Towards Autonomic Virtual Applications in the In-VIGO System

Jing Xu Sumalatha Adabala José A. B. Fortes
Advanced Computing and Information Systems Laboratory (ACIS)
Department of Electrical and Computer Engineering
University of Florida, Gainesville, Florida
{jxu, adabala, fortes}@acis.ufl.edu

Abstract

Grid environments enable users to share non-dedicated resources that lack performance guarantees. This paper describes the design of application-centric middleware components to automatically recover from failures and dynamically adapt to grid environments with changing resource availabilities, improving fault-tolerance and performance. The key components of the application-centric approach are a global per-application execution history and an autonomic component that tracks the performance of a job on a grid resource against predictions based on the application execution history, to guide rescheduling decisions. Performance models of unmodified applications built using their execution history are used to predict failure as well as poor performance. A prototype of the proposed approach, an Autonomic Virtual Application Manager (AVAM), has been implemented in the context of the In-VIGO grid environment and its effectiveness has been evaluated for applications that generate CPU-intensive jobs with relatively short execution times (ranging from tens of seconds to less than an hour) on resources with highly variable loads -- a workload generated by typical educational usage scenarios of In-VIGO-like grid environments. A memory-based learning algorithm is used to build the performance models for CPU-intensive applications that are used to predict the need for rescheduling. Results show that In-VIGO jobs managed by the AVAM consistently meet their execution deadlines under varying load conditions and gracefully recover from unexpected failures.

1. Introduction

Grid computing [4] enables users to share resources distributed across administrative domains. Idle and low-priority shared cycles from non-dedicated machines form an important class of grid resources. Two distinguishing features of such resources are, (a) intermittent participation (either voluntary or due to failure), and (b) highly variable load (due to their non-dedicated nature). This lack of performance guarantees makes it difficult to exploit such resources to support workloads that require a specified quality of service (QoS). The workloads of interest in this paper are scientific in nature, batch-oriented and characterized by relatively short execution times (ranging from tens of seconds to less than one hour). Educational usage scenarios, of grid-computing systems like PUNCH [16] and In-VIGO [3], offer many examples of large numbers of users running many relatively small jobs over concentrated periods of times (e.g. before homework deadlines), therefore, these applications may have time requirements. In spite of the short duration of these jobs, they play an important role in shaping the user experiences with a grid-computing system. Long response times for short jobs are less well tolerated than long execution times for tools known to take a long time to run. Unexpected long running times for supposedly quick tasks do discourage further use of a grid-computing system, and generate user discontent and customer losses. Our goal is to enable such workloads to adapt to the highly dynamic and fault-prone grid environments that result from sharing non-dedicated resources.

This paper describes an approach to autonomic computing [1][2] that is application-centric and leverages the use of virtualization technology in grid computing. The ideas have general applicability but the context is that of In-VIGO, a grid-computing system that provides application services on-demand using dynamically instantiated virtual machines, networks, data and applications.

Investigations of dependable computing techniques in the context of grid computing [17][18], have shown that the practical way of handling faulty resources is to make the client, i.e. the submitting endpoint,

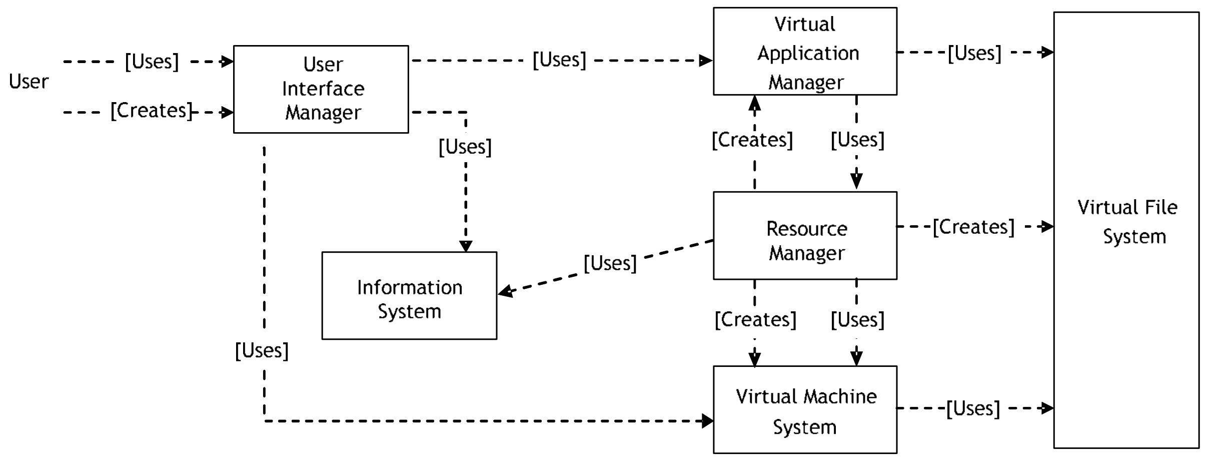

Figure 1. High-level block diagram of In-VIGO showing use-relationships and control interactions among its major components.

responsible for the progress and failure handling of applications. It is this client -- the application job management component, called the Virtual Application Manager (VAM) [3] in In-VIGO -- that the proposed approach aims to make autonomic, i.e. self-optimizing and self-healing.

Reliable grid components are typically built with one of two fault models. At one extreme is the Byzantine failure model, where the resource can exhibit arbitrary and malicious behavior, making it very hard to design the appropriate fault tolerant component. At the other extreme is the tractable, fail-stop model, where when a resource fails, all components are immediately aware of it. In the scenarios of interest, jobs whose execution times are expectably small can take an unacceptable long time to finish due to either failures or overloading of resources assigned to execute them. As such, a component built under the fail-stop model is prone to poor performance when resources are overloaded. The approach proposed in this paper takes into account these performance faults, and is designed under the fail-stutter fault model [19]. It relies on monitoring of job and resource conditions, predicting violations of user- and/or system-expected execution times and restarting jobs in resources capable of delivering acceptable times. Repeated/simultaneous executions (of jobs) are idempotent due to data virtualization techniques --Virtual File System (VFS) [20] -- used to manage the persistent state of jobs in In-VIGO, and this simplifies implementation of the autonomic component significantly.

A prototype of the proposed approach has been implemented in In-VIGO as the Autonomic Virtual Application Manager (AVAM). The AVAM has been evaluated for submissions of TunProb, a semiconductor electronics application, via In-VIGO to faulty resources, i.e. overloaded machines that result in performance faults. Results show that the In-VIGO jobs managed by AVAM consistently meet their execution deadlines by using available alternate resources while recovering from performance faults.

The rest of the paper is organized as follows. Section 2 introduces In-VIGO and the VAM component. Section 3 describes the architecture of the proposed framework and its implementation, AVAM, in In-VIGO. Section 4 presents results from quantitative performance analyses of the AVAM. Section 5 presents related work, and Section 6 concludes the paper.

2. Background

In-VIGO, In-Virtual Information Grid Organizations, is a grid-computing infrastructure that enables computational studies for engineering and science research on grid resources. Its distinctive feature is the extensive use of virtualization technologies to provide secure execution environments as needed by tools and users. A detailed discussion of its design and implementation appears in [3]. Its architecture is shown in Figure 1. Users interact with the middleware via a web-browser-based portal. Typically, a user initiates an application session, to run one or more instances of a tool on grid resources. An application session is managed by the VAM component in In-VIGO. The user's actions, for example input parameters to the tool, are directed via the User Interface Manager (UIM) to the VAM. The VAM interacts with the Resource Manager (RM) to launch the necessary tool executions on the available resources. The RM component obtains the status of available resources from the Information System (IS) component, allocates resources for a tool execution---creating them on demand if necessary via the VFS and Virtual Machine components---and manages the tool

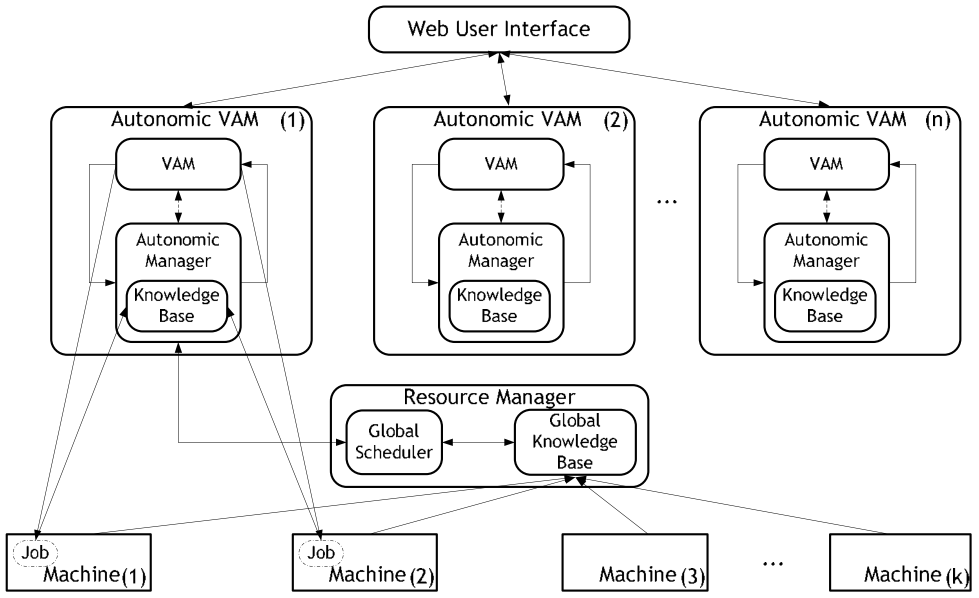

Figure 2. High-level view of the role of autonomic Virtual Application Manager in In-VIGO. This figure shows multiple AVAMs that connect to the Resource Manager to submit jobs on multiple machines.

execution on the resource.

The VAM manages all actions associated with user requests during an application session. The user can instantiate parameter-sweep jobs of a tool by specifying ranges of input parameters, or tools that use parallel programming interfaces such as MPI. The VAM responds to these requests by requesting the RM to provide the necessary resources and launching the tasks. The VAM has session-specific information, e.g. input parameters specified by the user, as well as application-specific knowledge, i.e. the logic needed to respond to user actions, specified in an application configuration file. The RM component that manages job executions on resources is an application neutral generic interface to heterogeneous resource APIs. An autonomic component that uses application-specific knowledge to manage application tasks can thus be implemented at the VAM. The following section describes the proposed application-centric approach to autonomic computing, and an implementation via extensions to the VAM.

3. Autonomic Virtual Application Manager

3.1. Architecture

Figure 2 provides a high-level view of the role of the autonomic Virtual Application Manager (AVAM) in In-VIGO. The user requests are directed from the web portal interface to the VAM component. This component implements the application-specific logic required to respond to the user requests. Rather than forwarding application tasks and their resource allocation to the RM component: the AVAM includes two additional components, an Autonomic Manager component that manages resource allocation for a job, by interpreting the current execution specific information in the Knowledge Base component and the application execution history in the Global Knowledge Base.

The Global Knowledge Base maintains application's execution history including the application parameters and resource usage. Resource independent parameters are used to measure the resource usage so that the AVAM can predict application performance based on execution history irrespective of the characteristics of the associated resource. The Global Knowledge Base also maintains the current status of computing resources, such as current CPU load and free memory size, measured by a monitoring daemon on each machine. The status updates sent at constant intervals serve as resource heart-beat signals. If these messages are not received in a timely manner from a machine, the machine is considered unreachable or overloaded. A global scheduler module that is part of the resource manager assists the AVAM to discover and allocate the most appropriate resource for each given run. The AVAM can query the Global Knowledge Base for resources that meet some specified constraints.

The self-optimizing and self-healing functions of the AVAM are implemented by the Autonomic Manager component. Information used by the Autonomic Manager is stored in the per-AVAM

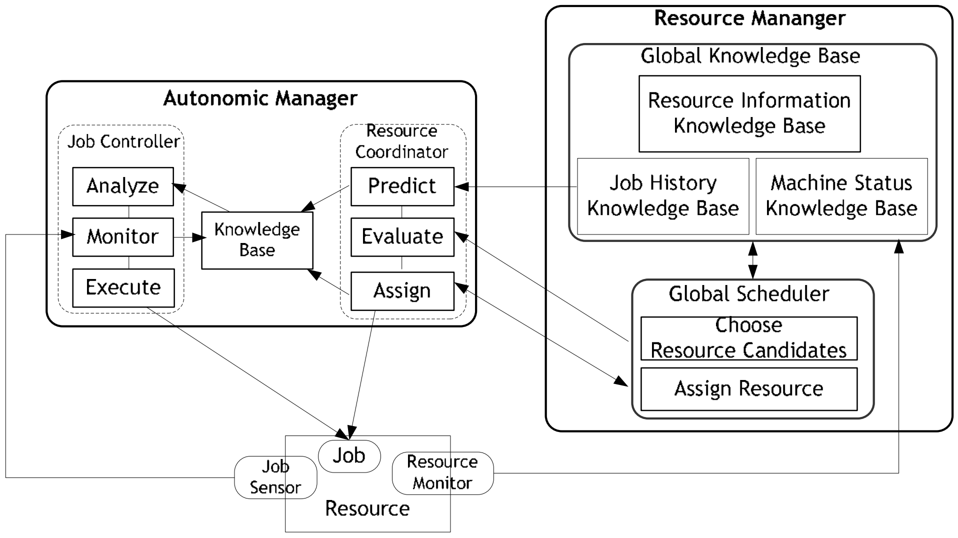

Figure 3. Autonomic Virtual Application Manager components and their interactions.

Knowledge Base. This includes user and application specific information (e.g. expected performance and input parameters), predicted resource usage, and updated job status and resource status.

The Autonomic Manager consists of two components, the local (per-AVAM) resource coordinator and the local controller. The local resource coordinator uses a learning algorithm to predict the resource usage for each given inputs-specific run from previous runs' history records extracted from the Global Knowledge Base. Based on the user's requirements and predicted resource usage, the local resource coordinator allocates, i.e. marks it as used, the proper resource for the run by cooperating with the global resource scheduler. The allocation of resources by the AVAM for tasks initiated by it allows optimal application-specific choice of resources, but its coordination with the global scheduler to mark resources as used makes it possible to have a central resource database that does not have scalability issues in grid environments.

The controller is responsible for controlling the job execution to achieve reliable and optimized performance. At the time of job submission, a job sensor is invoked on the chosen machine by the controller to monitor the job status and update the local Knowledge Base. Based on the monitored job status and predicted resource usage, the controller determines the progress of the given run and compares it with the user's QoS requirements to decide control actions. The actual submission and management of a job on the selected resource is accomplished via the RM component, which virtualizes the file space of the application via the VFS component. Each task gets its own copy of the file system to store its persistent state, i.e. file input/output. This makes In-VIGO jobs idempotent, and as a result there is no need to roll back applications to their initial state when rescheduling them via the RM component to alternate resources.

3.2. Implementation

Figure 3 illustrates the major functions of the local resource coordinator and job controller in the autonomic manager and the interactions within these functions and with outside. The modules in the local resource coordinator implement three functions named `predict`, `evaluate`, and `assign`. The functions named `analyze`, `monitor`, and `execute` are implemented in the local job controller.

When users execute a tool in In-VIGO, they may input various parameters, request different resources and optionally set different performance expectations (e.g. the execution of the given job should finish in 10 minutes). Resource allocation is accomplished by the interaction between the local resource coordinator and global scheduler. To choose the appropriate resources, it is necessary to know the specific requirements, such as how much memory and what kind of the CPU is needed for the given run. However, the users are usually not able to provide accurate values for these properties. It is the role of the **`predict` function** in the autonomic manager to estimate the resource utilization of any given job, and hence free the user from the responsibilities.

Resource usage prediction has been extensively examined in the past (e.g. [5][6][7] and [21]). In our case, one very important factor that must be considered in choosing the prediction method is its overhead since the jobs of interest are relatively short lived. In order to avoid high overhead, the `predict` function is implemented using a memory-based learning

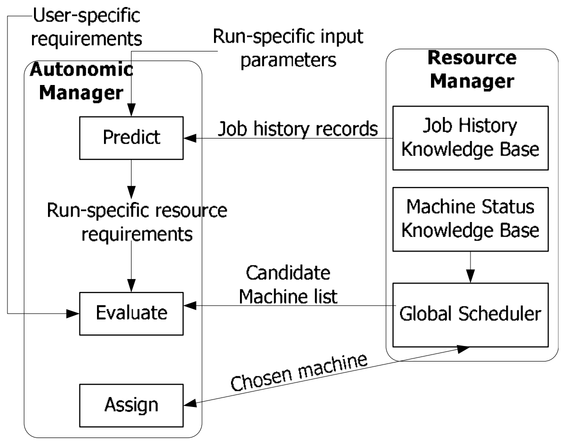
User-specific requirements
Run-specific input parameters
Autonomic Manager
Resource Manager
Predict
Job history records
Job History Knowledge Base
Run-specific resource requirements
Machine Status Knowledge Base
Candidate Machine list
Evaluate
Global Scheduler
Chosen machine
Assign

Figure 4. Information flow between functions in the Autonomic Manager and Resource Manager.

algorithm. In [6] three learning algorithms are evaluated, and the results indicate the nearest-neighbor algorithm is the most efficient one. This algorithm is used in the prediction function of the resource coordinator. The basic idea behind this algorithm is reviewed in the next paragraph (see [6] for a longer discussion and possible improvements).

The resources consumed by a particular run often depend on the input parameters supplied to the tool. Therefore, the "similarity" of two runs is defined by the distance metric of two sets of input values. The distance between a given query point and a data-point in the input space is computed as follows. For a string-type, exact match is used, and for a value-type, the distance is computed using the standard Euclidean metric. The `predict` function applies the following steps:

1. Given the application tool's name and input vector V, an SQL query statement is created.
2. The SQL query is forwarded to the Global Knowledge Base which stores tool's previous runs history. For every result that satisfies the query, the distance is computed.
3. The nearest neighbor to the given run's input parameters is extracted, and the whole record is selected from the history knowledge base including resource usage such as CPU cycles, memory usage etc.

Figure 4 illustrates the information flow between the `predict`, `evaluate` and `assign` functions. After the `predict` function retrieves the resource usage for the particular run, the resource coordinator queries the global scheduler for resources based on the predicted resource usage and user-specific requirements (e.g a Linux RedHat 5.0 system with at least 100 MB of memory). If there are resources satisfying the requirements, the global scheduler puts them in a list and also fetches their properties such as CPU speeds, and dynamic status (e.g. current load etc.) which are updated periodically by the resource monitor daemons (Figure 3.). This list of resources along with their properties is then sent back to the local resource coordinator.

After receiving the candidate machine list, the **`evaluate` function** determines which resource is more preferable based on the resource usage prediction of the given job. For example, if the job is I/O intensive, a resource that is close to the file server is preferable. For CPU-intensive jobs, the goal is to find the resource on which the given job can run fast. However, runtime does not only depend on the machine's CPU speed, but also is strongly related to the machine's load [7][8]. Therefore, the following equations are used to estimate the job's execution on each machine by assuming that the current load of the machine does not change in a relatively short period of time.

$$\text{CPU Time} = \frac{\text{CPU Cycle}}{\text{CPU Speed}}$$

$$\text{Execution Time} = \text{CPU Time} \times (1 + \text{Load})$$

The `evaluate` function computes the predicted runtimes of the machines on the candidate list, puts the best ones in a sorted preference list. The global scheduler then tries to allocate a best available machine for the autonomic VAM according to its preference list beginning from the most preferred one.

After choosing the "best" resource for the job, the **`assign` function** is responsible for submitting the job to the chosen resource, and storing this job's information (the executing machine's name, CPU speed, the submission time, and the predicted CPU time etc.) in the local Knowledge Base. A job sensor is also spawned by the assign function to monitor the job's status on the same machine where the job is launched. Although the resource manager chooses the machine that looks the "best" at the time the job is submitted, the machine's status such as CPU load may change dramatically during the job's execution. The job's status is monitored by the job sensor continuously, which keeps collecting the job's status and reports them to the local Knowledge Base at a constant interval. The job sensor is implemented by a script that measures process's CPU time, elapsed time, CPU percentage, and memory usage using the Unix utility *ps*.

Figure 5 shows how the local job controller works. Since an AVAM may submit multiple jobs on behalf of a user, the job controller keeps track of each job in a hash table. The **`monitor` function** checks every job in the table periodically by fetching the information from the local Knowledge Base. First, the controller checks whether the job is running. If the job finishes

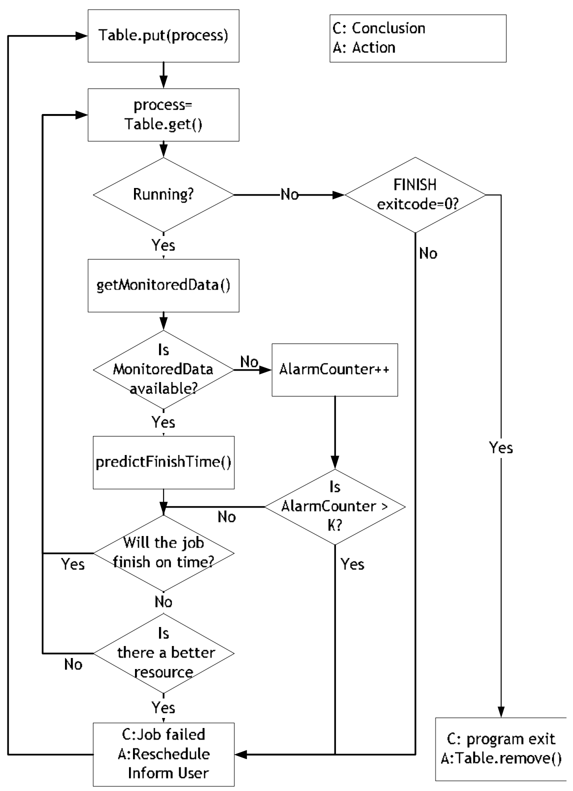

Figure 5. Task graph for job controller.

successfully, the controller collects some statistic data about this execution, e.g. the application's input parameters, performance and resource usage, and reports them to the Global Knowledge Base for historical records. If the job is still running, the controller fetches all the monitored data about the resource and the job to estimate the job's progress based on a general performance model explained in the next paragraphs. If the controller detects any abnormal behavior such as machine failure, job hanging, or intolerable performance, it tries to allocate another resource by querying the global scheduler. The allocation also uses the mechanism described above. If the global scheduler can find a resource job can be rescheduled and still finish before the deadline, then the controller kills the current job and resubmits it to that resource. This subsequent querying of the global scheduler by the rescheduling AVAMs can cause the global scheduler to become a bottleneck. This will depend largely on the likelihood of performance faults in the system. To avoid this bottleneck, the candidate machine list that was obtained at the time of initial allocation of the job, can be reused to find the next best possible resource. As the duration of the jobs under consideration is short, it is likely that this machine list will not be stale, and will reflect the load condition in the system fairly closely. This aspect of the implementation will be further investigated in future work.

For CPU-intensive jobs, how much CPU time is used is largely independent from load. Therefore, CPU time is chosen as the criterion in the controller to measure how jobs progress, given that the values of CPU cycles and CPU speed are available in the local Knowledge Base. The job controller's **`analyze`** **function** uses the following performance model to estimate the job's runtime:

$$\text{Runtime} = \frac{\text{CPU Time}}{\text{CPU Percentage}}$$

where CPU percentage represents the currently CPU percentage consumed by the given run, which is updated periodically by the job sensor.

The analyzer periodically estimates the execution time on the current machine using the following equations (where Current CPUTime, Current ElapsedTime are the CPU time and elapsed time already taken by the job, respectively, and Predicted CPUTime is the total CPU time estimated by the predictor for this run):

$$\begin{aligned}\text{Remaining CPUTime} &= \text{Predicted CPUTime} \\ &\quad - \text{Current CPUTime} \\ \text{Remaining ElapsedTime} &= \frac{\text{Remaining CPUTime}}{\text{CPU Percentage}} \\ \text{Predicted ElapsedTime} &= \text{Current ElapsedTime} \\ &\quad + \text{Remaining ElapsedTime}\end{aligned}$$

If the `analyze` function detects that the monitored job cannot finish before the deadline by comparing the predicted elapsed time with the deadline set by the user, the job controller asks the resource coordinator to allocate a "better" resource that can satisfy the user's goal or bring about benefits due to rescheduling. So the controller looks for a resource that satisfies one of the following conditions:

$$\begin{aligned}&\text{Condition 1:}\\ &\quad \text{Predicted CPUTime} \times (1 + \text{Load})\\ &\qquad + \text{Rescheduling Overhead} < \text{Time Left},\\ &\text{Condition 2:}\\ &\quad \text{Predicted CPUTime} \times (1 + \text{Load})\\ &\qquad + \text{Rescheduling Overhead}\\ &\qquad\quad < \text{Remaining ElaspedTime} \times k,\\ &\text{Time Left} = \text{Deadline} - \text{Current ElapsedTime},\end{aligned}$$

where Load is the load of the new resource, Remaining ElapsedTime is that predicted for the current resource, and k is the rescheduling benefit (k < 1).

The first condition means that if the job is rescheduled on another machine then it can still meet the deadline. The second condition means that the time taken by re-execution and rescheduling of the job on another machine is shorter by a factor of (1-k) than the execution time on the original machine. This condition fits the scenario when, even if rescheduling can not meet the deadline, there are still enough benefits compared with not rescheduling.

If the job sensor fails to update the status of the monitored job, the job controller alarms the system that something is wrong, maybe the machine or the network. When the number of consecutive alarms reaches a threshold, the job is considered failed on that resource, and the controller asks the coordinator to reallocate this job. Another failure condition to be detected is the "hanging" of a job. Under such condition, the job sensor may still be able to collect job's status and report it to the local Knowledge Base. Since it is possible to estimate the job execution time using the above equations, the time threshold to detect job hanging is set to three times that of the predicted execution time.

In order to reduce the overhead caused by querying the large historical data in the Global Knowledge Base, historical records are also cached locally at the AVAM. During an AVAM session, the user is likely to submit "similar" jobs, i.e. the same application with various similar parameters. Hence, from local records, the autonomic manager can predict these jobs' resource usage and performance very quickly. Furthermore, it can mark the resources' qualities based on the previous experiences so that it can prefer the "good" resources and avoid the "bad" resources for future jobs.

4. Evaluation

The AVAM implementation in In-VIGO is evaluated by answering the following questions:

1. Can it efficiently respond to the dynamic resource information and utilize it to achieve the expected execution performance?
2. Can it prevent and recover from job submission and execution failures?

In the following section, the experimental setup used for the evaluation is described.

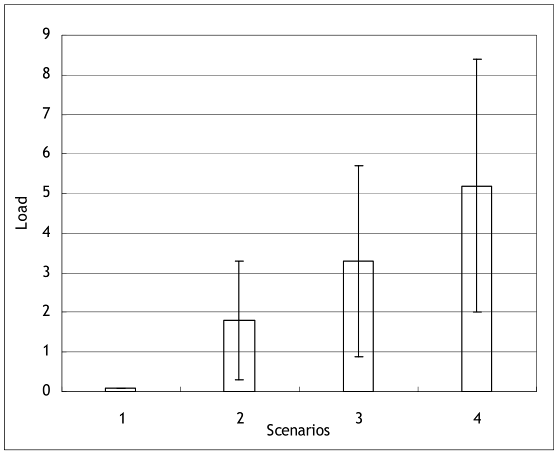

Figure 6. Average and standard deviation of load for the unloaded (1), lightly loaded (2), loaded (3), and heavily loaded (4) resource scenarios.

4.1. Experiment Setup

The evaluation experiments were conducted on a subset of the In-VIGO system. The compute resources consist of three single processor virtual machines (using VMware GSX server 2.5), hosted on a cluster of 32 Xeon 2.4GHz processors with 1GB memory and 18GB disk storage, and a physical machine with dual 927MHz Pentium III processors, 512MB memory and 23GB disk storage, all running Redhat 7.3. TunProb (Numerical Calculation of the Transmission Probability for One-Dimensional Electron Tunneling), a tool on the In-VIGO portal, is used as the workload.

4.1.1 Modeling Non-Dedicated Resources in Grid Environments. For purposes of evaluation of the AVAM approach, background load on non-dedicated resources in a Grid environment is generated by CPU-intensive processes, whose inter-arrival times and runtimes are both modeled by Poisson processes. By changing the ratio of the load application's average runtime to the average inter-arrival time, four different load environments from unloaded to relatively heavy loaded were created (Figure 6). Light load was

Table 1. Table 1. The probability of at least one machine being idle or having a light load in four load scenarios (P1: Probability of at least one machine being idle; P2: Probability of at least one machine having a load below 1; P3: Probability of at least one machine having a load below 2).

Arrial rate (1/s)	Runtime (s)	Avg load	P1/P2/P3 2 machines	P1/P2/P3 16 machines	P1/P2/P3 32machines	P1/P2/P3 256 machines
0	0	0	1/1/1	1/1/1	1/1/1	1/1/1
1/15	14	1.5	0.65/0.96/0.99	1/1/1	1/1/1	1/1/1
1/12	36	3.0	0.13/0.48/0.80	0.60/0.97/1	0.82/1/1	1/1/1
1/9	60	5.3	0.005/0.05/0.13	0.04/0.23/0.64	0.11/0.36/0.79	0.35 /0.97/1

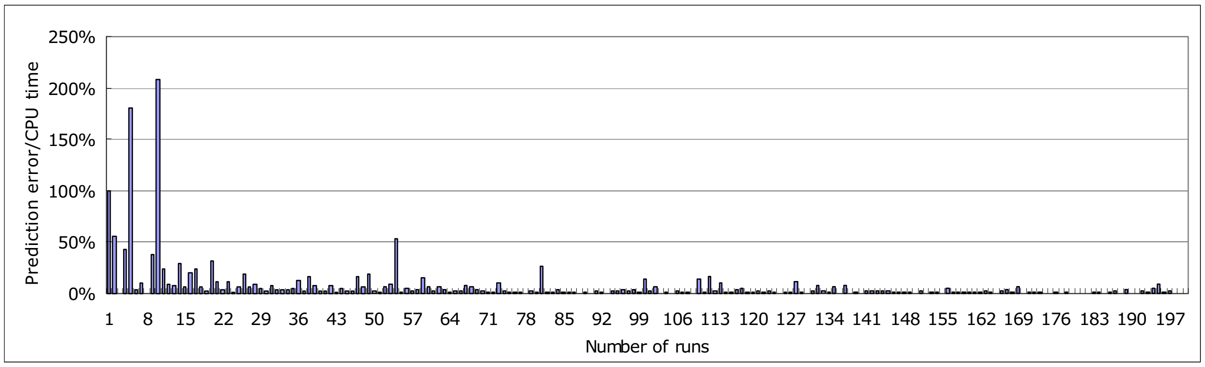

Figure 7. The percentage of CPU time prediction error for NN-algorithm on TunProb.

generated with processes whose average inter-arrival time (15s) is roughly equal to their runtime (14s). The loaded resource ran processes whose average runtime (36s) is three times the average inter-arrival time (12s). The heavily loaded resource ran processes' whose runtime (60s) is about six times the inter-arrival time (9s). Figure 6 shows the averages and standard deviations of the load for the four scenarios. For the last two scenarios, the standard deviations are relatively large, indicating the loaded scenarios are also highly dynamic.

Grid environments may typically have hundreds of non-dedicated resources. The following analysis determines the chances of finding an idle machine in a group of 2, 16, 32 and 256 machines artificially loaded with jobs each lasting a certain period of time and arriving at a rate that follows Poisson processes. The machines' load was measured over a period of time in the four scenarios described above, from unloaded to relatively heavily loaded, and the results reveal that there is a high possibility of at least one machine being idle (see Table 1). While this analysis is simplistic, it supports the intuitive assumption that in a real system with large numbers of machines, at any given time the probability of one machine being idle is high, providing an opportunity to recover from performance faults. On this basis, in order to evaluate the mechanisms proposed in this paper, the non-dedicated resources in a grid system, i.e. In-VIGO, are modeled with two loaded machines and one idle machine.

4.1.2 Modeling the Workload. TunProb is used as an application benchmark representative of CPU-intensive workloads with short execution times. TunProb requires four input parameters, an input file containing data points representing the desired one-dimensional tunneling barrier, minimum and maximum energies and number of energy steps between them for which the transmission probability must be calculated. TunProb is CPU-intensive, and its resource usage depends on three of the parameters, minimum and maximum energy and number of energy steps, which can therefore be used for performance prediction via a nearest-neighbor learning algorithm. The algorithm with the Euclidean distance metric has been evaluated in related work [6] for job execution time prediction of CPU-intensive jobs. Figure 7 shows its prediction accuracy for the TunProb application, executed with the minimum energy, maximum energy and number of energy steps randomly selected from 0 to 10, 0 to 1000, and 0 to 1000000 respectively. The nearest-neighbor algorithm learns fast and the percentage of the CPU time prediction error drops below 15% after a hundred runs. The AVAM implementation allows "plugging in" of per-application performance predictors, and the above results indicate that a nearest-neighbor learning algorithm based on the Euclidean distance metric is suitable for TunProb.

4.1.3 Metrics Used. Three different strategies are used for scheduling of TunProb jobs on the resources. The *round-robin* strategy does not consider the resources' load status and submits the jobs in a round-robin manner to the two loaded machines. *The best candidate without rescheduling* strategy uses dynamic resource information to choose the "best" resource, i.e. the more lightly loaded of the two loaded machines, for the job. It does not however, monitor the job's progress or reschedule it. The *best candidate with rescheduling* strategy initially schedules the job on the "best" resource, monitors its execution behavior, predicts its progress and takes appropriate actions, i.e. reschedules the job to the lightly loaded machine if it suffers from a performance fault. The job QoS requirements managed by Condition 1 and 2 (see Section 3.2) are (a) a job runtime deadline is no larger than 5 times of its runtime on an unloaded machine, and (b) a job is rescheduled if it can improve current performance by at least 30%. The alarm threshold/counter for establishing that a job has failed is set at 6. The AVAM is evaluated by comparing the runtime of jobs with the different scheduler strategies for the four loaded resource scenarios. Execution performance improvement with the *best candidate with*

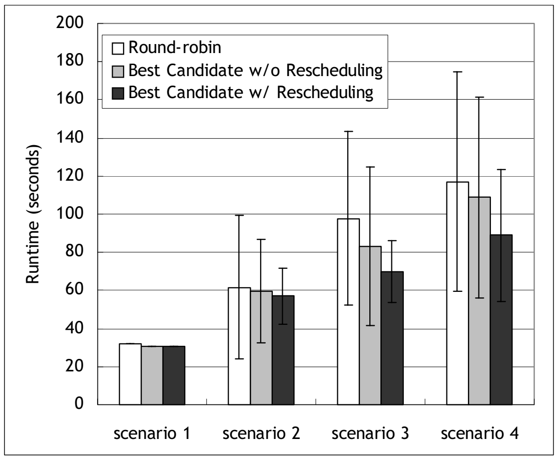

Figure 8. The average and the standard deviation of the execution times with fixed inputs in the four scenarios with three strategies: Round-robin, Best Candidate without Rescheduling and with Rescheduling.

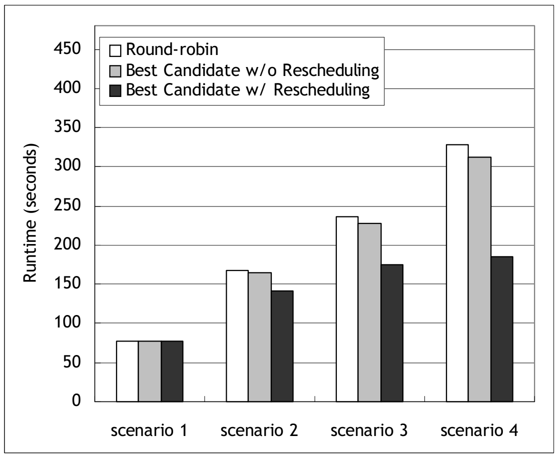

Figure 10. The average execution times with varying inputs in the four scenarios with three strategies: Round-robin, Best Candidate without Rescheduling and with Rescheduling.

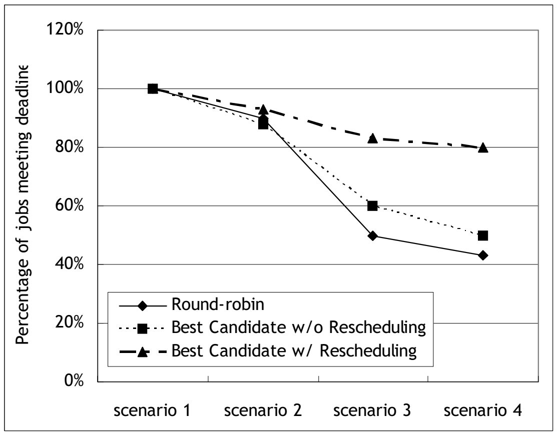

Figure 9. The percentage of jobs with fixed inputs meeting their deadline (set to 100 seconds) in the four scenarios with three strategies: Round-robin, Best Candidate without Rescheduling and with Rescheduling.

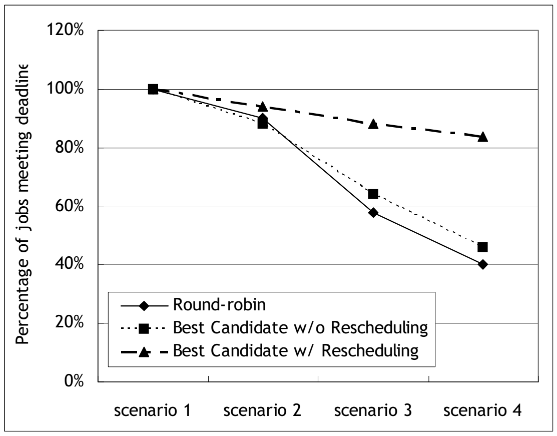

Figure 11. The percentage of jobs with varying inputs meeting their deadline (set to four times of execution time on an unloaded virtual machine) in the four scenarios with three strategies: Round-robin, Best Candidate without Rescheduling and with Rescheduling.

rescheduling strategy measures the performance improvement due to the AVAM. Performance improvement that is due to the triggering of Condition 2 or the alarm counter, measures AVAM's ability to recover from job submission failures.

4.2. Experiment Results

4.2.1 Performance Improvement. In the first set of the experiments, the inputs values of the benchmark application are fixed as follows: the energy step is 100000, the minimum energy is 2 and the maximum energy is 200. The average runtime of the benchmark with these input parameters is about 20s on the unloaded virtual machines and about 40s on the unloaded physical machine. Runtime increases approximately linearly with the load. The job runtime deadline is set to be 100s. At job submission time, the two virtual machines with the background load are available as candidate resources for the scheduler. After the job submission, the idle physical machine also becomes available as a candidate resource for rescheduling.

For each load scenario, fifty runs of TunProb were submitted to the test bed continuously. Figure 8 shows the average runtimes and the standard deviations of

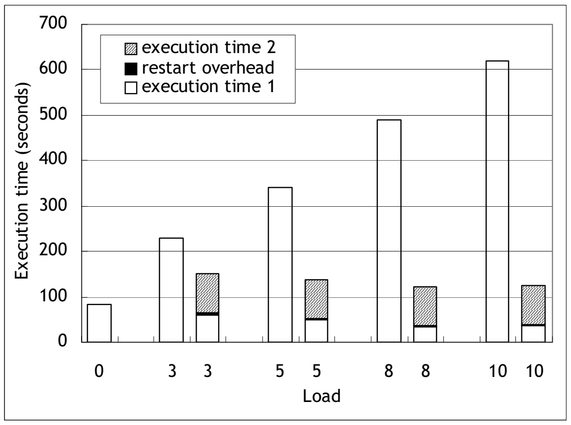

Figure 12. The execution time of TunProb with different background load. Execution time 1 is the runtime on the machine where the job is submitted; execution time 2 is the runtime on the machine where the job is rescheduled.

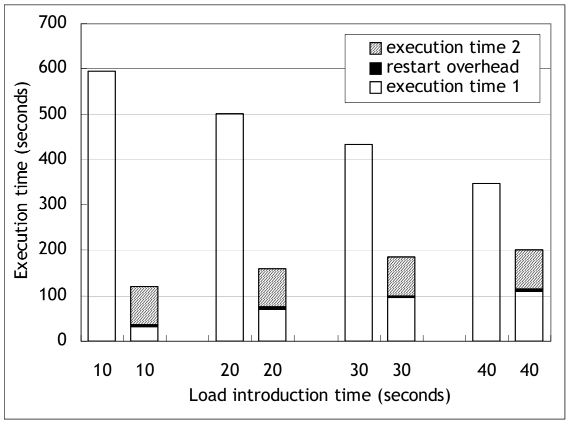

Figure 13. The execution time of TunProb with different load introduction time. Execution time 1 is the runtime on the machine where the job is submitted; execution time 2 is the runtime on the machine where the job is rescheduled.

executing the benchmark with the different strategies. Figure 9 indicates the percentages of the jobs that meet the deadline (of 100s) for the three strategies. It can be observed that when the load on the resources is light, all strategies work well because most of the resources can satisfy the jobs' requirements. When the load becomes heavier and more dynamic, the best candidate strategies perform much better than the Round-robin strategy, while the one with rescheduling substantially outperforms the one without rescheduling. The reasons behind the observations are as follows: first, the best candidate strategies can always find more appropriate resources for the jobs than the "blind" Round-robin; second, even though the chosen best candidate seems to be the "best" resource for the job at the time of submission, it may not be the "best" during the job's execution due to the highly dynamic computing environment (so rescheduling with relatively low overhead can improve the job's execution time and is more likely to meet the expected deadline). In the three loaded scenarios with rescheduling, 6%, 12%, and 18% of the jobs improve performance but do not meet the deadline with rescheduling. The rescheduling of these jobs is triggered by Condition 2, i.e. jobs are rescheduled because they can improve performance by at least 30%, indicating that the AVAM can handle job submission failure using this technique.

Effect of application CPU time prediction errors from the nearest-neighbor learning algorithm on performance is avoided in the above experiments by using fixed input parameters for the benchmark. This allows us to evaluate the rescheduling framework when the application performance modeling only takes into account errors in application runtime prediction for varying load. To take the CPU time prediction errors into account, the above set of experiments was repeated, while varying the input parameters (energy step, minimum energy and maximum energy) of the application by randomly selecting their values from uniformly distributed ranges of 0 to 1000000, 0 to 10, and 0 to 1000, respectively. Three virtual machines are used for this experiment, in which two of them are loaded with background processes. The job runtime deadline is set to be four times the job's execution time on an unloaded machine. Before the jobs' submissions, 100 runs of TunProb with random input parameters in the same above-mentioned range are used to warm the predictor. All the jobs are submitted to the two loaded virtual machines, and the third one is used for rescheduling. Figure 10 and Figure 11 show the average runtimes and the percentage of the jobs that meet the deadline for the three strategies. Performance of the three scheduling strategies is similar to the first set of experiments, reflecting the fact that small prediction errors have little effect on the efficiency of rescheduling.

4.2.2 Sensitivity to Load in the System. The third set of experiments investigates how efficiently the system responds to load changes. A TunProb job (with the following inputs: energy step of 500000, minimum energy of 2 and maximum energy of 200) was submitted to an unloaded machine. The job's average runtime on the unloaded machine was measured to be 80 seconds, so the deadline for job completion was set to 3 minutes. Various amount of background load was introduced into the machine 20 seconds after the job's submission. Figure 12 shows the benchmark application execution times with and without rescheduling under different levels of load. The horizontal axis represents the amount of load introduced in the machine, while the vertical axis is the job's runtime. The left bar indicates the execution time when the job continues to run on the machine after the load is introduced. The right bar represents the job's execution managed by the AVAM, which reschedules the job to a better resource because of failure or poor performance. From the figure, it can be seen that the greater the load introduced the quicker the system is at detecting and reacting to it. The reason for this is that when the load increases to a very high value (i.e. the load is increased to above 5 in the experiments), the job sensor fails to update the job status on time and thus the controller turns the alarm on. After several alarms (a threshold value of 6 was used in the experiments) the AVAM regards the job as failed and reschedules this job to another machine immediately. Otherwise, the manager utilizes the monitored data from the job sensor to estimate whether the job would finish before its deadline. The figure also shows the overhead for rescheduling the job, which averages less than 4 seconds in the experiments. Compared with the total execution time, this is insignificant for most cases.

In the fourth set of experiments, after the job is submitted to an unloaded machine, the same amount of high load is introduced, but at different points during the benchmark's execution. The input parameters for the benchmark are the same as for the third experiment. In Figure 13, the horizontal-axis indicates the elapsed time in seconds of the job's execution when the load is introduced, while the vertical axis is the job's runtime. The gain for rescheduling diminishes as the load is introduced later into the job execution. The AVAM recognizes this scenario and avoids unnecessary rescheduling of the application.

5. Related Work

The existing work most closely related to the system described in this paper falls into three categories: resource-usage prediction, resource allocation, and rescheduling.

Topics on resource-usage prediction have been examined by [6][7][8][9]. Most of these approahches rely on the use of past-execution-time knowledge. This paper's approach uses the predictive application modeling developed by [6] because of its efficiency and its consideration of the applications' input parameters. This information is available for In-VIGO tools such as the one (TunProb) uased as a benchmark in this paper. The prediction model employs a local learning algorithm for the prediction of run-specific resource usage on the basis of run-time input parameters supplied to tools. This model works well for our case.

Researchers have proposed a number of systems or approaches for resource selection in dynamic, heterogeneous computing environments. Some projects [13][14] aimed to support custom-specific systems whose users' specifications can be directly used by a scheduler. In contrast, the system described in this paper provides applications with a self-learning ability to predict their resource requirements at run-time. The Application-Level Scheduling Project (AppLes) [15] has developed an approach that incorporates static and dynamic resource information, performance predictions into resource scheduling. However, their performance models are application-specific and not easy to re-apply to new applications. Our performance prediction approach is based on a generic model that can be applied to any application.

Several projects have implemented rescheduling execution of applications to different resources. These systems either aim to use under-utilized resources, to provide fault resilience, or to reduce obtrusiveness into workstations. The most related to this paper is [12]. Their system continuously monitors applications and evaluates the remaining execution time of the application, and migration decisions are made whenever the applications are not making sufficient progress. The two main differences between their approach and this paper's solution reside in the evaluation model and the rescheduling action. Their evaluation model also depends on application-specific performance models and the overhead caused by migration including reading and writing checkpoints is much higher than restarting a job. High-overhead migration is not feasible for the relatively short-lived applications considered in this paper

6. Conclusions and Future Work

This paper describes an application-centric approach to the implementation of an autonomic middleware component that adapts the execution of applications to dynamically varying resources to achieve fault tolerance and improved performance. The prototype, the AVAM component, implemented in In-VIGO was evaluated and shown to be effective for managing CPU-intensive jobs with short execution times.

Future work will evaluate performance-prediction models in the context of applications with different characteristics, i.e. I/O intensive, and hybrid applications. Another direction of investigation, is the extension of the approach described in this paper to manage and optimize long running jobs. In this scenario, there is a need for checkpointing and migration of jobs (instead of simply restarting them) in order to not waste significant amounts of compute cycles consumed before the rescheduling decision. While there are several related works on adapting applications to resources via checkpointing and migration, virtualization techniques (virtual machines and file system) incorporated at the resource layers of

In-VIGO offer a basis for a different approach. A job that is checkpointed at regular intervals can be viewed as a sequence of short jobs (of duration consisting of the checkpoint interval) that are initiated from checkpoint state. The autonomic job management framework and its evaluation for short jobs, presented in this paper, are thus directly applicable to the case of long running jobs. Finally, the proposed application-centric approach can be applied to self-manage components of the In-VIGO middleware, such as the AVAM components, to achieve fault-tolerance and improved performance.

7. Acknowledgements

This work was supported in part by the National Science Foundation under Grants No. EIA-9975275, EIA-0224442, ACI-0219925, EEC-0228390, SCI-0438246 and NSF Middleware Initiative (NMI) collaborative grants ANI-0301108/ANI-0222828 and SCI-0438246, and by the Army Research Office Defense University Research Initiative in Nanotechnology. The authors also acknowledge two SUR grants from IBM and a gift from VMware Corporation. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, Army Research Office, IBM, or VMware.

8. References

[1] Steve R. White, James E. Hanson, Ian Whalley, David M. Chess, Jeffrey O. Kephart, "An Architectural Approach to Autonomic Computing", *International Conference on Autonomic Computing (ICAC'04),* May 17 - 18, 2004.

[2] J. O. Kephart and D. M. Chess, "The vision of autonomic computing", *Computer*, 36(1):41--52, 2003.

[3] Sumalatha Adabala, Vineet Chadha, Puneet Chawla, Renato Figueiredo, José A. B. Fortes, Ivan Krsul, Andrea Matsunaga, Mauricio Tsugawa, Jian Zhang, Ming Zhao, Liping Zhu, Xiaomin Zhu, "From Virtualized Resources to Virtual Computing Grids: The In-VIGO System", In *Future Generation Computing Systems*, 04/2004.

[4] Ian Foster, Carl Kesselman, and Steven Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations", *International Journal Supercomputer Applications*, vol. 15, no. 3, 2001.

[5] I-Hsin Chung, Jeffrey K. Hollingsworth, "Using Information from Prior Runs to Improve Automated Tuning Systems", *Proceedings of the ACM/IEEE SC2004 Conference*, November 06 - 12, 2004.

[6] Nirav H. Kapadia, Jose' A. B. Fortes, and Carla E. Brodley, "Predictive Application-Performance Modeling in a Computational Grid Environment," *Eighth IEEE International Symposium on High Performance Distributed Computing*, August 1999.

[7] P. Dinda, "The Statistical Properties of Host Load", *Scientific Programming*, 7:3-4, Fall, 1999.

[8] P. Dinda, D. O'Hallaron, "Host Load Prediction Using Linear Models", *Cluster Computin*g, Volume 3, Number 4, 2000.

[9] M. V. Devarakonda, and R. K. Iyer, "Predictability of Process Resource Usage: A Measurement-Based Study on UNIX," *IEEE Trans. Software Engineering*, pp. 1579-1586, 1989.

[10] R. Wolski, N. Spring, and J. Hayes, "Predicting the CPU Availability of Time-shared Unix Systems", *In Peoceedings of 8th IEEE High Performance Distributed Computing Conference (HPDC8)*, August 1999.

[11] Holly Dail, Henri Casanova, and Francine Berman, "A Modular Scheduling Approach for Grid Application Development Environments", *Journal of Parallel and Distributed Computing*, June 2002.

[12] Vadhiyar, S. Dongarra, J., "A Performance Oriented Migration Framework for the Grid," *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid*, Tokyo, Japan, pp. 130-137, May 12-15, 2003.

[13] David Oppenheimer, Jeannie Albrecht, David Patterson, and Amin Vahdat, "Distributed resource discovery on PlanetLab with SWORD", *First Workshop on Real, Large Distributed Systems (WORLDS '04)*, December 2004.

[14] Rajesh Raman, Miron Livny, and Marvin Solomon, "Matchmaking: Distributed Resource Management for High Throughput Computing", *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, July 28-31, 1998, Chicago, IL.

[15] Francine Berman, Richard Wolski, Henri Casanova, and Walfredo Cirne, "Adaptive Computing on the Grid Using AppLes", *In IEEE Transactions On Parallel and Distributed Systems*, volume 14, pages 369--382, April 2003.

[16] N. Kapadia, R. Figueiredo, and J. Fortes, "PUNCH: WebPortal for Running Tools", *IEEE Micro, 20(3), pp 38-47,* May/June 2000.

[17] M. Livny and D. Thain, "Caveat emptor: Making grid services dependable from the client side". *In 2002 Pacific Rim International Symposium on Dependable Computing (PRDC '02),* December 2002.

[18] D. Thain and M. Livny, "Building reliable clients and servers", In I. Foster and C. Kesselman, editors, The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, 2003.

[19] R. Arpaci-Dusseau and A. Arpaci-Dusseau, "Fail-stutter Fault Tolerance", *In Workshop on Hot Topics in Operating Systems,* May 2001.

[20] R. Figueiredo, N. Kapadia, and J. A. B. Fortes. "The PUNCH Virtual File System: Seamless Access to Decentralized Storage Services in a Computational Grid", *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC),* August 2001.

[21] Valerie Taylor, Xingfu Wu, Jonathan Geisler, and Rick Stevens, "Using Kernel Couplings to Predict Parallel Application", *Proc. of the 11th IEEE International Symposium on High-Performance Di*stributed *Computing (HPDC 2002)*, Edinburgh, Scotland, July 24-26, 2002.