control of many institutions, in what has been called *metacomputing* or *grid computing* [Foster 99b]. Metacomputing is a recent area. Therefore much of its early efforts went into building basic services to allow an application to run across widely distributed resources and administrative boundaries. Many of these services, such as communication [Bhat 98] [Fagg 97] [Foster 97], job submission [Czajkowski 98] [Karpovich 96], security [Ferrari 98] [Foster 98],

[1] By *scheduling* we mean "the assignment of work to resources within a specified time frame" [Berman 99].

and naming [Fitzgerald 97], are now in place. Of course, some additional services need to be developed to ease the task of building metacomputing applications (e.g., code distribution and automatic compilation across multiple platforms). Nevertheless, it is possible today to build fully functional metacomputing applications.

The research focus in metacomputing is thus currently shifting from being able to run an application to making it perform well. The metacomputer is much more heterogeneous and dynamic than the traditional parallel computation environment, in which applications run over dedicated partitions of very similar, if not identical, processors. Consequently, it is much harder to achieve good and consistent performance in the metacomputing scenario.

1.2. Metacomputing Scheduling

In the metacomputing scenario, having a single entity scheduling for the whole system raises severe technical and administrative problems. From the technical standpoint, if such a scheduler is centralized, it creates a bottleneck in terms of availability and performance. On the other hand, if this scheduler is distributed, keeping consistent state over wide-area networks has proven to be a problem [Babaoglu 97] [Previato 97]. The administrative problem is that such a scheduler would require multiple administrative domains to accept scheduling decisions from a single source, which would represent a lost of domain autonomy.

The only alternative is to have multiple schedulers in the metacomputer. Groups of resources are therefore independently controlled by different resource schedulers. Each of these schedulers independently arbitrates how multiple users access the resources it controls. Therefore, in order to use resources controlled by multiple schedulers, one needs to select the resources of interest, determine what piece of work is to be assigned to each to them, and then craft requests to their resource schedulers to have each piece of work carried out. This is the job of the *application scheduler*. Figure 1.1 depicts the relationship between the different kinds of schedulers in the metacomputing environment.

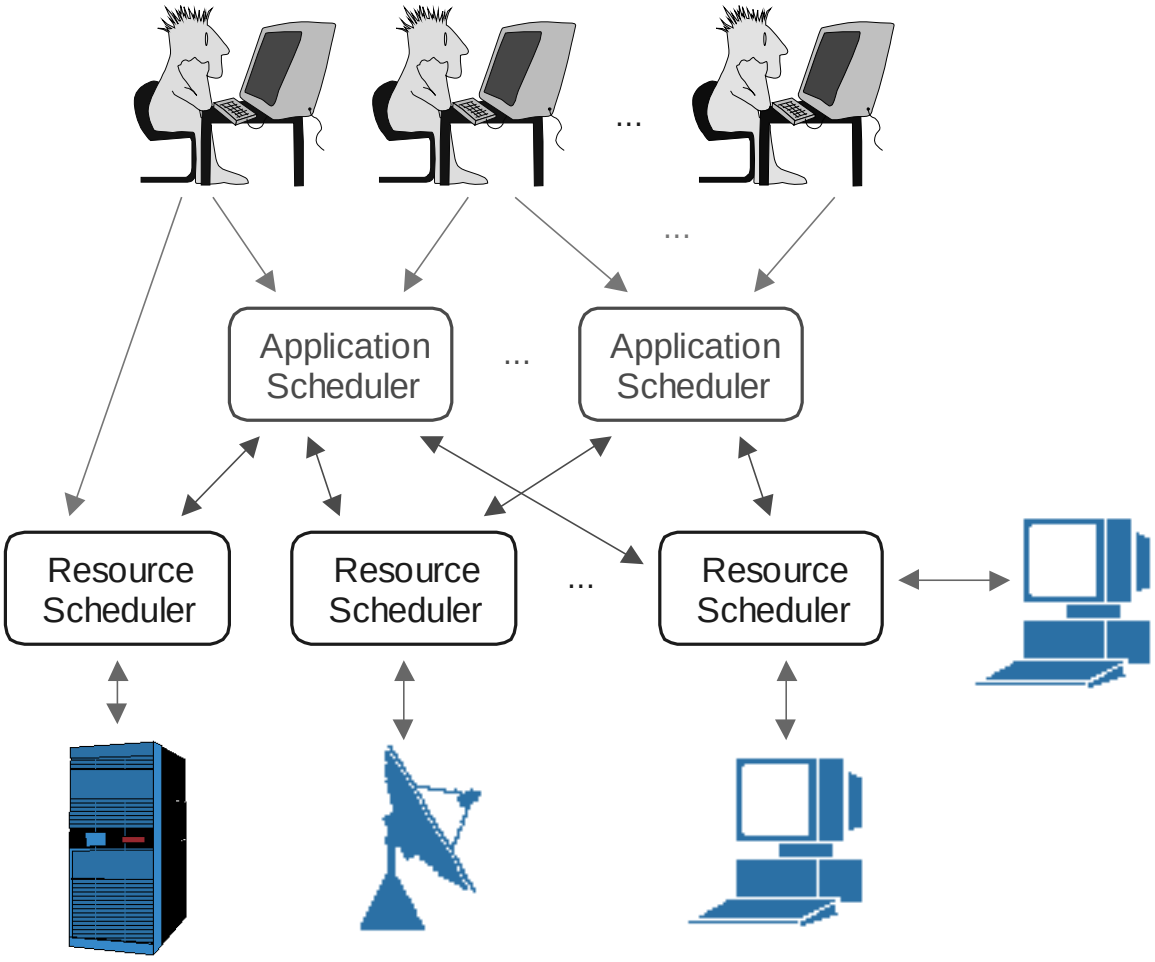

Figure 1.1 – The Schedulers Involved in Metacomputing Scheduling

Note that application schedulers do not control any resource. They get access to resources by submitting requests to the appropriate resource schedulers. This implies that application schedulers do not have to arbitrate among different users. Their goal is solely to improve the performance of the applications they serve. They can even limit themselves to schedule a single application. By targeting a single application (or a class of similar applications), application schedulers can rely on the application's structure and characteristics to come up with good schedules.

As an example of application schedulers, let's look at AppLeS (Applications-Level Schedulers) [Berman 96]. AppLeS are the application schedulers developed

by Fran Berman's group at UCSD and Rich Wolski's group at University of Tennessee [Berman 96] [Berman 97] [Cirne 99b] [Spring 98] [Su 99]. Figure 1.1 shows the general structure of an AppLeS. An AppLeS starts by obtaining information about the environment. The Network Weather Service (NWS) plays a fundamental role in this phase. After this, the AppLeS uses heuristics to select sets of resources to be evaluated. The *planner* then generates a schedule for each of these sets of resources. The control is then transferred to the *performance estimator*, which uses a performance model to evaluate the possible schedules. The *coordinator* finished the process by filtering unfeasible schedulers and commanding the implementation of the best remaining schedule via the *actuator*.

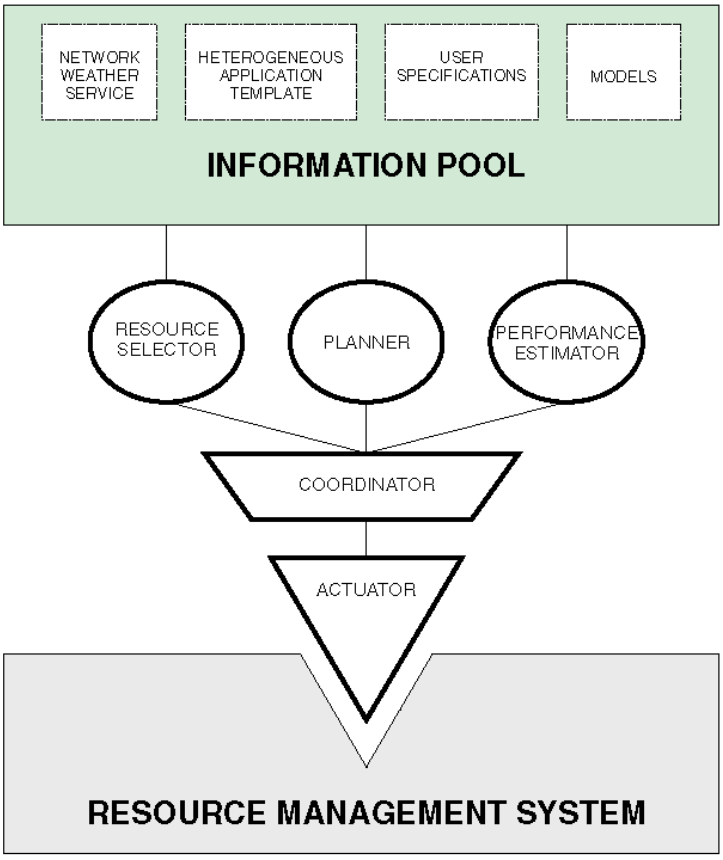

Figure 1.2 – The Structure of an AppLeS (from [Berman 96])

1.3. Challenges in Metacomputing Scheduling

Application scheduling is a key technology for obtaining good application performance out of metacomputing environments. Since it is not feasible to have all resources under the control of a single scheduler, one has to (i) select the resources to use, (ii) partition the work across the selected resources, and (iii) submit requests to the appropriated resource schedulers to have the selected resources carry out the work assigned to them.

Obtaining Information about Resource Schedulers

In order for the application scheduler to make good decisions, it needs to know how long a resource scheduler is going to take to process a given request. Obtaining such information is a major challenge because resource schedulers were not designed with this need in mind. This problem has been circumvented by *monitoring systems* that probe resource schedulers and forecast their availability, such as NWS [Wolski 98] [Wolski 99a], Komodo [Ranganathan 96], and Remos [Lowekamp 98]. The application scheduler is then able to figure out the execution time of a given request by combining resources' availability with applications' benchmarks, as in [Andersen 98], [Berman 96], [Casanova 96], [Shao 97], [Spring 98], [Su 99] and [Weissman 95].

The great advantage of this approach is that it enables application scheduling over unmodified, off-the-self resource schedulers. However, it seems that some resource schedulers are easier to forecast than others. For example, good results have been obtained on forecasting the behavior of the Unix time-sharing scheduler [Dinda 99] [Wolski 99b]. On the other hand, no one has been able to build monitoring systems for space-shared distributed-memory parallel computers. The efforts towards this end [Downey 97c] [Gibbons 97] [Smith 98] [Smith 99] have not delivered techniques that are accurate enough for application scheduling over such machines.

Easing the Development of Application Schedulers

Another important challenge in metacomputing scheduling is the very process of building application schedulers. Application schedulers are currently closely coupled to the applications they schedule. They are usually based on detailed performance models of the applications. Many of the proposed scheduling strategies rely on the application's characteristics. While this makes for really good schedulers, it also makes building an application scheduler an effort-intensive and

non-trivial task. In order for application schedulers to be widely deployed, much research is needed to ease the process of building them.

Bushel of AppLeS

Finally, the effect multiple application schedulers have on each other and on the system as whole, an issue that has been named *the Bushel of AppLeS problem* [Berman 97], is not clearly understood. This is indeed a very important issue because there is theoretical evidence that systems in which resource allocation is performed by many independent entities can exhibit performance degradation [Mitzenmacher 97] and even chaotic behavior [Hogg 91]. Sadly, it has been very difficult to investigate the Bushel of AppLeS question under realistic scenarios because application scheduling is only in its infancy. There aren't that many application schedulers in production use for any emergent behavior to have appeared in current systems.

2. Establishing The Focus

There is certainly a lot of research to be done in Metacomputing Scheduling. In order to achieve solid results, however, one has to focus on a well-defined set of issues. **In our case, we plan to focus in investigating how resource schedulers can better support application scheduling.**

The resource schedulers in current use were designed under the implicit assumption that they were the only schedulers in the system. This may make it difficult to use them as part of a metacomputing scheduling solution, in which multiple schedulers must co-exist. For example, most resource schedulers do not provide a priori information on how long a given request is going to take, which is a vital information for application schedulers. We would like to investigate the benefits of considering the metacomputing scenario in the design of resource schedulers. Our hope is that such an endeavor would result in a better infrastructure in which to build application schedulers.

Note that this goal requires something to be said about the Bushel of AppLeS problem. Introducing a new resource scheduler requires one to argue that it is going to promote the performance of the resources it controls. Since such a new scheduler would be designed to better support application scheduling, one would have to show that having many application schedulers in the system would not be a problem in and of itself.

2.1. Defining the Resource Model

Different kinds of resources require different resource scheduling approaches. For instance, the algorithms used for interactive, batch, and real-time scheduling vary quite a bit, as do the interfaces to submit requests to such resources. Establishing the focus of our research includes choosing what kind of resource we will be dealing with.

We intend to target distributed-memory parallel supercomputers[1]. They are particularly interesting as a target for this research because monitoring systems have failed to produce accurate enough predictions of their behavior. And without information on how requests are going to be processed, application scheduling is

[1] In this text, we use “supercomputer” as shorten for “distributed-memory parallel supercomputer”.

virtually impossible. Moreover, supercomputers are the fastest machines today available and therefore they would be a very important resource to be effectively added to the metacomputing environment.

Another reason to pick supercomputers is that we have preliminary evidence that application scheduling benefits from information that can be provided by the supercomputer scheduler. In fact, the PTomo AppLes [Cirne 99b] strongly relies on information (namely, which requests are going to start running immediately) provided by a supercomputer scheduler (namely, the Maui scheduler) to improve the application's performance. Although successful, such an AppLeS is limited in the requests it can generate. Since Maui provides information only about requests that start running immediately, the PTomo AppLeS can only produce such requests. We intend to design an interface that allows an application scheduler to obtain enough information to craft an arbitrary request to a supercomputer.

2.2. Characterizing Supercomputers

Since we are going to concentrate on supercomputers, it is important to characterize them in some detail. Such machines are composed of many processors, each with its own memory. Some supercomputers implement distributed shared-memory schemes (e.g., SGI Origin 2000). But this is only for the convenience of the application developer. Under the hood, shared-memory is implemented using message-passing (possibly followed by remapping memory pages to be local).

The processors are interconnected by very fast internal networking. Nevertheless, processes can be very large nowadays and thus context-switching a process from one processor to another has a considerable cost. This makes traditional time-sharing scheduling inadequate for these machines.

Applications receive a dedicated partition to run for a pre-established amount of time. Having a dedicated partition greatly simplifies work distribution concerns. It reduces work distribution to balancing the load across the processors. Although this often is a hard task by itself, it is surely easier then work distribution in a dynamically-changing non-dedicated heterogeneous environment.

Granting dedicated partitions imply that applications have to wait when there aren't resources available to make up the partition they ask for. Requests that cannot run immediately are *queued* until enough resources become available. Since

applications are not preempted, determining effective queue disciplines is the major focus in supercomputing scheduling [Feitelson 97a].

Therefore, in order to estimate the turn-around time of a request, one needs to predict the waiting time and the execution time for such a request. Since the application runs in a dedicated partition, the prediction of execution time can often be determined by benchmarking. However, queue waiting times have proven to be hard to predict. Indeed, the lack of good queue time predictions has been the major impediment in using supercomputers in the metacomputing environment.

2.3. Reassessing the Challenges

Focusing on supercomputers to investigate how resource schedulers can support application scheduling has two main advantages. First, it establishes a research arena that is narrow enough to allow for an extensive investigation. Second, it is something useful by itself, due to the current difficulties with queue time predictions. However, we also need to consider how singling supercomputers out affect our study.

Obtaining Information about Resource Schedulers

A key aspect in supporting application scheduling consists of enhancing the predictability of the resource scheduler. In order to keep our model as simple as possible, we are not assuming that some gang-scheduling scheme is available. Therefore, we cannot use preemption, what makes it harder to implement a predictable schedule. However, the previous work we have done in the Computational Co-op suggests that, even when preemption is not an option, it is possible to be enough predictable to support application scheduling [Cirne 99a].

Bushel of AppLeS

The Bushel of AppLeS problem seems to be more critical when parallel applications are involved because they are not “work-conservative” across partition sizes. That is, for most parallel applications, the amount of computational work they require varies depending on the size of the partition. In fact, since speed-up is often sublinear, the computational work generally grows with the partition size. This represents a potential problem because if each application scheduler individually chooses a large cluster size, the amount of work for the system as a whole increases, which is likely to affect everybody’s performance.

3. Related Work

Any research effort should start with a comprehensive survey of the literature. Towards this end, we have identified four areas that are strongly related to our research agenda. They are application scheduling, systems with independent decision-makers, supercomputing scheduling, and predictability in resource scheduling.

Application Scheduling

There has been a great deal of interest in application scheduling in recent years. As with any evolving area, there seems to be reasonable agreement on some aspects of application scheduling, but not on others. Perhaps the point that is most commonly agreed upon is that application scheduling demands good information about the system. In particular, using the last measured value of the system state doesn't seem to be enough [Berman 96] [Casanova 96] [Lowekamp 98] [Ranganathan 96] [Shao 97] [Spring 98] [Su 99] [Weissman 95] [Weissman 98] [Zhu 98]. This observation is the main motivation behind monitoring systems [Wolski 98] [Wolski 99a] [Ranganathan 96] [Lowekamp 98].

Although sometimes it is possible to formulate a scheduling problem in a way that can be solved in polynomial time (e.g., [Amoroso 98]), most instances of scheduling are NP-Hard problems. Therefore, most application schedulers use heuristics to navigate through the space of possible schedules. The main components of such heuristics are (i) how two schedules are compared, and (ii) how the space of schedules is traversed.

Most application schedulers use a performance model to compare two possible schedules [Amoroso 98] [Andersen 98] [Berman 96] [Casanova 96] [Ranganathan 96] [Shao 97] [Su 99] [Weissman 95]. Others have devised a mechanism to rank schedules without actually estimating the application's performance [Lowekamp 98] [Zhu 98]. Since application schedulers that rely on performance models provide an estimate on the application's execution time, they can be more easily used as a component of another application scheduler. This ability makes for compositional and scalable solutions and hence is important for large systems [Weissman 98]. On the other hand, performance models are hard to build. It might be that ranking-based application schedulers are easier to deploy because they don't require detailed knowledge about the application structure.

There are situations in which the space of possible schedules is small. For example, selecting the best server from among a small number of possibilities. In these cases, application schedulers can simply perform an exhaustive search [Andersen 98] [Casanova 96] [Ranganathan 96] [Su 99] [Zhu 98]. When the number of possible schedules is non-trivial, heuristics are used to search such space. Sometimes a polynomial-time optimal solution for part of the problem is used as a component of the heuristic. For example, there are application schedulers that heuristically select the resources to be used, and then perform the work distribution via time balancing [Berman 96] [Shao 97] [Weissman 95].

Systems with Independent Decision-Makers

The Bushel of AppLeS phenomenon can be seen as a particular case of systems in which multiple independent agents make decisions: In the Bushel of AppLes problem, such decisions are limited to be scheduling decisions. Due to the relatively little literature on the Bushel of AppLeS problem per se, we have decided to investigate the more general area of systems with independent decision-makers.

An area in which systems with independent decision-makers are commonplace is economics. Therefore we expect some of the economics literature to be useful in dealing with the Bushel of AppLeS problem. For example, economic regulations can be seen as mechanisms to control systemic problems by reducing the freedom of the decision-makers [Bos 94].

As a matter of fact, there has been research on how economic principles can be used to provide innovative solutions for computer science problems [Clearwater 96] [Cheng 98] [Harty 96] [Mullen 96] [Stonebraker 96] [Tucker 96] [Waldspurger 92]. Of particular interest to us are those papers that, assuming the agents to implement a particular strategy, address emergent characteristics of the system as a whole, such as convergence and stability [Walsh 98] [Walsh 99]. Computational markets seem to be a natural scenario to explore the stability and performance of systems with multiple independent decision-makers.

Although small, there is some literature on the Bushel of AppLeS problem. Some fundamental work has already been done by [Hogg 91] and [Mitzenmacher 97]. They highlight the importance of diversity for the stability and performance of systems with independent decision-makers. Intuitively, when all decision-makers employ very similar strategies, there is a greater the chance for “herd behavior” to happen. Diverse systems are in general more robust, a fact that is starting to be explored in computer science (e.g., [Forrest 97]).

Supercomputer Schedulers

There are a handful of supercomputer schedulers currently in use, including Easy [Lifka 95], PBS [Henderson 95], Maui [Maui 99], and LSF [Platform 99]. Unfortunately, details about their scheduling algorithms are often not available. Even worse, most of these schedulers radically change their behavior depending on their configuration, which makes characterizing them a very complex task. There are also numerous simulation-based studies on queue disciplines for supercomputer schedulers. For a nice survey on the area, we refer the reader to [Feitelson 97a].

From our research perspective, the most interesting results are those that deal with:

i) The predictability of supercomputer schedulers [Downey 97c] [Feitelson 98] [Gibbons 97] [Smith 98] [Smith 99],

ii) The impact the accuracy of the requests has on scheduling [Feitelson 98] [Zotkin 99], and

iii) Modeling supercomputers' workloads [Downey 97b] [Downey 99] [Feitelson 97b] [Jann 97].

Predictability in Resource Scheduling

Within the operating systems community, there has been considerable effort to make resource scheduling more predictable [Fong 95] [Stoica 96] [Waldspurger 94] [Waldspurger 95]. However, many of these efforts primarily aim to provide a fine level of control on how the resources are shared among their users. Predictability comes from the fine-grained implementation of the policy that determines how resources are to be shared.

In a smaller scale, some researchers have started exploring the predictability of resource scheduling as a way to enable multiple schedulers to coexist [Chapin 95] [Cirne 99a] [Harty 96]. These results are naturally more metacomputing oriented. The focus here is on where to draw the line dividing the responsibility of resource and application schedulers, and what interface should one export to the other. Predictability appears as a requisite for application scheduling.

Very closely related to the work we intend to do, there has been considerable interest in enhancing supercomputer schedulers to provide *reservations* [Foster 99a]. These efforts address the need for resource schedulers to be predictable in the metacomputing environment. However, reservation is not a complete solution. One also needs information that empower application schedulers in dis-

covering which reservation ask for. A trial-and-error strategy (as suggested in [Foster 99a]) would likely result in slow and poor application scheduling.

4. An Initial Solution

In order to provide some evidence that our research agenda is promising, we have been exploring how supercomputer schedulers can be made more metacomputing-friendly. This effort has started with the identification of resource scheduler features that benefit the performance of application schedulers. We then built *S*^{3} (Space-Shared Scheduler) to provide such features, and an AppLeS to use them. S^3 implements a strategy called *conservative backfilling* [Feitelson 98] and provides unique support for application schedulers that target supercomputers.

4.1. Design Goals

A resource scheduler with good support for application scheduling still has to perform well according to its traditional metrics, otherwise no administrator will install it. Therefore, our design has to reconcile promoting resource performance with supporting application scheduling.

Supporting Application Scheduling

It seems that application scheduling depends on two closely related properties of the underlying environment: (i) the predictability of requests' service times, and (ii) the availability of information useful for application scheduling.

With unpredictable requests, any application schedule can go wrong. This comes with no surprises. Application schedulers rely upon the expected service time of the requests that form the schedule. When real service times largely differ from the expected ones, the schedule can perform poorly. The more predictable the requests, the better the application schedule. In particular, there has to be a minimum of predictability in the system for application schedulers to do any good.

But, in general, predictability is not enough by itself. Additionally, application schedulers need a good way to search large spaces of possible schedules. For example, consider the original AppLeS, which targets Jacobi 2D applications running over independent workstations [Berman 96]. Jacobi 2D is a data-parallel application in which each workstation is assigned a partition (strip) of the data set. The AppLeS decides on the size of each partition. It does so by using predictions of CPU and network availability to model the performance of each workstation. More precisely, the performance of each workstation is estimated by an equation whose

free variable is the size of the partition allocated to that workstation. Good performance is achieved when all workstations finish simultaneously. To accomplish this, the AppLeS equates all equations and solves the resulting system in order determine the size of the partition to be allocated to each workstation. The aspect we highlight here is that the Jacobi 2D AppLeS depends upon a particular kind of information (namely, predictions of CPU and network availability) to schedule the application without exhaustively evaluating of all possible schedules.

There is a close relation between request predictability and information that is useful for application scheduling. Such kind of information *embeds* predictions. To continue with the same example, CPU and network availability can be used (together with benchmark data) to estimate how long a given processor would take to process a certain area. On the other hand, predictions of this kind by themselves are not enough for producing a good schedule in a reasonable amount of time.

Keeping Good Resource Performance

Of course, supporting application scheduling should not be the sole goal of resource schedulers. Resource-related metrics are, if not more, at least as important. For example, by releasing applications in a first-come-first-served (FCFS) basis, one can make resource scheduling very predictable, but with a huge penalty in the machine's utilization and throughput.

4.2. Fixing the Target Environment

Even though distributed-memory parallel computers share many commonalities, they can still differ in some aspects. Likewise, not all parallel applications present the same characteristics; nor do all schedulers provide the same functionality. It is therefore important to clearly state what features a machine is assumed to have, what parallel applications are allowed to do, and which functionality S^3 supports.

Supercomputer Characteristics

S^3 assumes the supercomputer to:

i) Not have a topological constraint in processor allocation. That is, any subset of processors can make up an allocable partition.

ii) Be homogeneous (i.e., be formed of identical processors).

Nevertheless, a heterogeneous supercomputer could be targeted by a two-level scheduling architecture based on S^3. The heterogeneous

supercomputer could be divided into homogeneous pieces, each independently controlled by an instance of S^3. A submission interface for the whole system could be provided by a generic application scheduler. Of course, regular application-specific AppLeS could also be used.

Parallel Application Model

As with current supercomputer schedulers, applications request a certain number of processors for a given amount of time from S^3. Applications that run beyond the time they've requested are killed. There are two basic implications of this requirement:

i) Applications have to be able to produce an upper-bound of their execution time. There is incentive in providing good estimates. A tighter upper-bound will likely make the application to start executing sooner.

ii) Applications cannot change the size of their partition in the middle of the execution. As a matter of fact, it would be relatively simple to allow applications to *release* processors. But since most of the programming infrastructure currently used (e.g. MPI 1.1 [MPI 95]) do not support changes in the partition size, we will not implement releasing part of the processors initially.

S^3 Functionality

S^3 exports a complete interface to promote the use of a supercomputer in the metacomputing environment. There are calls for *submission*, *reservation*, *cancellation*, and *status inquiry* (see details in the next Section). However, currently no provision is made for priorities. All applications are treated equally.

4.3. S^3 – Space-Shared Scheduler

The heart of S^3 is the conservative backfilling algorithm [Feitelson 98]. Conservative backfilling uses an allocation list that keeps, for any given time, which processors are already committed to which applications. The allocation list can be implemented as a linked list whose nodes represent time frames in which all processors in system are allocated in the same way. Arriving applications are put in the first "slot" they fit. For example, Figure 4.1 depicts the submission of five requests in the following order: *A*, *B*, *C*, *D*, and *E*. Note that *C* is placed before *B* because, at time t_1, the available resources cannot fulfill *B*, but they are enough for *C*.

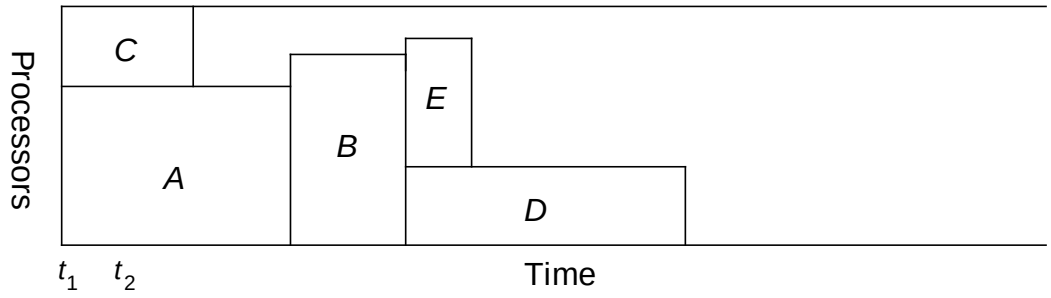

Figure 4.1 – Allocation list after the submission of five requests

Whenever an application finishes using less time than it allocated, conservative backfilling traverses the queue (in submission order) and "promotes" the first application that fits in the just-made-available slot. Of course, this may create another available slot that is backfilled in the same way. The process stops only when no more backfilling can be done. For example, Figure 4.2 shows what happens when *A* finishes at t_2: *B* is backfilled to start immediately after *C*, *D* "follows" *B*, and *E* can finish before *B* starts and thus is backfilled all the way to start running immediately.

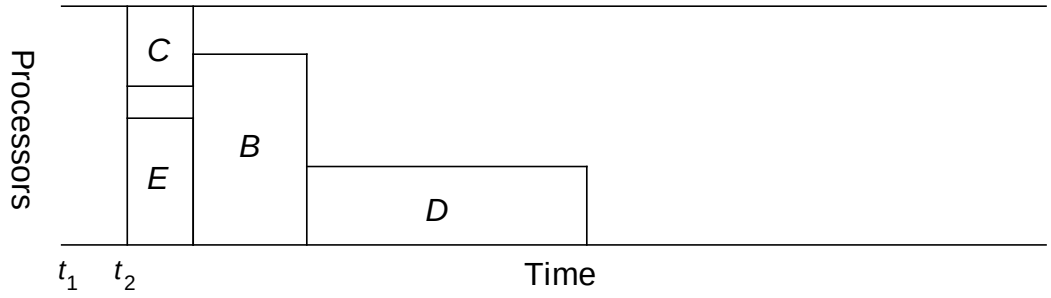

Figure 4.2 – Allocation list after the backfilling initiated by A *finishing at* t_2

The reasons for selecting conservative backfilling for S^3 are:

i) First and foremost, conservative backfilling makes it possible to export the system's *availability list*. Such a list contains the number of free processors per time frame. For example: [(from 0, to 10, 15 processors), (from 10, to 50, 80 processors), (from 50, to 210, 0 processors), (from 210, to 300, 30 processors), (from 301, to ∞, 128 processors)].

ii) Previous research has shown conservative backfilling to be a good scheduler. More precisely, [Feitelson 98] compared conservative backfilling against the widely used Easy scheduler [Lifka 95]. While both schedulers are comparable in terms of utilization, conservative backfilling provides guarantees on the start time of jobs, whereas Easy doesn't.

iii) At least Maui and PBS can be configured to use conservative backfilling, which makes it practical.

iv) Conservative backfilling is naturally reservation-ready. In fact, it implicitly creates a reservation for each application submitted to the system. That's a key feature because reservations are also very important to support metacomputing.

The idea is that application schedulers can use the availability list as the basis of their decision. For example, an application scheduler can gather the availability lists from multiple supercomputers in order to select the one that first finish the application. Going further, the application scheduler can consider partitioning the application and co-allocating its pieces across multiple supercomputers. Like any information that supports application scheduling, the availability list informs the application scheduler about what is going to happen to a given request.

4.4. Gas – Generic AppLeS for Supercomputers

The availability list can be used even when only one supercomputer is available. Applications are usually able to run over partitions of different sizes. Large partitions generally give better execution time than small ones, but requests for large partitions might wait longer in queue. Therefore, the request that will deliver the best turn-around time depends upon the current load of the supercomputer. And this is exactly the kind of decision the availability list is to enable the application scheduler to make.

We have built an AppLes for this scenario, and named it *Gas* (Generic AppLeS for Supercomputers). Gas' goal is to improve the application's turn-around time by tailoring the request to be sent to the supercomputer. In other words, by properly choosing the partition size and the maximum execution time based on the availability list, Gas seeks to reduce the application's turn-around time.

Gas uses a very straightforward strategy. All it needs is a performance model of how much time the application will take to execute over partitions of dif-

ferent sizes. It traverses the availability list evaluating possible requests and then picks the best. The algorithm is:

```
# Gas pseudo-code
for each time frame f in the availability list
    let f = (s, e, n), where s is the start of the time frame,
                              e is its end, and
                              n is the number of processors available
    # walk on the availability list to determine d, the last moment until
    # which we can allocate n processors, starting from s
    let d = e
    let g be the time frame succeeding f
    let g = (s_g, e_g, n_e)
    while n_e ≥ n
        let d = e_g
        let g be the time frame succeeding g
    #
    if the application can run on n processors in time (d - s)
          consider this request a candidate
choose the candidate request with the least turn-around time
```

Figure 4.3 – Gas pseudo-code

For example, assume that Gas is scheduling an application that can run over 10, 20, or 30 processors. It needs 5 units of time when using 10 processor, 3 with 20 processors, and 2 with 30 processors. Assume that the availability list is [(from 0, to 1, 5 processors), (from 1, to 5, 10 processors), (from 5, to 6, 0 processors), (from 6, to 7, 10 processors), (from 7, to 11, 20 processors), (from 11, to ∞, 40 processors)], as graphically shown by Figure 4.4. In this case, Gas finds three candidate schedules: (*A*) 10 processors starting at 6 and finishing at 11 (Figure 4.5), (*B*) 20 processors starting at 7 and finishing at 10 (Figure 4.6), and (*C*) 30 processors, starting at 12 and finishing at 14 (Figure 4.7). Since *B* is expected to finish earlier, Gas submits to S^3 a request asking for 20 processors and 3 units of time.

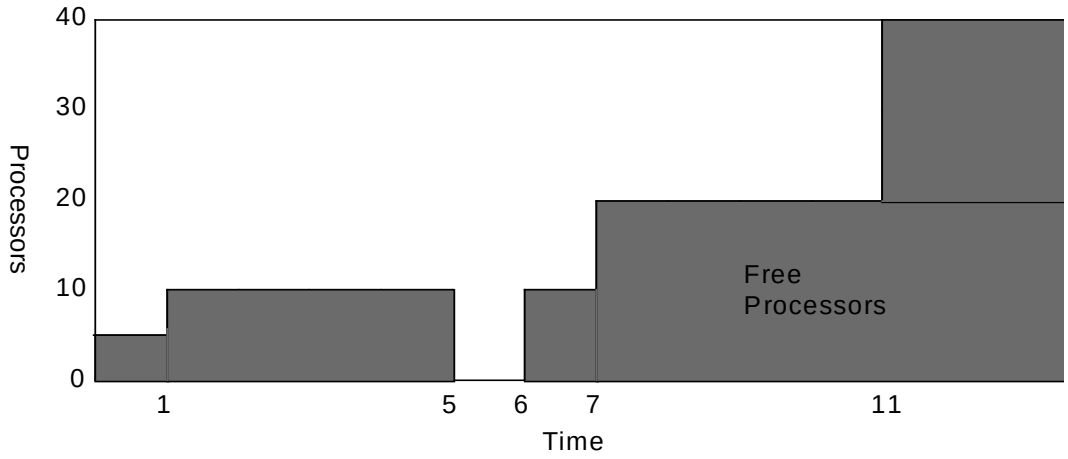

Figure 4.4 – Gas example: allocation list

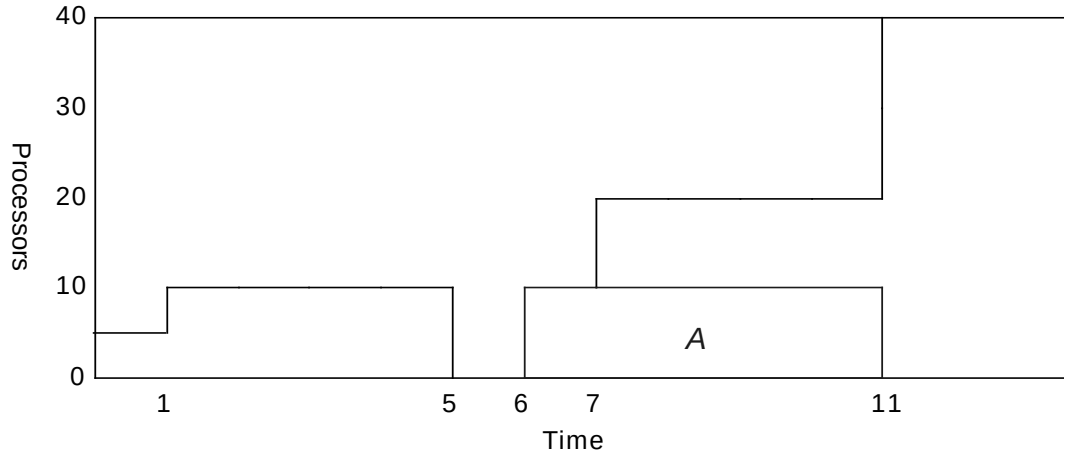

Figure 4.5 – Gas example: schedule A

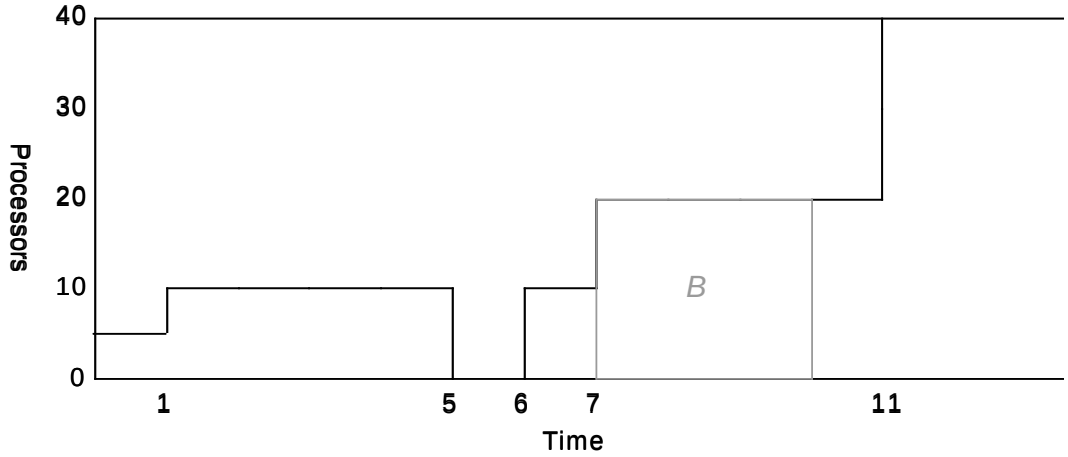

Figure 4.6 – Gas example: schedule B

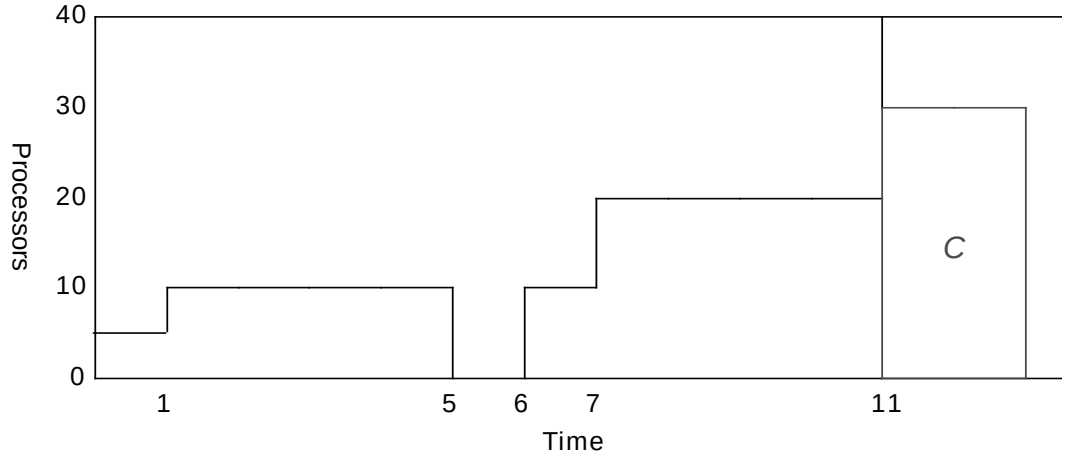

Figure 4.7 – Gas example: schedule C

Note that Gas does not use any sophisticated scheme to estimate the execution time of the applications that are in the queue. It schedules as if all the applications in the queue would take all the time they've requested. Nevertheless, it does much better than the traditional, non-adaptive approach (as we show in the next Section). This is not to say, however, that it cannot be improved by using some execution time forecast technique. This is something to be explored.

5. Initial Results

We have implemented both S^3 and Gas. However, their evaluation in a production environment would involve replacing the resource scheduler of some supercomputer, a hard-to-sell proposition, especially when the goal is to *evaluate* a new idea. We have addressed this difficulty by running S^3 and Gas in event-driven simulations. Hopefully the good results thereby obtained are going to help to sell the installation of S^3 in a production supercomputer.

5.1. The Experimental Set-up

We are *not* simulating S^3 and Gas per se. In fact, we are using the *real* S^3 and Gas. We simulate the components surrounding our schedulers: the users and the supercomputer. More precisely, we generate the requests that are to be fulfilled by S^3 and Gas, and simulate the supercomputer on which S^3 "executes" applications. Figure 5.1 portrays the simulation scenario.

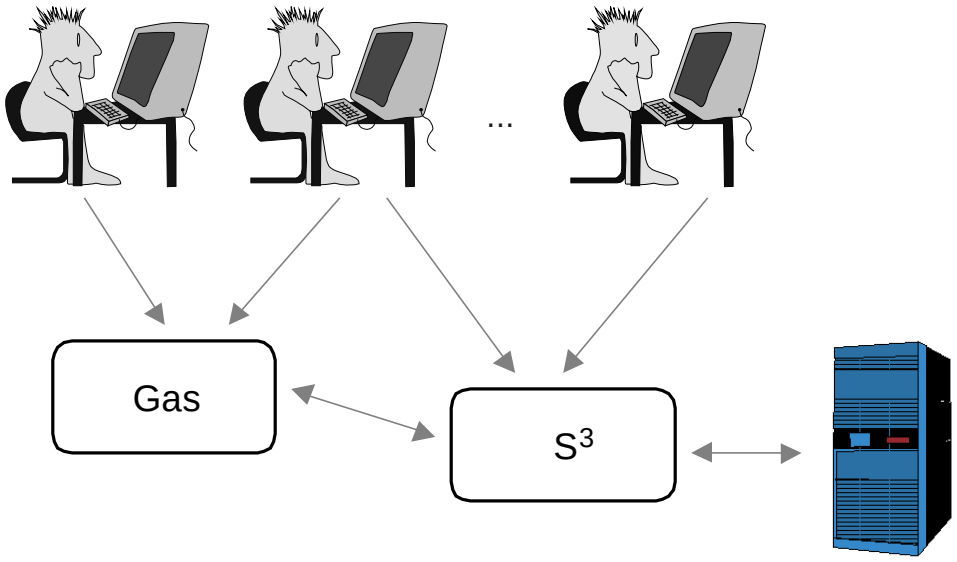

Figure 5.1 – Simulation environment used for the evaluation of S^3 and Gas. The users and the supercomputer are simulated. S^3 and Gas are the real code.

For our purposes, simulating a supercomputer is a straightforward task. All we need is the execution time of each application. When S^3 starts an application in

the supercomputer, the simulator uses its execution time to calculate when it finishes. When the simulation time reaches the moment an application finishes, the simulator informs S^3 that the application is done (and hence that the processors it occupied are now free).

Unfortunately, simulating the request stream generated by the users is not that easy. In fact, it is a very hard task [Chapin 99] [Downey 99]. With this in mind, we've opted to use the submission log of real supercomputers in our simulations. In particular, we use the following submission logs: San Diego Supercomputer Center (SDSC) SP2, 128 nodes, January to April 1999; Cornell Theory Center (CTC) SP2, 430 nodes, July 1996 to May 1997; Swedish Royal Institute of Technology (KTH) SP2, 100 nodes, October 1996 to August 1997.

Of course we cannot use any log "as is" because they do not contain the applications' speed-up behavior. We circumvent this problem by *replacing* applications in the log by applications with known speed-up behavior. This preserves the inter-arrival time of the original workload. Moreover, we search for applications whose consumed CPU time is as close as possible to the applications we introduce into the workload. The idea is to alter the aggregated load as little as possible.

Now, how do we figure out the speed-up function of the applications we introduce into the workload? We are using NAS benchmarks for our substitute applications. The good thing about NAS benchmarks is the availability of their execution times over a variety of supercomputers and partition sizes. Such data is publically available at http://science.nas.nasa.gov/Software/NPB/. Moreover, since they are used to evaluate performance, they are representative of real workloads. Finally, some of the NAS benchmarks are constrained with respect to the number of processors they can use. For some, such a number has to be a perfect square. For others, it has to be a power-of-two. This represents another real-world constraint for Gas.

For the record, we are using five NAS benchmarks: MG, LU, SP, BT, and EP. MG and LU require a power-of-two partition size and thus are the most constrained applications. SP and BT require perfect-square partition size. There are no restrictions for EP. It can run over any number of processors. For each benchmark, there are two predefined inputs, called B and C. C inputs are greater than B ones and thus NAS benchmarks take longer to run with an input of type C. Further details can be found in http://science.nas.nasa.gov/Software/NPB/.

5.2. AppLeS Power

First, let's see how Gas compares to submitting requests directly to S^3, which is the current modus operandi. In order to increase the number of scenarios, we split the three workloads into one-month workloads, getting therefore 22 different workloads. For each workload, we introduced *one* NAS benchmark. The substitution of one application in a month-long log does not significantly affect its aggregated load. The benchmarks are randomly chosen from the following distribution: 1/3 requires power-of-two partition sizes (MG and LU), 1/3 requires perfect-square partition sizes (SP and BT), and 1/3 are not restricted (EP). The class of the input (B or C) is randomly chosen from a uniform distribution.

We then compared the turn-around time of the introduced NAS benchmark when Gas generates the request against the one obtained with a randomly picked static value for the number of processors. Taking care to always select a different application to replace, we repeated this experiment more than 200 times for each one-month workload. In total, we conducted this experiment 4579 times.

The results are very encouraging. They show the potential of the AppLeS approach in reducing the application turn-around time, which is a key performance metric from the user standpoint. The following statistics summarize the turn-around time of the 4579 experiments we conducted:

Turn-Around Time (One AppLeS in the System)		
	Gas	Static
mean	2739.90	15174.25
standard deviation	7261.54	30955.29
median	136	935
minimum	2	2
maximum	84369	256658

Table 5.1 – Turn-around time of Gas and static requests. One Gas request in the system.

Not only does Gas show an amazing improvement of the mean turn-around time; it also reduced its variance. We conjecture this is because Gas was able to adapt to the current state of the supercomputer and thus deliver a more consistent turn-around time.

However, we also need to point out that Gas does worse than static 4.6% of the time. We believe this is because it doesn't know (or try to predict) the execution times of the applications already in the queue. It uses only the total time they've requested. Nevertheless, even such a simple approach does very well on average and most of the time. Moreover, the best static request represents an improvement of 17.6 times over the corresponding Gas, while the best Gas request improves over static by 43794 times!

Of course, these statistics simply summarize the results. In order to provide a more complete description of such results, below we present the distribution of improvements attained by Gas. The improvement factor denotes how many times the Gas improved on static (or the other way around, when the value is negative).

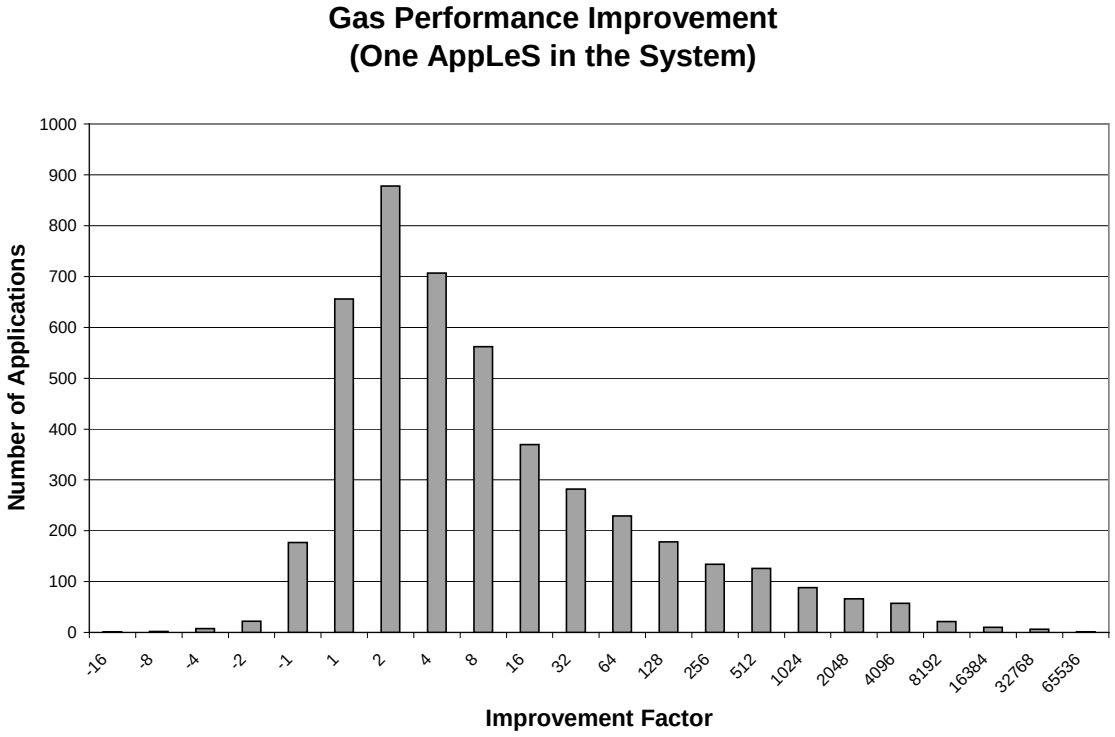

Figure 5.2 – The improvement factor obtained by Gas.
One Gas request in the system.

Separating the Results by Supercomputer

A natural question to ask is whether the performance improvements were similar across the three different workloads. We therefore separated the experiments based on which submission log was used. The statistics summarizing the

turn-around time and the distribution of the improvement obtained by Gas on each supercomputer are:

Turn-Around Time (One AppLeS in the System; SDSC workload)		
	Gas	Static
mean	4808.39	39722.34
standard deviation	9920.45	52527.31
median	431	15793
minimum	2	2
maximum	79743	256658

Table 5.2 – Turn-around time of Gas and static requests. Simulation based on the SDSC workload. One Gas request in the system.

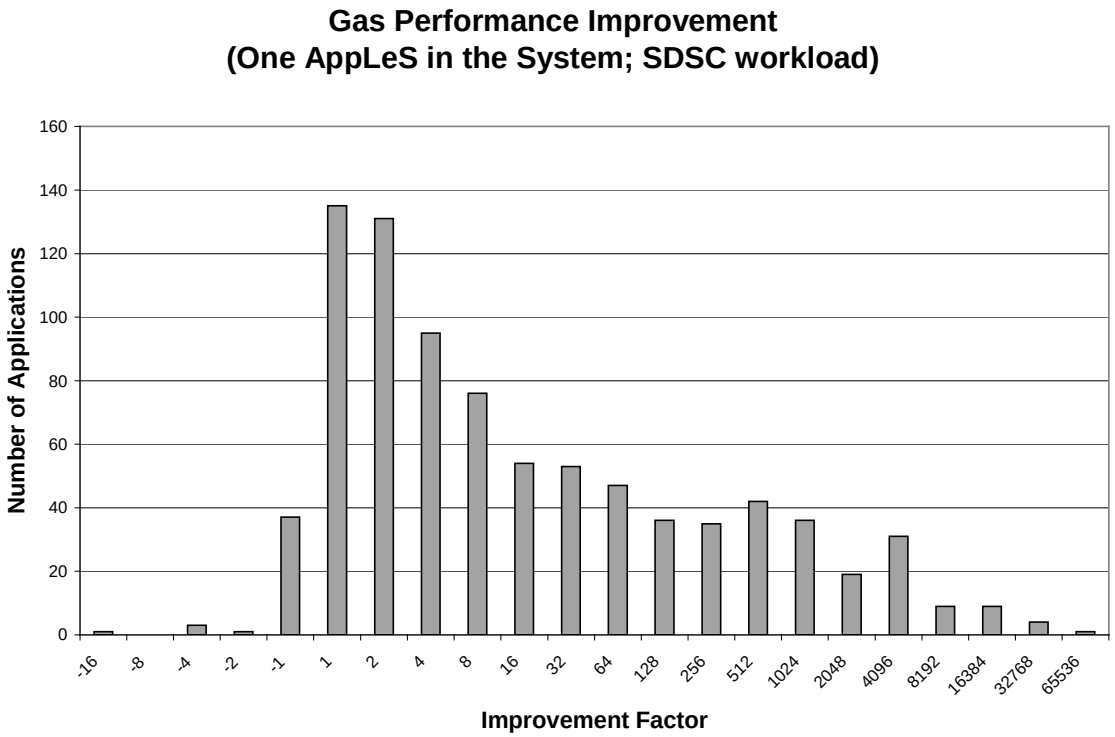

Figure 5.3 – The improvement factor obtained by Gas. Simulation based on the SDSC workload. One Gas request in the system.

Turn-Around Time (One AppLeS in the System; CTC workload)		
	Gas	Static
mean	869.48	5164.76
standard deviation	2877.85	11833.55
median	44	255
minimum	2	2
maximum	34379	98405

Table 5.3 – Turn-around time of Gas and static requests. Simulation based on the CTC workload. One Gas request in the system.

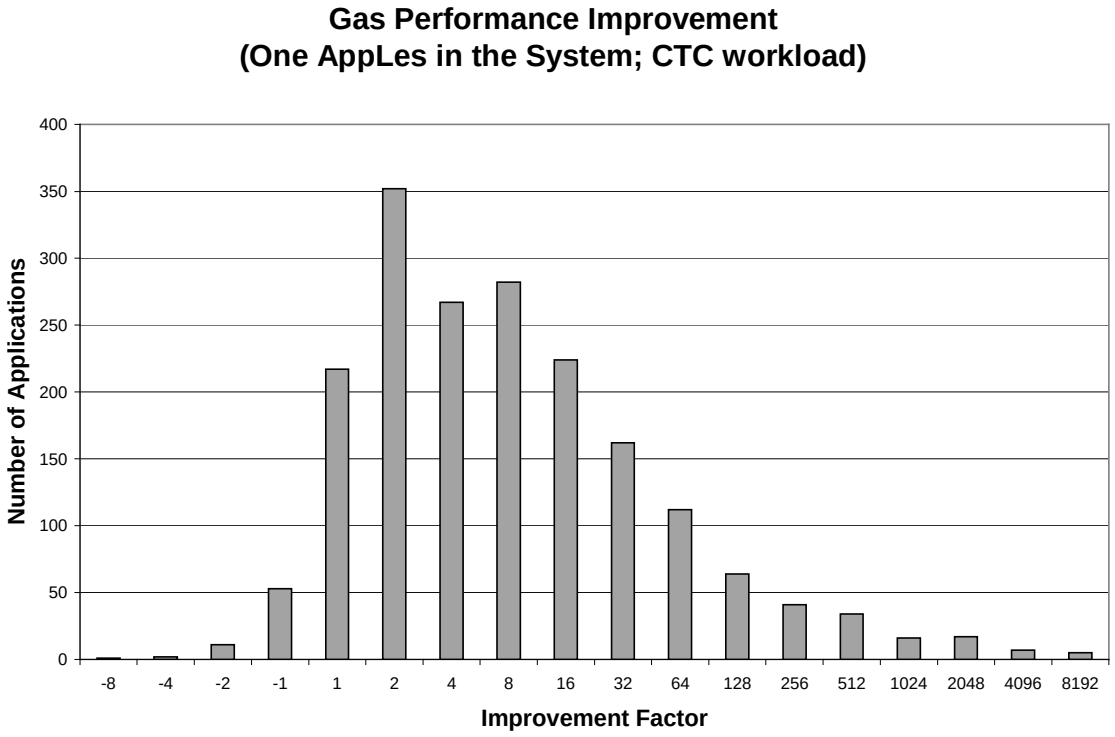

Figure 5.4 – The improvement factor obtained by Gas. Simulation based on the CTC workload. One Gas request in the system.

Turn-Around Time (One AppLeS in the System; KTH workload)		
	Gas	Static
mean	3668.02	13935.20
standard deviation	8372.64	23943.78
median	252	3059
minimum	3	3
maximum	84369	189060

Table 5.4 – Turn-around time of Gas and static requests. Simulation based on the KTH workload. One Gas request in the system.

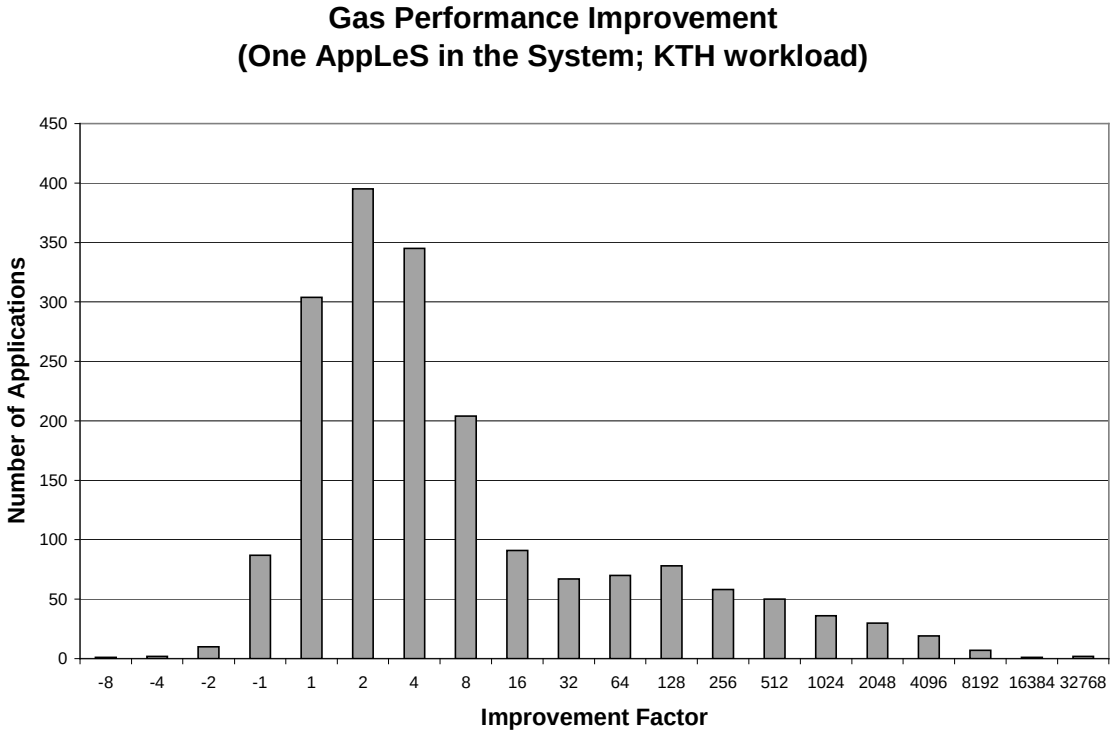

Figure 5.5 – The improvement factor obtained by Gas. Simulation based on the KTH workload. One Gas request in the system.

As can be seen, the application's turn-around time vary substantially from one supercomputer to another (for both Gas and static requests). We believe this is due to the differences in the capabilities of the supercomputers and the aggregated load submitted to each of them.

However, even under these different conditions, Gas was always able to significantly improve the application's turn-around time. Also, Gas reduced the variance of the turn-around times across all three supercomputers. Finally, the distributions of improvements are all of similar shape. These observations suggest that Gas can deliver good application performance in a variety of conditions.

Separating the Results by Benchmark

Another question that immediately follows from these results is whether the restrictions on the application's partition size impacts on the performance that Gas can deliver. In order to investigate this issue, we have separated the experiments in which the NAS benchmark requires power-of-two number of nodes (LU and MG)…

Turn-Around Time (One AppLeS in the System; LU and MG Benchs)		
	Gas	Static
mean	3222.74	15552.51
standard deviation	7930.61	31704.97
median	136	704
minimum	2	2
maximum	78945	256658

Table 5.5 – Turn-around time of Gas and static requests for LU and MG NAS benchmarks. One Gas request in the system.

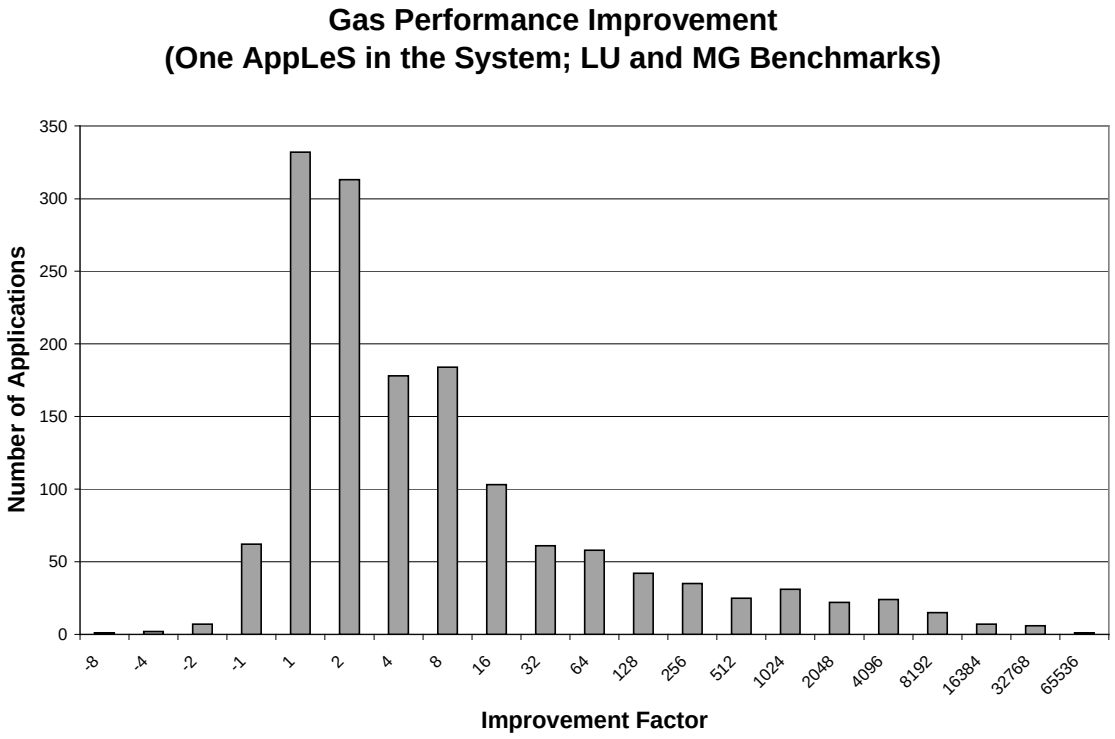

Figure 5.6 – The improvement factor obtained by Gas for LU and MG NAS benchmarks. One Gas request in the system.

… those that require prefect-square nodes (SP and BT) …

Turn-Around Time (One AppLeS in the System; SP and BT Benchs)		
	Gas	Static
mean	3988.52	15056.68
standard deviation	8859.51	30170.12
median	352	1347
minimum	26	44
maximum	84369	245020

Table 5.6 – Turn-around time of Gas and static requests for SP and BT NAS benchmarks. One Gas request in the system.

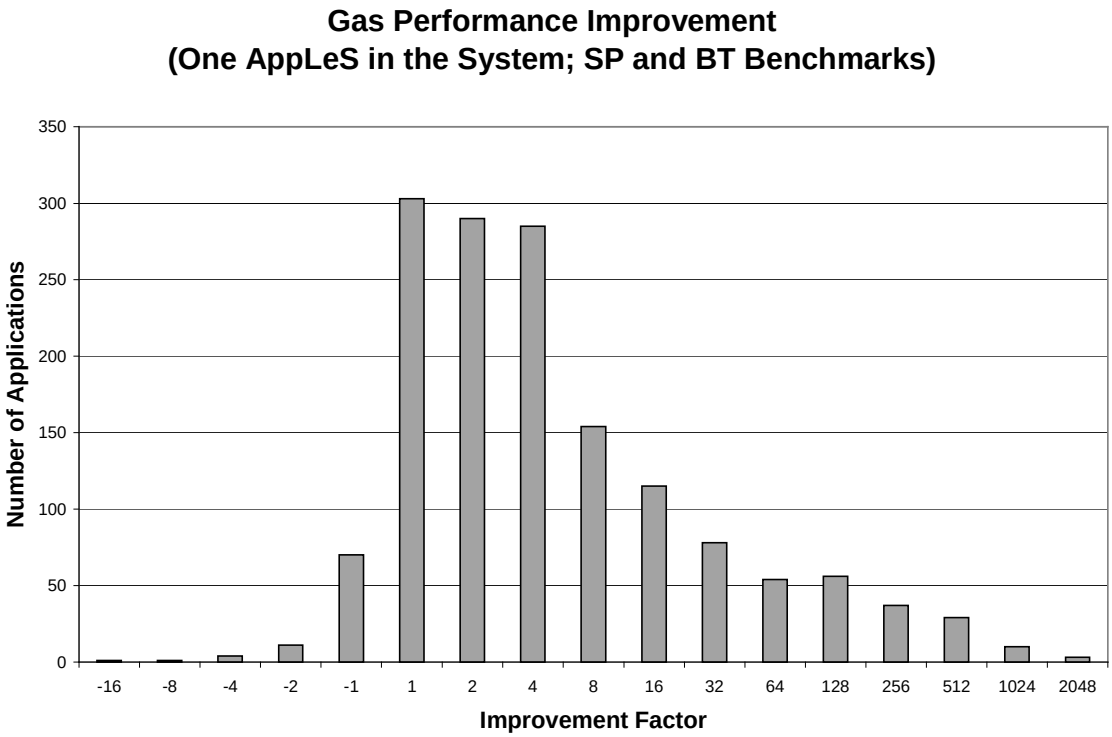

Figure 5.7 – The improvement factor obtained by Gas for SP and BT NAS benchmarks. One Gas request in the system.

… and those benchmarks that have no restrictions on the number of nodes (EP):

Turn-Around Time (One AppLeS in the System; EP Bench)		
	Gas	Static
mean	1081.02	14922.92
standard deviation	3729.27	30983.44
median	47	706
minimum	2	7
maximum	47803	242621

Table 5.7 – Turn-around time of Gas and static requests for EP NAS benchmarks. One Gas request in the system.

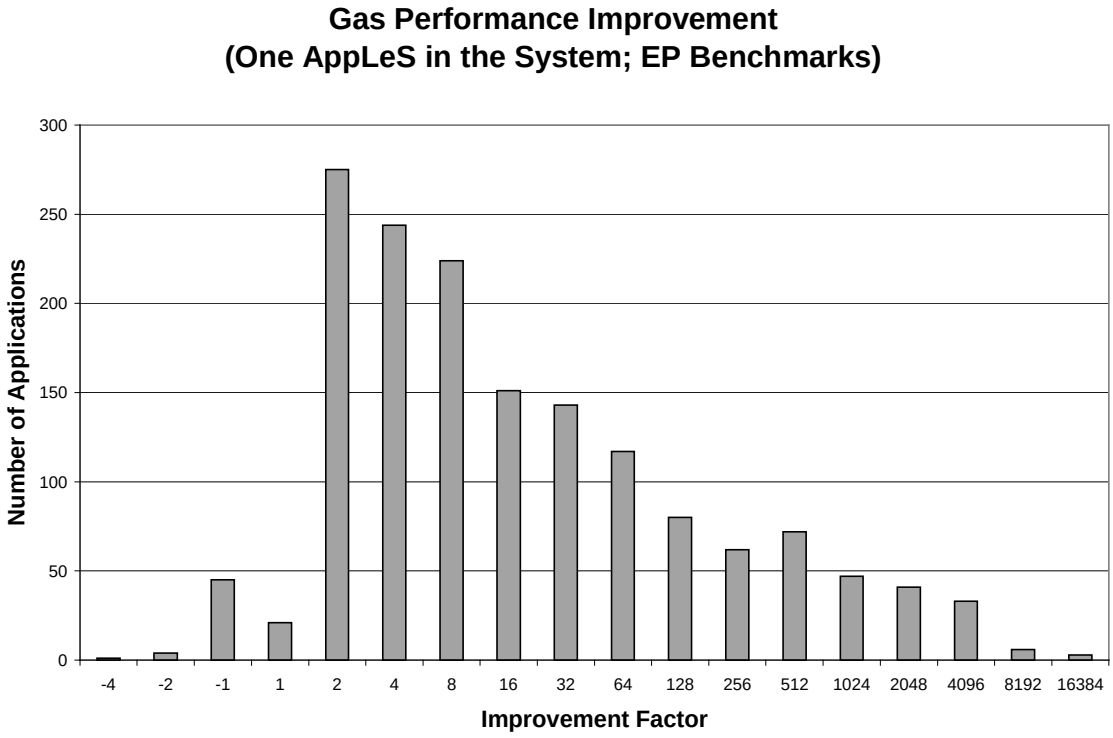

Figure 5.8 – The improvement factor obtained by Gas for EP NAS benchmarks. One Gas request in the system.

As one would expect, Gas does a better job for unrestricted applications than for restricted ones. However, it still works well with restricted applications, which suggest that, in this environment, the AppLeS approach is useful even when the application poses restrictions to the scheduler.

5.3. The Bushel of AppLeS

Of course, there is the question of how multiple AppLeS affect each other and the system as a whole. There are two basic concerns about a system in which many entities make decisions independently. First, is the system as a whole stable, or does it oscillate in some thrashing cycle? Second, what is the impact of multiple AppLeS on the performance attained by each of them?

In this environment, the stability of the system is *not* a problem. That's because each Gas makes only one decision. There is no chance for feedback behavior. However, the performance impact of having multiple AppLeS in the system is definitely an issue. We expect the improvements to individual application performance to be smaller if many applications have their requests crafted by AppLeS because the system as whole becomes more efficient, and thus it is harder for Gas to find a good slot in the availability list. That is, the competition for resources becomes tougher with multiple application schedulers.

However, investigating this hypothesis is an experimental challenge because we need to replace applications in the original workloads with application of known speed-up behavior to be able to use Gas. The problem is that the more applications we replace, the more we alter the original workloads, and thus the less realistic they become.

Although we cannot avoid this problem within our current experimental set-up, we can gauge how much a replacement affects a workload. This is done by comparing the altered workload's total requested CPU time against the original workload's total requested CPU. More precisely, we get the *load ratio* by dividing the altered workload's total requested CPU time by the original workload's one. We use this value to determine the "grain of salt" with which we should look at a particular simulation.

Using the same parameters as before[1], we have replaced up to 50% of the original workloads with NAS benchmarks. Table 5.8 presents the load ratio of the obtained workloads. The KTH workload was more amendable to have application replacement. It just happens that the KTH workload had more applications whose requests were similar to the NAS benchmarks ones.

Load Ratio			
replacement	**SDSC**	**CTC**	**KTH**
5%	0.9456	0.9592	0.9801
10%	0.9063	0.9164	0.9537
15%	0.8660	0.8749	0.9263
20%	0.8217	0.8306	0.9075
25%	0.7868	0.7758	0.8903
30%	0.7568	0.7465	0.8785
35%	0.7239	0.7083	0.8572
40%	0.7130	0.7130	0.8425
45%	0.6937	0.6976	0.8328
50%	0.6649	0.6883	0.8201

Table 5.8 – Load ratio of replacing different percentages of the total applications in the SDSC, CTC, and KTH workloads

Gas performed better than static in all 30 scenarios represented in Table 5.8. Tables 5.9, 5.10, and 5.11 show the mean turn-around time for both Gas and static for all these scenarios. Although not as good as before, the results still show substantial performance improvement in all cases. Unfortunately, the scenarios with low load ratio are less realistic and thus the simulations involving them should be considered with caution.

[1] Benchmarks are distributed in the following way: 1/3 require power-of-two nodes (MG and LU), 1/3 require perfect square nodes (SP and BT), and 1/3 are not restricted (EP). Inputs (B or C) are uniformly distributed.

Mean Turn-Around Time (Bushel of AppLes; SDSC workload)		
replacement	**Gas**	**Static**
5%	5586.05	16828.81
10%	6420.48	14662.31
15%	6265.74	14247.19
20%	6197.55	17724.67
25%	7021.05	17212.81
30%	6952.22	17282.07
35%	7716.13	15543.10
40%	7558.03	15810.28
45%	7151.94	15462.95
50%	7182.95	13403.38

Table 5.9 – Mean Turn-Around Time for Gas and static requests when replacing different percentages of the total applications in the SDSC workload

Mean Turn-Around Time (Bushel of AppLes; CTC workload)		
replacement	**Gas**	**Static**
5%	1386.92	2405.83
10%	1489.86	2426.74
15%	1526.67	2505.07
20%	1766.67	2660.16
25%	1916.97	2661.39
30%	1847.66	2583.77
35%	1918.92	2517.71
40%	2009.95	2597.10
45%	1959.91	2601.63
50%	2052.23	2579.12

Table 5.10 – Mean Turn-Around Time for Gas and static requests when replacing different percentages of the total applications in the CTC workload

Mean Turn-Around Time (Bushel of AppLes; KTH workload)		
replacement	**Gas**	**Static**
5%	4457.15	8078.23
10%	4388.35	8030.74
15%	4459.20	8303.79
20%	4576.61	8689.26
25%	4500.23	8410.98
30%	4642.08	8250.78
35%	4863.60	7939.14
40%	4540.10	7864.41
45%	4586.62	7424.85
50%	4307.59	7482.15

Table 5.11 – Mean Turn-Around Time for Gas and static requests when replacing different percentages of the total applications in the KTH workload

Taking a closer look

Looking at the mean turn-around time conveys an overview of the results. However, statistics can be misleading and thus it is important to take a closer look at the performance improvement obtained by Gas.

We here show detailed data of one scenario for each supercomputer. Since scenarios with low load ratio are less realistic, let's establish the "acceptable" load ratio to be 0.8. This gives us a maximum of 20% replacement for SDSC and CTC, and 50% for KTH. Various statistics summarizing the turn-around time and the distribution of the improvement obtained by Gas under these conditions follow:

Turn-Around Time (20% of AppLeS in the System; SDSC workload)		
	Gas	Static
mean	6197.55	17724.67
standard deviation	11694.15	33401.58
median	789	2961
minimum	7	58
maximum	134825	259442

Table 5.12 – Turn-around time of Gas and static requests. Simulation based on the SDSC workload. 20% of the requests in the system are generated by Gas.

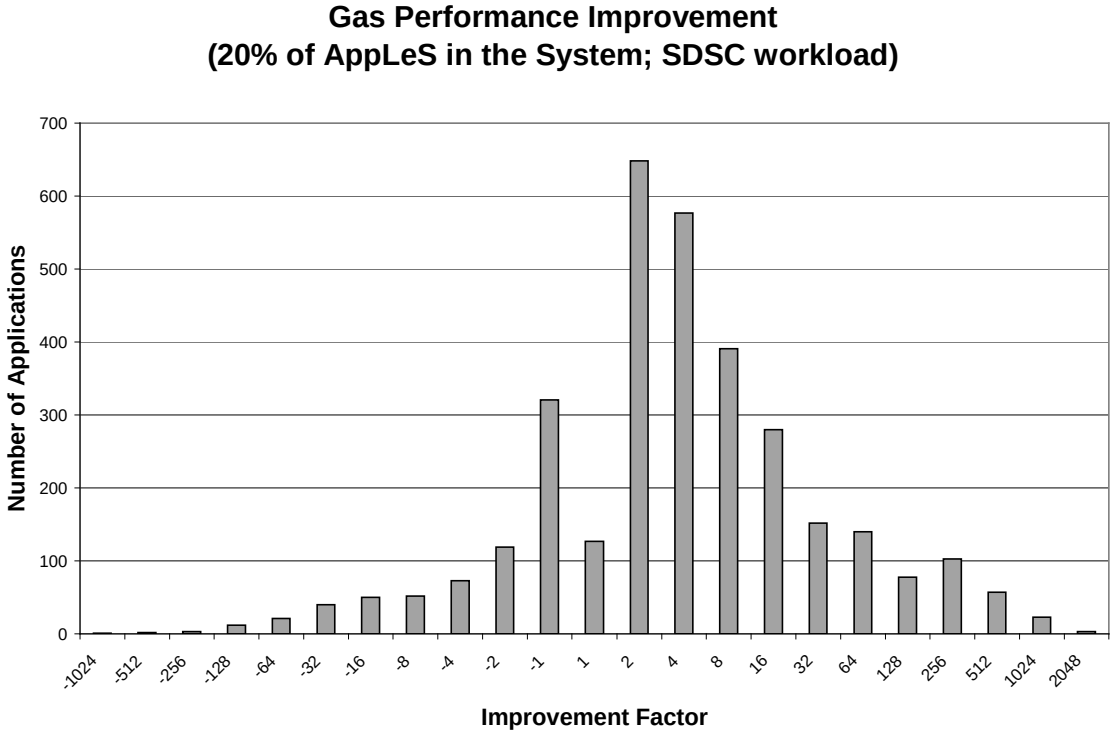

Figure 5.9 – The improvement factor obtained by Gas. Simulation based on the SDSC workload. 20% of the requests in the system are generated by Gas.

Turn-Around Time (20% of AppLeS in the System; CTC workload)		
	Gas	Static
mean	1766.67	2660.16
standard deviation	4901.55	6057.09
median	143	431
minimum	2	58
maximum	70847	75313

Table 5.13 – Turn-around time of Gas and static requests. Simulation based on the CTC workload. 20% of the requests in the system are generated by Gas.

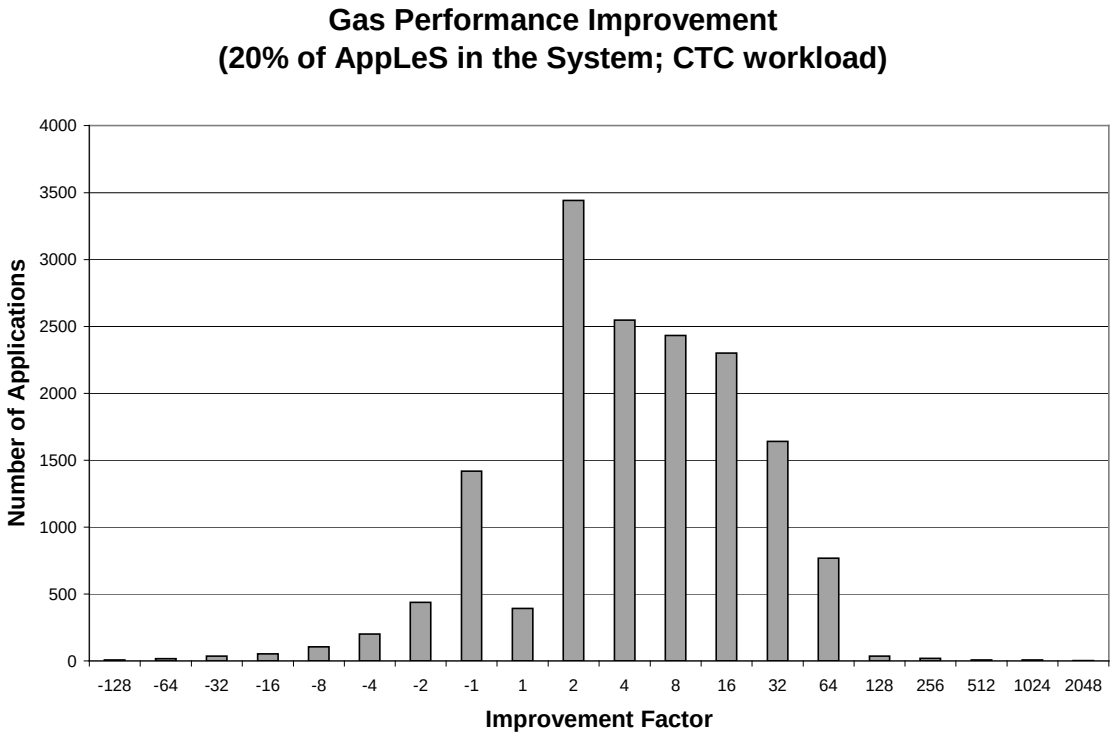

Figure 5.10 – The improvement factor obtained by Gas. Simulation based on the CTC workload. 20% of the requests in the system are generated by Gas.

Turn-around Time (50% of AppLeS in the System; KTH workload)		
	AppLes	Static
mean	4307.59	7482.15
standard deviation	8482.43	13756.20
median	698	2336
minimum	3	3
maximum	81768	208049

Table 5.14 – Turn-around time of Gas and static requests. Simulation based on the KTH workload. 50% of the requests in the system are generated by Gas.

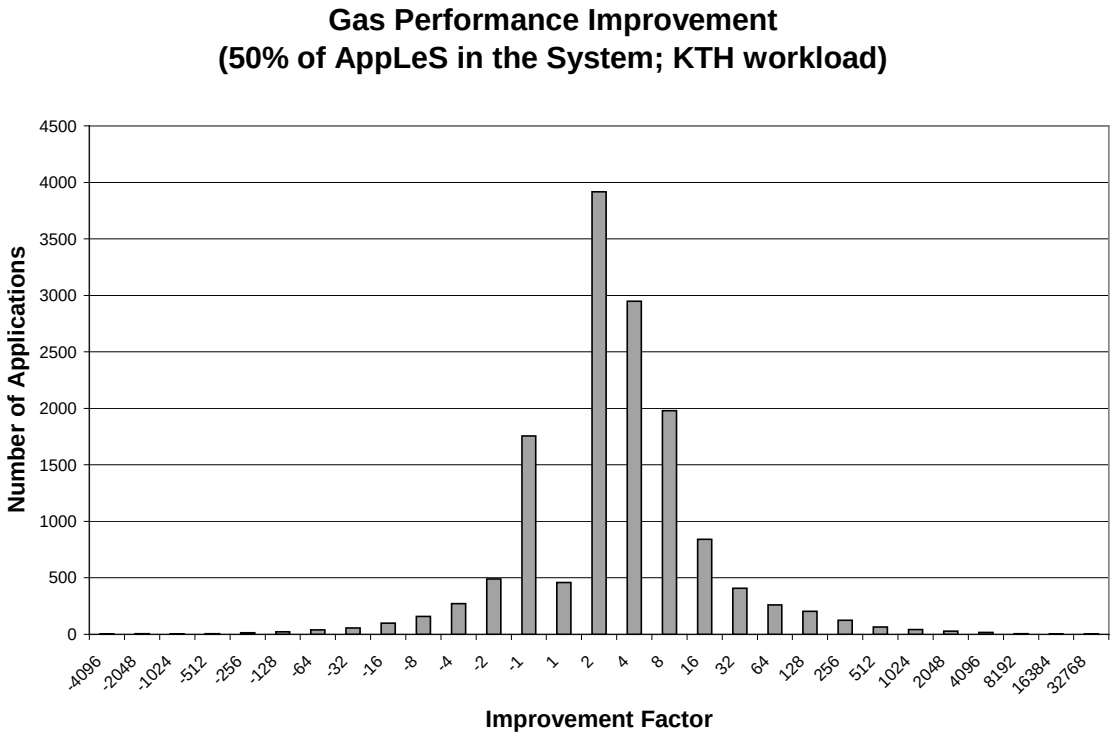

Figure 5.11 – The improvement factor obtained by Gas. Simulation based on the KTH workload. 50% of the requests in the system are generated by Gas.

Note that the number of times Gas does worse than static (and also by how much) is greater than the previous, one-AppLes-in-the-system case. We believe that this is because multiple instances of Gas make the system as whole more efficient, and thus the cases where the static was “lucky” (e.g., the application arrives and runs immediately) appear to lose, because it is harder to be lucky in a more efficient environment.

The Influence of Restrictions

In order to evaluate whether the number-of-processors restrictions of certain NAS benchmarks play a different role when multiple instances of Gas are in the system, we reran the simulations over these 30 scenarios with two different replacement parameters, therefore generating 60 new scenarios. First, we introduced into the workload only benchmarks that place no restriction (i.e., EP). Second, we introduced only benchmarks that have restrictions (i.e., SP, BT, LU, and MG)[2].

The results showed no surprises. They were similar to those obtained in the one-AppLeS-in-the-system case. Gas was able to deliver a better performance improvement for unrestricted applications. But even for restricted applications, Gas managed to substantially improve their performance compared to static. In fact, Gas achieved better performance than static for all 60 scenarios.

Tables 5.15, 5.16, and 5.17 show the mean turn-around time for Gas and static request, as well as the load ratio of the modified workloads. Again, the results for workloads with low load ratio should be taken with caution. For example, for all three original workloads, the mean turn-around time for static requests decreases with the growth of the percentage of NAS benchmarks in the workload. However, we don’t think this is a real phenomenon. Instead, we attribute it to the fact that the overall load decreases in inverse proportion to the percentage of NAS benchmarks in the workload (as can be seen via the load ratios). The turn-around time therefore declines because the system as a whole is less loaded.

[2] In this case, the benchmarks were uniformly distributed.

Mean Turn-Around Time and Load Ratio (Bushel of AppLes; SDSC workload)						
Replace-ment	Unrestricted			Restricted		
	Turn-Around Time		Load Ratio	Turn-Around Time		Load Ratio
	Gas	Static		Gas	Static	
5%	2116.27	7412.32	0.9731	7506.35	18697.95	0.9370
10%	2859.34	8169.34	0.9429	8179.76	18076.80	0.8812
15%	3235.89	9858.24	0.9094	8883.49	19422.11	0.8350
20%	4231.36	11073.22	0.8763	9987.68	20302.63	0.7934
25%	4072.27	11435.42	0.8342	9256.60	18001.05	0.7379
30%	4709.02	11299.93	0.8103	9509.98	17093.03	0.7067
35%	4480.59	11442.08	0.7939	9119.92	15758.79	0.6477
40%	4613.76	11783.23	0.7611	8969.71	14095.55	0.6188
45%	4699.68	10814.17	0.7448	8910.98	12971.04	0.5817
50%	4346.42	10864.12	0.7302	8271.05	11348.15	0.5490

Table 5.15 – Load Ratio and Mean Turn-Around Time for replacing different percentages of the total applications in the SDSC workload by restricted and unrestricted NAS benchmarks

Mean Turn-Around Time and Load Ratio (Bushel of AppLes; CTC workload)						
Replace-ment	Unrestricted			Restricted		
	Turn-Around Time		Load Ratio	Turn-Around Time		Load Ratio
	Gas	Static		Gas	Static	
5%	659.40	1356.38	0.9882	2008.83	3037.53	0.9592
10%	686.44	1376.04	0.9738	2114.45	3169.64	0.9164
15%	709.16	1430.28	0.9587	2176.52	3240.86	0.8749
20%	904.06	1584.78	0.9420	2435.71	3316.11	0.8306
25%	1011.53	1646.98	0.9068	2383.49	3089.23	0.7758
30%	1140.63	1711.19	0.8648	2303.54	2898.63	0.7465
35%	1155.27	1791.02	0.8363	2049.54	2715.09	0.7083
40%	1391.84	1944.87	0.8306	2268.68	2934.85	0.7130
45%	1643.25	2320.77	0.8095	2312.17	2894.31	0.6976
50%	1642.72	2360.01	0.7858	2291.15	2937.44	0.6883

Table 5.16 – Load Ratio and Mean Turn-Around Time for replacing different percentages of the total applications in the CTC workload by restricted and unrestricted NAS benchmarks

Mean Turn-Around Time and Load Ratio (Bushel of AppLes; KTH workload)						
Replace-ment	Unrestricted			Restricted		
	Turn-Around Time		Load Ratio	Turn-Around Time		Load Ratio
	Gas	Static		Gas	Static	
5%	1702.86	6852.41	0.9801	5966.57	8479.06	0.9690
10%	1955.42	7552.06	0.9537	5863.45	8298.82	0.9389
15%	2075.39	7914.31	0.9263	6017.33	8936.67	0.9083
20%	2151.62	7351.20	0.9075	6163.20	8829.82	0.8785
25%	2342.17	7395.70	0.8903	5997.08	8292.42	0.8496
30%	2360.50	7200.26	0.8785	5697.56	8104.84	0.7965
35%	2472.68	7276.24	0.8572	5323.97	7713.79	0.7753
40%	2470.89	7055.69	0.8425	4511.77	6619.57	0.7227
45%	2551.17	7084.53	0.8328	3924.95	5686.92	0.6830
50%	2691.49	6547.67	0.8201	3774.14	5330.33	0.6575

Table 5.17 – Load Ratio and Mean Turn-Around Time for replacing different percentages of the total applications in the KTH workload by restricted and unrestricted NAS benchmarks

6. Future Work

This is one of first efforts on designing resource schedulers for the meta-computing environment. As one would expect, there are many issues that need further research.

We here identify a number of these research opportunities, and propose a subset of them to be the core of this project. We then elaborate on how we intend to approach the selected research topics. In particular, much detail is given on how we plan to address our proposed next topic.

6.1. Research Opportunities

The good initial results of S^3 and Gas set the stage for a number of interesting investigations:

Bushel of AppLeS

The Bushel of AppLeS phenomenon that appears when multiple instances of Gas are in the system needs to be investigated further. We have already approached this issue, but our current experimental procedure has limitations. The source of our limitations is the need to replace applications in logged workloads. It degrades the workload in direct proportion to the percentage of replaced applications. Since Gas can work only with replaced applications, this precludes us from examining what happens when most applications are AppLeSized.

A major challenge here is the derivation of workloads that are both realistic and contain speed-up information about most of the applications. A major modeling effort is clearly necessary. Previous work suggests that diversity plays an important role in systems with multiple decision-makers [Hogg 91] [Mitzenmacher 97]. This calls for the increased sophistication of the model (in order to expose application diversity) and makes the modeling effort even more challenging.

Accuracy of the Performance Model

Part of the application diversity comes from the variability in the accuracy of the estimates of applications' execution times. Some applications are more deterministic and simpler to estimate than others, and thus can be represented by more accurate performance models. Moreover, different users are probably putting different amounts of effort in building their application's performance model.

Not only is this relevant for modeling workloads to study the Bushel of AppLeS problem; the accuracy of the performance model is crucial for the AppLes per se. For example, Gas requires a performance model in order to schedule an application. It is therefore important to determine the minimum accuracy of a performance model for it to be useful for Gas. This will enable us to determine whether automatically generated performance models (such as those in [Kapadia 99] and [Smith 98]) are accurate enough to be used by Gas.

Bushel of AppLes over Multiple Supercomputers

Although we are currently focusing on one supercomputer, the whole motivation behind metacomputing-friendly resource schedulers is to empower users to efficiently use multiple resources, including multiple supercomputers.

In such environment, a simple application scheduling scheme seems very attractive: Submit the application to all possible resources. Although this is a good idea from the user's viewpoint, if everybody does it, the aggregated load over all supercomputers rises, and thus the performance degrades. Consequently, unless we make the option of submitting to all possible resources unattractive, a Bushel of AppLeS effect is very likely to degrade the performance of an environment formed by multiple supercomputers.

A natural way to address this issue is to "charge" for submissions, extending the widely applied policy of accounting utilization to assure users don't exceed their granted CPU time. The goal would be to determine how much CPU time should be granted and how requests should be charged in order to make it in the users' best interest not to submit to all possible resources.

Coallocation

Another important issue is whether the resource scheduler can provide some additional service to ease coallocation across multiple supercomputers. Reservations and availability lists make possible for the AppLeS to come up with the requests to be sent to multiple supercomputers in order to coallocate a set of resources. However, the backfilling of these requests would happen independently from one another, probably breaking the coallocation apart.

There is the naive solution of marking requests as non-backfillable, but not benefiting from backfilling would probably hurt the application's turn-around time. Such a naive solution might render coallocation uncompetitive compared with using a single supercomputer (where backfilling can be used with no problem).

Maybe there is a way to enhance the interface provided by the resource scheduler to better support coallocation. Two ideas currently come to mind. First, the AppLeS can inform the resource scheduler which time slots are most interesting. Second, the backfilling can be *conditional* (i.e., you don't lose your original time slot). The idea behind conditional backfilling is that you get two (or maybe more) allocation slots, but you can use only one. The AppLeS would have to release the slots that the application is not going to use.

Priorities

The introduction of priorities into conservative backfilling (while preserving the features that make it a friendly substratum for application scheduling) is another important topic for future research. Priorities are a requirement of many supercomputer administrators. If properly decoupled from accounting, they can enable both administrators to implement local scheduling policies, and users to express the relative importance of their applications.

Execution-Time Predictions

As mentioned before, Gas acts as if the applications in the system would use all the time they've asked for. Maybe we can do better by using some prediction scheme. This probably would include a better understanding of how the number of nodes of a request affects how the request is backfilled.

6.2. Establishing the Project Goals

We envision our research as setting up the basis for a production-quality supercomputer-based metacomputing environment. We therefore propose to address the most critical problems that need to be solved to turn this vision into reality. More precisely we propose to extend the results presented here to:

i) *Readdress the Bushel of AppLeS question in the environment formed by one S^3 and multiple instances of Gas*. That is, model the supercomputer submission stream to be able to investigate the Bushel of AppLeS question more comprehensively.

ii) *Develop a cost scheme that discourages submitting to all possible resources*. This can be seen as dealing with the Bushel of AppLeS question in the environment formed by multiple instances of both S^3 and Gas. Of course, Gas has to be extended to deal with multiple super-

computers. We intend to do so by adding resource selection to Gas, which is a natural extension of the current algorithm.

iii) *Deploy S^3 and Gas over a real system*. This would enable us to run some real-life experiments to complement our simulation-based results. Since S^3 and Gas are already running, this step would consist of implementing user and administrative interfaces, as well as enabling S^3 to control real resources. We intend to use the LoadLeveler API [Skovira 96] as the mechanism S^3 will use to control resources. This would make it automatically SP2-ready.

We plan to deploy S^3 and Gas at first over the Circus Cluster located at UCSD's Parallel Computation Lab. Larger resources will be targeted if we have the opportunity.

6.3. The Next Step

Our next step (bullet i) in the last Subsection) starts by modeling the supercomputer submission stream with enough detail to investigate the Bushel of AppLeS question in the environment formed by one S^3 and multiple instances of Gas. The salient features we need to have in such a model are:

i) *Applications' interarrival time*.

Previous studies maintain that the application's interarrival time follows a uniform-log distribution [Downey 97b] [Feitelson 95]. Our own observations suggest the same. We can also use the interarrival time of existent logs. Therefore, we don't expect to encounter much trouble with this component of the model.

ii) *Applications' speed-up behavior*.

Downey has developed a model that accurately captures the speed-up behavior of parallel applications with only two parameters [Downey 97a]. However, it is not known how these parameters are distributed in a real submission stream (although Downey has some conjectures about this [Downey 97b]).

We intend to find realistic distributions for Downey's parameters by trying to fit his model to applications taken from logged workloads. Unfortunately, most workloads do not have enough information to allow for this (namely, they miss the application name and parameters).

If we cannot solve the problem this way, we plan to conduct a survey among NPACI users to gather the information needed to construct a realistic model for the speed-up behavior.

iii) *Possible applications' partition sizes*.

We don't know of any study characterizing this aspect of a supercomputer submission stream. We intend to model it by observing existing logs. Unfortunately, we suffer from the same problem of lack of information in most currently available logs. Again, the back-up plan is to base modeling this facet of the submission stream on a survey with NPACI users.

iv) *Accuracy and completeness of the performance model*.

Although there are works that take this feature of the submission stream into consideration [Feitelson 98] [Zotkin 99], they simply assume it to be uniformly distributed. We believe this to be an oversimplifying assumption. Observations of the accuracy of the requests in existing submission streams show it to be far from uniformly distributed.

The performance model may also be incomplete, in the sense that it might not cover all possible partition sizes a particular application can use.

We intend to use the accuracy of automatically generated performance models as the basis for this aspect of our submission stream model. That's because we believe that most users won't want to spend much effort in building refined performance models of their applications. Therefore, automatically generated performance models are likely to be deployed with most instances of Gas.

With the submission stream model completed and validated, we can then use it to drive simulations involving multiple instances of Gas running over S^3 and therefore address the Bushel of AppLeS problem in this environment.

References

[Amoroso 98] Alessandro Amoroso, Keith Marzullo, and Aleta Ricciardi. *Wide-Area Nile: A Case Study of a Wide-Area Data-Parallel Application*. ICDCS'98 – International Conference on Distributed Computing Systems. May 1998.

[Andersen 98] D. Andresen, Tao Yang, O. Ibarra, and O. Egecioglu. *Adaptive partitioning and scheduling for enhancing WWW application performance*. Journal of Parallel and Distributed Computing, vol.49, (no.1), Academic Press, 25 Feb. 1998. p.57-85.

[Babaoglu 97] O. Babaoglu, A. Bartoli, G. Dini. *Enriched View Synchrony: a Programming Paradigm for Partitionable Asynchronous Distributed Systems*. IEEE Transactions on Computers, vol.46, (no.6), IEEE, June 1997.

[Berman 96] Fran Berman, Richard Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. *Application-Level Scheduling on Distributed Heterogeneous Networks*. Supercomputing'96.
http://www-cse.ucsd.edu/groups/hpcl/apples/hetpubs.html

[Berman 97] Fran Berman and Rich Wolski. *The AppLeS Project: A Status Report*. In Proceedings of the 8th NEC Research Symposium, Berlin, Germany, May 1997.
http://www.cs.ucsd.edu/groups/hpcl/apples/hetpubs.html

[Berman 99] Fran Berman. *High-Performance Schedulers*. In [Foster 99b]. 1999.

[Bhat 98] Prashant Bhat, Viktor Prasanna, and C. S. Raghavendra. *Adaptive Communication Algorithms for Distributed Heterogeneous Systems*. Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC'98). Chicago, Illinois, USA. July 1998.

[Bos 94] Dieter Bos. *Pricing and Price Regulation: An Economic Theory for Public Enterprise and Public Utilities*. Elsevier Science, 1994.

[Bunt 76] R. B. Bunt. *Scheduling Techniques for Operating Systems*. Computer. October 1976.

[Casanova 96] Henri Casanova and Jack Dongarra. *NetSolve: A Network Server for Solving Computational Science Problems*. Supercomputing'96, 1996.
http://www.netlib.org/utk/people/JackDongarra/papers.html

[Chapin 95] Steve Chapin. *Distributed Scheduling Support in the Presence of Autonomy*. Proceedings of the 4th Heterogeneous Computing Workshop, pp. 22-29, Santa Barbara, CA, April 1995.
http://www.cs.virginia.edu/~chapin/papers/hcw95.ps

[Chapin 99] Steve Chapin, Walfredo Cirne, Dror Feitelson, James Jones, Scott Leutenegger, Uwe Schwiegelshohn, Warren Smith, and David Talby. *Benchmarks and Standards for the Evaluation of Parallel Job Schedulers*. In Job Scheduling Strategies for Parallel Processing, D. Feitelson and L. Rudolph (Eds.) Springer-Verlag, Lecture Notes in Computer Science, vol. 1659, pp. 66-89, 1999.
http://www-cse.ucsd.edu/users/walfredo/resume.html#publications

[Cheng 98] John Cheng and Michael Wellman. *The WALRAS algorithm: A Convergent Distributed Implementation of General Equilibrium Outcomes*. Computational Economics, 12:1-24, 1998.
http://ai.eecs.umich.edu/people/wellman/Publications.html#MOP

[Cirne 99a] Walfredo Cirne and Keith Marzullo. *The Computational Co-op: Gathering Clusters into a Metacomputer*. In Proceeding of IPPS/SPDP'99, April 1999.
http://www-cse.ucsd.edu/users/walfredo/resume.html#publications

[Cirne 99b] Walfredo Cirne, Jaime Frey, Shava Smallen, Francine Berman, Rich Wolski, Steve Young, Mark Ellisman, Mei-Hui Su, and Carl Kesselman. *Combining Workstations and Supercomputers to Support Metacomputing Applications: The Parallel Tomography Experience*. UCSD/CSE Tech Report CS99-620, April 1999.
http://www-cse.ucsd.edu/users/walfredo/resume.html#publications

[Clearwater 96] Scott Clearwater (editor). *Market-Based Control: A Paradigm for Distributed Resource Allocation*. World Scientific. 1996.

[Czajkowski 98] K. Czajkowski, I. Foster, C. Kesselman, N. Karonis, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for

Metacomputing Systems. IPPS/SPDP'98 Workshop on Job Scheduling Strategies for Parallel Processing, 1998.
http://www-fp.globus.org/documentation/papers.html

[Dinda 99] P. Dinda, D. O'Hallaron. *An Evaluation of Linear Models for Host Load Prediction*. Proceedings of the 8th IEEE Symposium on High-Performance Distributed Computing (HPDC-8), Redondo Beach, CA, August 1999.
http://www.cs.cmu.edu/afs/cs/usr/pdinda/html/papers.html

[Downey 97a] Allen B. Downey. *A model for speedup of parallel programs*. U.C. Berkeley Technical Report CSD-97-933.
http://www.sdsc.edu/~downey/model/

[Downey 97b] Allen B. Downey. *A parallel workload model and its implications for processor allocation*. 6th IEEE International Symposium on High Performance Distributed Computing (HPDC'97).
http://www.sdsc.edu/~downey/allocation/

[Downey 97c] Allen B. Downey. *Predicting queue times on space-sharing parallel computers*. 11th International Parallel Processing Symposium, Geneva (IPPS'97), Switzerland, April 1997.
http://www.sdsc.edu/~downey/predicting/

[Downey 99] A. B. Downey and D. G. Feitelson. *The elusive goal of workload characterization*. Perf. Eval. Rev. 26(4), pp. 14-29, Mar 1999.
http://www.cs.huji.ac.il/~feit/pub.html

[Fagg 97] Graham Fagg, Keith Moore, Jack Dongarra, and Al Geist. *Scalable Networked Information Processing Environment (SNIPE)*. Supercomputing '97. San Jose, CA, USA. 1997.
http://www.supercomp.org/sc97/proceedings/TECH/MOORE/INDEX.HTM

[Feitelson 95] D. G. Feitelson and B. Nitzberg. *Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860*. In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (Eds.), Lecture Notes in Computer Science Vol. 949, pp. 337-360, Springer-Verlag, 1995.
http://www.cs.huji.ac.il/~feit/pub.html

[Feitelson 97a] D. G. Feitelson, L. Rudolph, U. Schweigelshohn, K. Sevcik, and P. Wong. *Theory and Practice in Parallel Job Scheduling*. 3rd Workshop on Job Scheduling Strategies for Parallel Processing, Springer-Verlag Lecture Notes in Computer Science, vol. 1291, pp. 1-34, April 1997.
http://www.cs.huji.ac.il/~feit/parsched/parsched97.html

[Feitelson 97b] D. G. Feitelson. *Packing schemes for gang scheduling*. In Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science vol. 1162, D. G. Feitelson and L. Rudolph (eds.), pp. 89-110, Springer-Verlag, 1996.
http://www.cs.huji.ac.il/~feit/pub.html

[Feitelson 98] D. G. Feitelson and A. Mu'alem Weil. *Utilization and predictability in scheduling the IBM SP2 with backfilling*. In 12th Intl. Parallel Processing Symp., pp. 542-546, Apr 1998.
http://www.cs.huji.ac.il/~feit/pub.html

[Ferrari 98] Adam Ferrari et al. *A Flexible Security System for Metacomputing Environments*. Technical Report CS-98-36, Department of Computer Science, University of Virginia.
http://www.cs.virginia.edu/~legion/papers.html

[Fitzgerald 97] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, S. Tuecke. *A Directory Service for Configuring High-Performance Distributed Computations*. 6th IEEE Symposium on High-Performance Distributed Computing, pg. 365-375, 1997.
http://www-fp.globus.org/documentation/papers.html

[Fong 95] Liana Fong and Mark Squillante. *Time-Function Scheduling: A General Apporach to Controllable Resource Management*. Technical Report RC 20155 (89194), IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY 10598, August 1995.
http://domino.watson.ibm.com/library/cyberdig.nsf/a3807c5b4823c53f85256561006324be/05a54a208d1f0f6a852565930072576a?OpenDocument

[Forrest 97] Stephanie Forrest et al. *Building Diverse Computer Systems*. Proceeding of the 6th Workshop on Hot Topics in Operating Systems, pp. 67-72. 1997.
ftp://ftp.cs.unm.edu/pub/forrest/hotos-97.ps

[Foster 97] I. Foster, J. Geisler, C. Kesselman, S. Tuecke. *Managing Multiple Communication Methods in High-Performance Networked Computing Systems*. Journal of Parallel and Distributed Computing, 40:35-48, 1997.
http://www-fp.globus.org/documentation/papers.html

[Foster 98] I. Foster, C. Kesselman, G. Tsudik, S. Tuecke. *A Security Architecture for Computational Grids*. Proc. 5th ACM Conference on Computer and Communications Security Conference, pg. 83-92, 1998.
http://www-fp.globus.org/documentation/papers.html

[Foster 99a] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, A. Roy. *A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation*. Submitted for publication.
http://www-fp.globus.org/documentation/papers.html

[Foster 99b] Ian Foster and Carl Kesselman (editors). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers. July 1998.

[Gibbons 97] Richard Gibbons. *A Historical Application Profiler for Use by Parallel Schedulers*. Lecture Notes in Computer Science, vol. 1297, 58-75, Springer-Verlag, 1997.

[Harty 96] Kieran Harty and David Cheriton. *A Market Approach to Operating System Memory Allocation*. In [Clearwater 96], 1996.
ftp://ftp.dsg.stanford.edu/pub/papers/memmarket.ps.Z

[Henderson 95] Robert L. Henderson. *Job Scheduling Under the Portable Batch System*. In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (Eds.), Lecture Notes in Computer Science Vol. 949, pp. 337-360, Springer-Verlag, 1995.

[Hogg 91] Tad Hogg and Bernardo Huberman. *Controlling Chaos in Distributed Systems*. IEEE Transactions on Systems, Man, and Cybernetics, vol. 21, no. 6, November/December 1991.

[Jann 97] Joefon Jann, Pratap Pattnaik, Hubertus Franke, Fang Wang, Joseph Skovira, and Joseph Riordan. *Modeling of Workload in MPPs*. In Job Scheduling Strategies for Parallel Processing, Springer-Verlag, Lecture Notes in Computer Science Vol. 1291, D. G. Feitelson and L. Rudolph (eds.), 1997.

[Karpovich 96] John F. Karpovich. *Support for Object Placement in Wide Area Heterogeneous Distributed Systems*. UVa CS Technical Report CS-96-03. January 1996.
http://www.cs.virginia.edu/~legion/papers.html

[Kapadia 99] Nirav Kapadia, José Fortes, and Carla Brodley. *Predictive Application-Performance Modeling in a Computational Grid Environment.* Eighth IEEE Symposium on High-Performance Distributed Computing, July 1999.

[Lifka 95] David Lifka. *The ANL/IBM SP Scheduling System*. In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (Eds.), Springer-Verlag, Lecture Notes in Computer Science Vol. 949, 1995.
http://www.tc.cornell.edu/UserDoc/SP/Batch/what.html

[Lowekamp 98] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. *A Resource Query Interface for Network-Aware Applications*. Seventh IEEE Symposium on High-Performance Distributed Computing, July 1998.
http://www.cs.cmu.edu/~cmcl/remulac/papers.html

[Maui 99] Maui High Performance Computing Center. *The Maui Scheduler Web Page*.
http://wailea.mhpcc.edu/maui/

[Mitzenmacher 97] Michael Mitzenmacher. *How Useful is Old Information?* PODC 97. 1997.
http://www.research.digital.com/SRC/personal/michaelm/WORK/papers.html

[Mullen 96] Tracy Mullen and Michael Wellman. *Market-based negotiation for digital library services*. Second USENIX Workshop on Electronic Commerce. November 1996.
ftp://ftp.eecs.umich.edu/people/wellman/usenix96.ps.Z

[MPI 95] MPI Forum. *MPI: A Message-Passing Interface Standard*. June 1995.
http://www.mpi-forum.org/docs/mpi-11.ps

[Platform 99] Platform Computing Corp. *Load Sharing Facility Web Page*.
http://www.platform.com/platform/platform.nsf/webpage/LSF?OpenDocument

[Previato 97] Fabio Previato, Michael Ogg, and Aleta Ricciardi. *Experience with Distributed Replicated Objects: The Nile Project*. European Research Seminar in Advanced Distributed Systems Zinal, Switzerland, 17-21 March 1997.
http://www.nile.utexas.edu/Nile/conferences/papers/

[Ranganathan 96] M. Ranganathan, A. Acharya, and J. Saltz. *Distributed Resource Monitor for Mobile Objects*. IWOOOS'96.
http://www.cs.umd.edu/users/acha/publications.html

[Shao 97] Gary Shao, Rich Wolski, and Fran Berman. *Predicting the Cost of Redistribution in Scheduling*. In Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing, 1997.
http://www.cs.ucsd.edu/groups/hpcl/apples/hetpubs.html

[Skovira 96] Joseph Skovira, Waiman Chan, and Honbo Zhou, IBM, and David Lifka. *The EASY-LoadLeveler API project*. 2nd Workshop on Job Scheduling Strategies for Parallel Processing, Springer-Verlag Lecture Notes in Computer Science, vol. 1162, pp. 41-47, April 1996.
http://www.tc.cornell.edu/UserDoc/SP/Batch/what.html

[Smith 98] W. Smith, I. Foster, and V. Taylor. *Predicting Application Run Times Using Historical Information*. Lecture Notes in Computer Science, 1459:122-142, Spring-Verlag, 1998.
http://www-fp.mcs.anl.gov/~wsmith/papers.html

[Smith 99] W. Smith, V. Taylor, and I. Foster. *Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance*. In Proceedings of the IPPS/SPDP'99 Workshop on Job Scheduling Strategies for Parallel Processing, 1999.
http://www-fp.mcs.anl.gov/~wsmith/papers.html

[Spring 98] Neil Spring and Rich Wolski. *Application Level Scheduling of Gene Sequence Comparison on Metacomputers*. 12th ACM International Conference on Supercomputing, Melbourne, Australia, July, 1998.
http://www-cse.ucsd.edu/groups/hpcl/apples/hetpubs.html

[Su 99] Alan Su, Francine Berman, Richard Wolski, and Michelle Strout. *Using AppLeS to Schedule Simple SARA on the Computational Grid*. In International Journal of High Performance Computing Applications, vol. 13, 1999.l
http://www.cs.ucsd.edu/groups/hpcl/apples/hetpubs.html

[Stoica 96] Ion Stoica et al. *A Proportional Share Resource Allocation for Real-Time, Time-Shared Systems*. 17th Real-Time Systems Symposium, Washington DC, p. 288-299, December 1996.
http://www.cs.odu.edu/~stoica/pubs.html

[Stonebraker 96] Michael Stonebraker et al. *Mariposa: A Wide-Area Distributed Database System*. VLDB Journal 5, 1, p. 48-63, January 1996.
http://epoch.cs.berkeley.edu:8000/mariposa/papers.html

[Tucker 96] Paul Tucker and Fran Berman. *On Market Mechanisms as a Software Technique*. UCSD Technical Report #CS96-513.
http://www.cs.ucsd.edu/groups/hpcl/apples/hetpubs.html#Miscellaneous

[Waldspurger 92] Carl Waldspurger et al. *Spawn: A Distributed Computational Economy*. IEEE Transactions on Software Engineering, vol. 18, no. 2, pp. 103-17. February 1992.
http://www.research.digital.com/SRC/personal/caw/papers.html

[Waldspurger 94] Carl Waldspurger and William Weihl. *Lottery Scheduling: Flexible Proportional-Share Resource Management*. In Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI '94), pages 1-11, Monterey, California, November 1994.
http://www.research.digital.com/SRC/personal/caw/papers.html

[Waldspurger 95] Carl Waldspurger and William Weihl. *Stride Scheduling: Deterministic Proportional-Share Resource Mangement*. Technical Memorandum MIT/LCS/TM-528, MIT Laboratory for Computer Science, June 1995.
http://www.research.digital.com/SRC/personal/caw/papers.html

[Walsh 98] William Walsh and Michael Wellman. *A market protocol for decentralized task allocation*. Extended version of a paper in Proceedgins of the Third International Conference on Multiagent Systems, July 1998.
http://ai.eecs.umich.edu/people/wellman/Publications.html

[Walsh 99] William Walsh and Michael Wellman. *Efficiency and equilibrium in task allocation economies with hierarchical dependencies*. In Sixteenth International Joint Conference on Artificial Intelligence, pages 520-526, August 1999.
http://ai.eecs.umich.edu/people/wellman/Publications.html

[Weissman 95] Jon Weissman and Andrew Grimshaw. *A Framework for Partitioning Parallel Computations in Heterogeneous Environments*. Concurrency: Practice and Experience, Vol. 7, No. 5, August 1995.
http://ringer.cs.utsa.edu/faculty/weissman.html/pub.html

[Weissman 98] Jon Weissman. *Gallop: The Benefits of Wide-Area Computing for Parallel Processing*. Journal of Parallel and Distributed Computing, Vol. 54(2), November 1998.
http://ringer.cs.utsa.edu/faculty/weissman.html/pub.html

[Wolski 98] Rich Wolski. *Dynamically Forecasting Network Performance Using the Network Weather Service*. In Journal of Cluster Computing, 1998.
http://www.cs.ucsd.edu/groups/hpcl/apples/hetpubs.html#NWS

[Wolski 99a] Rich Wolski, Neil T. Spring, and Jim Hayes. *The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing*. To appear in the Journal of Future Generation Computing Systems, 1999.
http://www.cs.ucsd.edu/groups/hpcl/apples/hetpubs.html#NWS

[Wolski 99b] R. Wolski, N. Spring and J. Hayes. *Predicting the CPU Availability of Time-shared Unix Systems on the Computational Grid*. 8th International Symposium on High Performance Distributed Computing (HPDC'99), Redondo Beach, California, USA, 3-6 Aug 1999.
http://www-cse.ucsd.edu/~rich/publications.html

[Zhu 98] Huican Zhu et 0al. *Adaptive Load Sharing for Clustered Digital Library Servers*. Seventh IEEE International Symposium on High Performance Distributed Computing, Chicago, Illinois, July, 1998. http://www.alexandria.ucsb.edu/~zheng/publications.html

[Zotkin 99] D. Zotkin and P. Keleher. *Sloppiness as a Virtue: Job-Length Estimation and Performance in Backfilling Schedulers*. 8th International Symposium on High Performance Distributed Computing (HPDC'99), Redondo Beach, California, USA, 3-6 Aug 1999.