Optimal Load Balancing in a Beowulf Cluster

by

Daniel Alan Adams

A Thesis

Submitted to the Faculty

of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

May 2005

APPROVED:

Dr. David Finkel, Major Advisor

Dr. Michael Gennert, Head of Department

Abstract

PANTS (PANTS Application Node Transparency System) is a suite of programs designed to add transparent load balancing to a Beowulf cluster so that processes are transfered among the nodes of the cluster to improve performance. PANTS provides the option of using one of several different load balancing policies, each having a different approach. This paper studies the scalability and performance of these policies on large clusters and under various workloads. We measure the performance of our policies on our current cluster, and use that performance data to build simulations to test the performance of the policies in larger clusters and under differing workloads. Two policies, one deterministic and one non-deterministic, are presented which offer optimal steady-state performance. We also present best practices and discuss the major challenges of load balancing policy design.

Table of Contents

1 Introduction

PANTS is an approach to load balancing in a Beowulf cluster (Claypool and Finkel, 2002). A Beowulf cluster is a cluster of off-the-shelf computers, called nodes, running an open source operating system, such as Linux, connected via a standard network, such as a 100 or 1000 Mbps local area networks (Beowulf Web Site, 2005).

The advantage of using a Beowulf cluster is achieving the high performance of a multiprocessor computer at moderate cost. In addition, increased stability and flexibility are provided as nodes can be added or removed without major system reconfiguration.

A disadvantage of using a Beowulf cluster can be found in distributing the work. Typically, software running on a Beowulf cluster distributes work by dividing a problem into subproblems to be executed in parallel on different nodes of the cluster. However, the software must be explicitly designed to distribute work and collect the results. The programmer who starts with a non-distributed program faces a significant task to convert it into a distributed program to run on a Beowulf cluster, including designing a load-balancing algorithm and modifying the source code to distribute the work. Several toolkits are available for creating distributed applications, including PVM (PVM Web Site, 2005) and MPI (MPI Web Site, 2005).

The goal of the PANTS project is to mitigate the disadvantages of using a Beowulf cluster by transparently performing load balancing and relieving the programmer of the necessity of modifying the program's source code (Claypool and Finkel, 2002). PANTS accomplishes this goal by providing background processes that communicate with each other and by intercepting operating system calls that initiate program execution. Load balancing is performed by different policies or load balancing algorithms which can be selected at run-time.

Our cluster consists of 16 computers, running the Fedora Core 2 Linux distribution, connected by a 100 Mbps LAN. We used this cluster to run a series of performance tests and collected data on overall processing times as well as detailed measurements of the operation of the different policies. This data was used to parameterize a simulation model of the system. After the simulation model was validated against the operation of the current cluster, experiments were run to test the performance of the load balancing polices in larger clusters and with varying workloads.

We will outline the development process of the current policies and so present the behaviors common to each good policy. Building on this we will present best practices in designing load balancing policies and discuss the major challenges in policy design and implementation.

2 Load Balancing Policies

There are currently seven load balancing policies provided by PANTS: Leader, Timeline, Threshold, Random, LRU, Round Robin, and NFS. Each policy takes a different approach to distributing processes. Some policies, however, are very similar and one policy is often a variation of another modified to change some key aspect.

In several of these policies, nodes in the system are classified as heavily loaded or lightly loaded depending on their current workload. Previous research has investigated methods for measuring load and classifying nodes as heavily loaded or lightly loaded (Lemaire and Nichols, 2002). In these policies, when a process is about to be initiated on a heavily loaded node, the policy attempts to identify a lightly loaded node to which the work can be transferred

2.1 Leader and Random Policy

The Leader policy was the first policy implemented (Claypool and Finkel, 2002). In this policy, one node, called the leader, keeps track of the status of the nodes and receives requests to distribute work. The use of the leader node created a bottleneck in the cluster which decreased system performance and stability, even in a small cluster (Adams and LaFleur, 2004).

The Random policy is similar to the Leader policy in that they both use random selection to select from a list of lightly loaded nodes (Adams and LaFleur, 2004). A key difference is that in Random, each node maintains its own list which is updated by nodes broadcasting their status. In this way, the Random policy is decentralized and the role of the leader node eliminated. Similar policies can be found in systems such as openMosix which distributes work randomly based on partial node status knowledge (openMosix Web Site, 2005).

2.2 NFS Policy

The NFS policy is similar to the Random policy and uses files in a distributed filesystem, currently NFS, which is accessible to all the nodes in the system (Adams, 2004). There are two directories, the available (lightly loaded) and unavailable (heavily loaded) directories. Each file in these directories is given a filename that is the IP address of a node. When a work request is received, a file is randomly selected from the available directory and the work is transferred to that node. If the available directory is empty, then a node is selected from the unavailable directory. The NFS policy may perform poorly in

a large cluster as accessing these shared files may create a bottleneck. Conversely, performance may increase in a larger cluster due to the random node selection.

2.3 Timeline and Threshold Policies

The Timeline and Threshold policies are similar to one another (Adams and LaFleur, 2004). In these policies, when a node is needed for a work request, a message is broadcast to a multicast address listened to by the lightly loaded nodes. The first node to respond performs the work. The Timeline policy is very basic and does not provide any optimizations. The disadvantage of this approach is that high network load may be created in large clusters when there are many lightly loaded nodes. The Threshold policy provides an approach to reducing the number of unnecessary responses sent. While currently supported, these policies are being deprecated and thus will not be discussed further.

2.4 LRU and Round Robin Policies

In the Round Robin policy, a list of all nodes sorted by IP address, including both available and unavailable nodes, is maintained and the policy stores a pointer to the next node in the list. When a request is received, the node currently pointed to in the list is returned and the pointer is incremented to the next node in the list. After the last node in the list is returned, the pointer is set to the first node in the list. In this way, the policy continually cycles through the list of nodes, disregarding node status. The motivation behind this is that, in theory, the load will be balanced over the cluster by this cycling through the nodes. It is meant to address a problem common to all of the random selections policies that often a node will be selected two or more times in a row. The solution in this policy is to ensure that once a node receives a work request, it cannot receive another until all other nodes do also.

An expected behavior of this policy is that it performs well when jobs are of equal size, but performs poorly the more jobs sizes vary. This is due to the policy's simplicity in traversing the list; a single work request of an unusual size should upset the order of node selection to reflect the size of the job but it does not. For instance, consider a cluster of five nodes and a test, such as the render test, where jobs are sent out five at a time. If all jobs are of equal size they are correctly sent to nodes 1, 2, 3, 4, 5 and then 1 again once the job on node 1 finishes. In this way, each node only processes one job at a time. Suppose, however, that the third job is abnormally large so that by the second time node 3 is reached in the list it is still processing the third job. Optimally, the policy would skip node 3 and send the work to the next free node in the list. This is not the case, however, and node 3 will be sent the job making the load in the cluster unbalanced. The Round

Robin policy is implemented in the simulation only for the purpose of illustrating the performance of the LRU policy.

The LRU policy is a modification of the Round Robin policy in which the list of nodes is sorted by the number of currently executing jobs on each node and then by the last time each node was sent a job. This corrects the problem given above and ensures that, for instance, a node currently executing two jobs will not be sent another job unless all other nodes are also executing two jobs. Additionally, tracking the last time a node was sent a job ensures, as in the Round Robin policy, that if all nodes are executing the same number of jobs then once a node receives a job it cannot receive another until all other nodes have also received jobs. These two modifications should ensure improved performance despite job size.

An expected possible disadvantage common to both policies is that they are not designed to handle multiple originating nodes. This is the originating of work requests from more than one node. It is possible that since the lists and node status are not shared between nodes that this could result in a non-optimal distribution of work. Possible implementations of a solution to this will be presented later.

3 Performance Tests

Two different tests were used to test the performance of both the existing system and the simulation, bigsum and the render test. These tests also serve as a means to validate the simulation. Each test is designed to ascertain the system's, and each policy's, ability to operate in different environments.

3.1 Bigsum

The bigsum test computes the sum of a range of numbers by dividing a given range into a number of segments, distributing these segments to the cluster, and recombining the results to get the final sum (Adams and LaFleur, 2004). The purpose is to test the systems ability to handle a sudden flood of work requests as the segments, each being a work request, enter the system at once. A library enabling the use of large integers allows the test to compute large sums above the limit set by the architecture.

While not very interesting in itself, this test simulates the behavior of a number of applications, particularly those in scientific computing. Often to parallelize a computation the processing is separated into sub-parts, distributed at once, and then recombined at the end for the final result.

To run the test, one must specify the range of numbers to sum as well as the number of

processes to divide the range into. A set of scripts and programs then use PANTS to distribute the computation and collect and display the result. Normally, the number of processes is simply the number of nodes in the cluster to ensure the possibility of an optimal distribution of work.

3.2 Distributed Animation Rendering

The render test measures the system's ability to handle a possibly indefinite stream of work requests. An animation is divided into a number of work requests, each being one or more frames to render. A number of work requests are distributed at once initially, similar to the bigsum test. However, each time a work request is completed the next in the sequence is sent out until there are no remaining work requests. After the test is completed the rendered frames can be combined or processed into the final animation.

While there is some similarity to the bigsum test, the heart of this test is the system's having to distribute work requests while any number of nodes are heavily loaded. Additionally, as the length of the test increases the impact of the initial work requests decreases. This is a much better test of system's capabilities than the bigsum test which is why it is used here more often and several variations of it are also used.

4 Simulation

4.1 Background

The simulations are performed using an object-oriented discrete event simulator written by the author in the Java programming language. Most of the selection from statistical distributions was provide by the COLT library, although others such as the triangular and log normal distributions were written manually (Colt Web Site, 2005). The existing cluster and a set of performance tests were used to collect data on processing times as well as benchmarks of the policy internals. These were then used as parameters to create the simulation.

We ran two sets of simulation experiments. The first compares the simulation behavior to that of the existing system in order to validate that the simulation accurately characterizes the system’s behavior. The second set uses the simulation to examine how the system performs with larger cluser sizes and under various load conditions. All simulation experiments were run with a minimum of 100 replications.

4.2 Validation

To validate the simulation against the existing system the following tests were used; bigsum 1 to 200,000, bigsum 1 to 1,000,000, rendering 30 frames, and rendering 100 frames. The bigsum 1 to 200,000 test results are shown in Table 1 comparing the performance of the existing system and the simulation. The error, or difference between the simulation and existing system, in most cases for the LRU simulation is less than 1%. Although not as tight as with LRU, the simulation shows strong agreement in behavior with the system for the Random and NFS policies. The reason for this is that LRU is much more deterministic in its behavior as it does not use randomization. Additionally, there are internal differences between the random number generators used in the system and the simulation which have a impact on the performance due to its effect on node selection.

	5 Nodes		10 Nodes		15 Nodes	
	Time (s)	Error	Time (s)	Error	Time (s)	Error
LRU	34.84		18.44		13.12	
LRU Simulation	34.79	0.14%	13.12	0.46%	11.42	0.67%
NFS	73.31		42.92		32.15	
NFS Simulation	68.41	6.69%	32.15	1.58%	33.41	3.94%
Random	76.49		45.66		35.58	
Random Simulation	80.55	5.32%	35.58	6.58%	36.31	2.05%

Table 1

The bigsum 1 to 1,000,000 test results are summarized in Table 2. As in the previous test, the LRU policy has a much lower error percentage than the Random and NFS policies yet all policies show similar behavior in the simulation.

	10 Nodes		15 Nodes	
	Time(s)	% Error	Time(s)	% Error
LRU	341.889		239.190	
LRU Simulation	338.23	1.07%	220.34	0.88%
NFS	518.44		400.67	
NFS Simulation	588.82	13.57%	420.22	4.97%
Random	629.258		445.174	
Random Simulation	622.68	1.05%	432.85	2.77%

Table 2

Table 3 shows the 30 frame render test results for the simulation and the system. Again we see a low error percentage for the LRU policy and a similar error percentages for the Random and NFS policy to that of the first bigsum test.

	5 Nodes		10 Nodes		15 Nodes	
	Time (s)	Error	Time (s)	Error	Time (s)	Error
LRU	203.63		103.23		70.63	
LRU Simulation	202.83	0.39%	102.38	0.82%	69.32	1.85%
NFS	318.82		198.77		158.99	
NFS Simulation	301.06	5.57%	196.19	1.30%	162.42	2.16%
Random	400.38		226.57		175.06	
Random Simulation	364.73	8.90%	214.88	5.16%	169.05	3.43%

Table 3

Table 4 summarizes the results of the 100 frame render test. As with the other tests, the LRU policy shows a low error percentage. The Random and NFS policies again display strong agreement.

	10 Nodes		15 Nodes	
	Time(s)	% Error	Time(s)	% Error
LRU	85.777		58.482	
LRU Simulation	86.002	0.26%	58.456	0.04%
NFS	226.305		164.204	
NFS Simulation	209.554	7.40%	159.862	2.64%
Random	209.536		164.051	
Random Simulation	229.624	9.59%	171.285	4.41%

Table 4

When selecting random numbers, a node will often be chosen more than once in a row. In the render test of 100 frames and 10 nodes, for instance, 23.016% of the 200,000 node selections in the simulation replications were repeats. This problem of random selection will be addressed more fully later when discussing the simulation performance results of larger clusters. To help reduce the performance loss due to this the NFS policy tracks the last node selected and will not allow the same node to selected twice in a row. The combination of this optimization and the NFS policy's second list of nodes result in error that differs from that of the Random policy.

These tests show that there is excellent agreement between the results of the existing system and those of the simulation. This validates our simulation in that it accurately represents the behavior of the existing system.

5 Simulation Results

5.1 Bigsum

We will now look at the test results to predict the performance in larger clusters. Efficiency here is measured as policy performance compared to ideal, or the lowest possible time to execute the work, given a certain cluster size. For n nodes, ideal performance is calculated as the time required to run on one node divided by n.

Figure 1 shows the results of a bigsum 1 to 1,000,000 test using clusters from 10 to 100 nodes. There tests show that all policies decreased in performance as cluster size increased. The LRU and Round Robin policies decreased from 94.8% to 48.8% efficency and NFS and Random decreased from 39.2% to 21.9% efficiency. This is due to the fact that, since the number of work requests do not also increase, the amount of work per node decreases which increases the impact of overhead on performance. One may also note that the lines of the LRU and Round policies are indistinguishable. This is due to the fact that, in situations where work request size is constant, they are essentially the same policy and will be discussed more fully with the test of varying work request sizes.

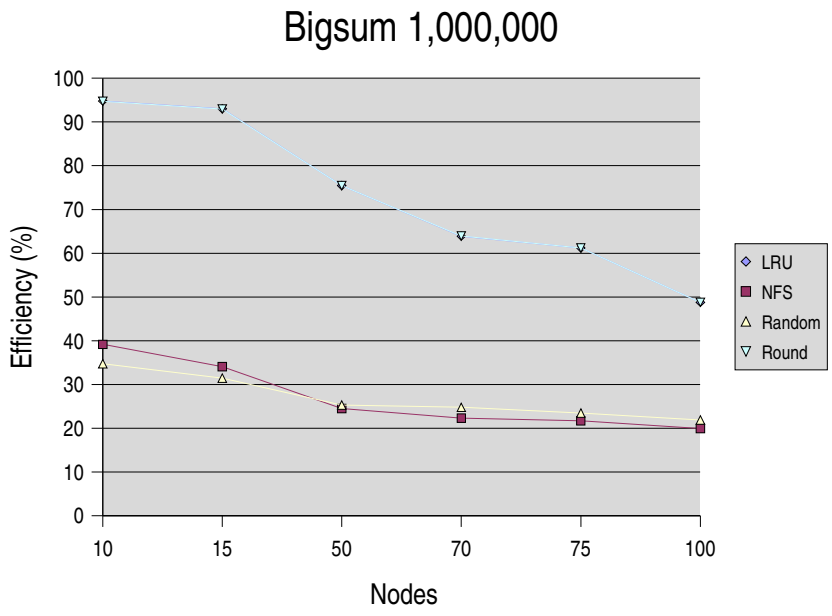

Figure 1

Another important result is that the LRU policy achieves a maximum 94.81% efficiency while the maximum efficiency of a random selection policy here is 39.18%. In this test the LRU policy selects each node exactly once every time which is an optimal node selection. The random selection policies, however, always select at least one node more

than once meaning that some nodes will have two or more work requests and others will have none due to random selection. Predictably, when the number of nodes increases the number of repeat selections decrease. While the LRU policy decreases 46% in efficiency the random selection policies decrease approximately 12%-19%. As the cluster size increases the number repetitions decreases which in turn, due to the performance increase, reduces the performance loss incurred by increased overhead.
From this test we can see that random selection policies do not perform well in tests such as this where all node selection is done at the start, yielding performance lower than that of the LRU policy.

5.2 Distributed Animation Rendering

The results of a test rendering 100 frames with cluster sizes increasing from 10 to 100 nodes are shown in Figure 2 and are similar to those found in the previous bigsum test; as the cluster size increases the work per node and performance decreases. Efficiency of the LRU policy decreased from 97.71% to 71.04% and the random selection policies decreased from 55.97% to 29.32%. Conversely, a test shown in Figure in which the cluster size is fixed at 100 nodes and the frames rendered ranges from 100 to 10,000 frames shows predictably reverse results; as the number of frames increases the impact of overhead decreases, thus increasing performance. The LRU policy's efficiency increased from 71.20% to 98.07% and the random selection policies' performance increased from 29.66% to 74.3%.

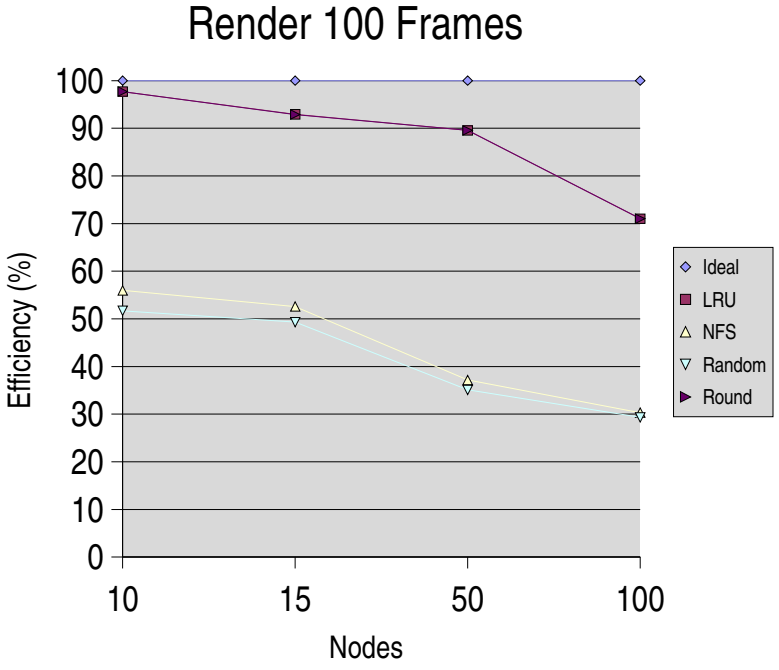

Figure 2

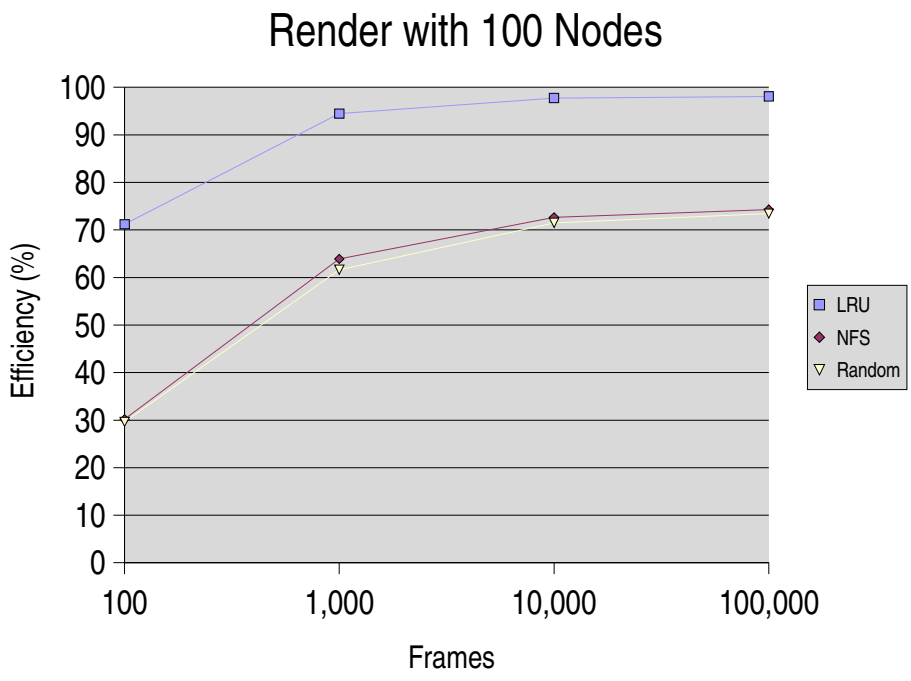

Figure 3

Figure 4 shows the results of increasing work and cluster size simultaneously. The number of frames is eight times the number of nodes in each case. Although it appears that the performance of LRU decreases, this is due to an initial period in which the first set of work requests are sent out and an ending period in which the last set of requests are completing. These periods are essentially overhead and increase as the number of nodes increases, thus reducing the calculated performance. Figure 5 shows the ratio of busy nodes through a run of the test with 1,600 frames and 200 nodes. Between the initial and end periods, the LRU policy operates at 100% cluster utilization.

Despite the initial and end periods, the random selection policies show an increase in performance. This indicates that as the cluster size increases the node selection errors due to random selection are decreased thus increasing performance. In fact, in this test the NFS policy shows a steady state performance, the performance between initial and end periods, of approximately 80%, much higher than the calculated overal performance of 63.6%.

One other interesting feature of Figure 5 is the difference in slope in the end period of the random selection policies compared to LRU and Round. The sharp, constant slope of the LRU policy represents the policy's not being forced to wait for remaining jobs to finish. Conversely, the more gradual, jagged slope of the random selection policies denotes this occurring.

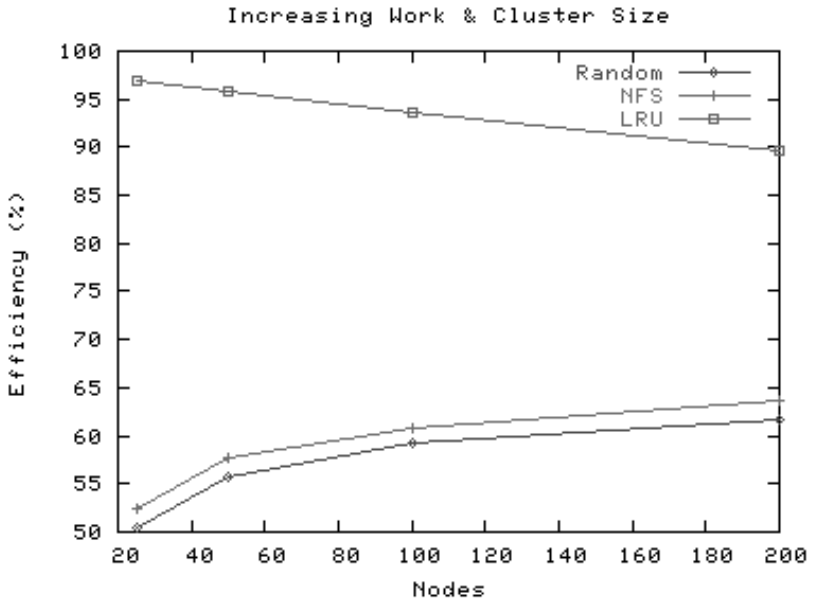

Figure 4

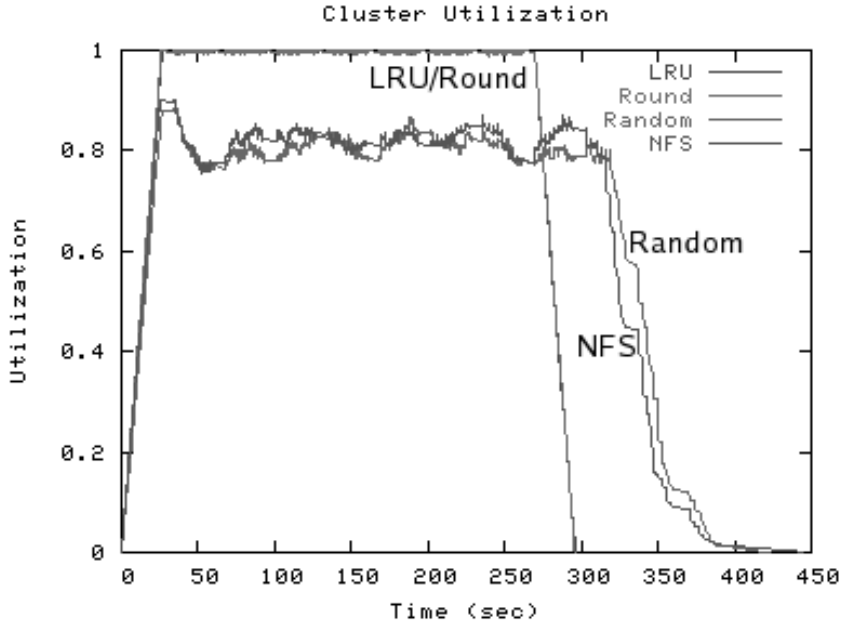

Figure 5

Due to the flexibility and nature of the render test, it will be the used for all further experiments.

5.3 Multiple Originating Nodes

Another situation we wished to test was one in which there were multiple originating nodes, or nodes that start work requests in the system. Thus far it has been assumed that work requests originate from one node, but this may not always be the case. Figure 6 shows the results of a test using 1,000 frames and a cluster of 50 nodes in which the number of originating nodes, or input sources, is increased from 1 to 50 in which case all nodes originate work requests. This test demonstrates the worst-case performance of the LRU and Round policies due to the fact that these policies do not have a universal list, or a node list shared by or synchronized between all originating nodes in the cluster. Also, this is a worst case for these policies because, in this case, the lists on each originating node end up being essentially identical meaning that after one originating node sends a work request to node 4, for instance, the next originating node will also send its next work request to node 4. Therefore when all nodes are originating nodes, the first set of initial work requests will all be sent to the same node. As for the random selection policies, they both use universal lists so there is almost no difference in performance.

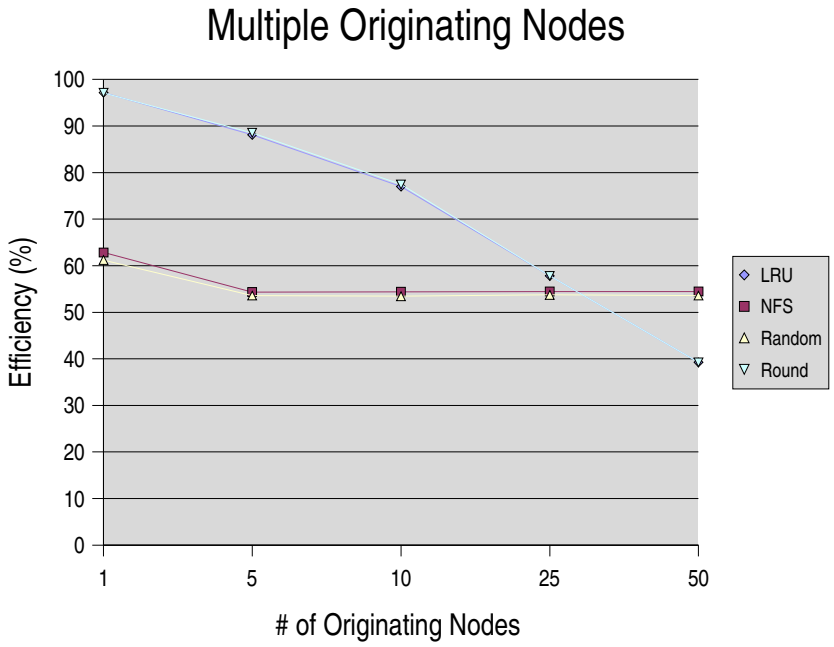

Figure 6

While the current design of the LRU policy does not support multiple originating nodes, it would be possible to modify it to do so. One simple, yet reasonable, way of doing this would be to broadcast a small message to the cluster each time a job starts or stops indicating the node the job will be sent to. Whenever a node receives a message it updates its list. Even in a larger cluster, the overhead incurred by the broadcast of such small

messages should be minimal.

5.4 Variable Work Request Size

Thus far the work request size has been constant. In this test the cluster will be tested by varying the request size randomly to different degrees using 1000 total frames and a cluster of 50 nodes. Three degrees of variability will be used in which different work request sizes occur at different frequencies:

Level 0: All requests consist of 1 frame.
Level 1: 80% of the requests consist of 1 frame and 20% consist of 3 frames.
Level 2: 50% of the requests consist of 1 frame, 30% consist of 3 frames, and 20% consist of 10 frames.

Figure 7 shows the results of the test. One result displayed in the graph is that this is the first test in which the performance of the NFS and Random policies have not been nearly equal. When there is no variation the performance difference is the same as in previous tests, but when variation is introduced into the job sizes the policy performances begin to deviate from one another. In particular, the Random policy decreases in performance much more so than the NFS policy. This is due to the optimization within the NFS policy which will not allow it to pick a node twice or more in a row. As the test executes the number of free nodes in the cluster decreases to a small fraction of the total severely reducing the range from which to select. It is here that this optimization aids in mitigating some of the problems with random selection, thus increasing performance.

The clearest result of this test is its display of the weakness of the Round policy. Namely, the policy is designed with the assumption that job sizes are constant. Once job sizes begin to vary the policy's simplistic walk through its list of nodes causes performance loss. The LRU policy, however, continues to show high performance. The Round policy shows a total performance loss of nearly 50% while the LRU policy shows a performance loss under 20%. The reason for this performance loss is again due to the design of the test; LRU performs optimally until the end of the test at which point all nodes with jobs of 1 or 3 frames have completed and are now idle. Execution is then forced to wait until nodes with jobs of 10 frames complete. It is this waiting for a small number of nodes to finish which decreases performance so sharply.

In order to illustrate this, Figure 8 shows the cluster utilization of the policies over time during a run of this test with Level 2 variability. The data for this graph is captured in the following way; each time a job starts or finishes, the current time of the simulation clock and the percentage of nodes which are currently not idle are recorded. This results in a detailed record of changes in cluster utilization.

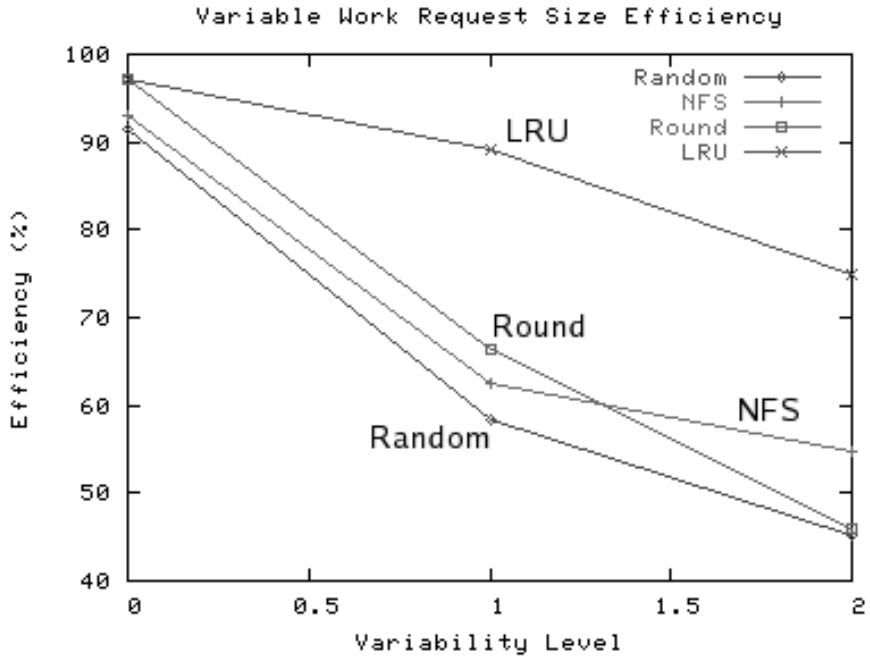

Figure 7

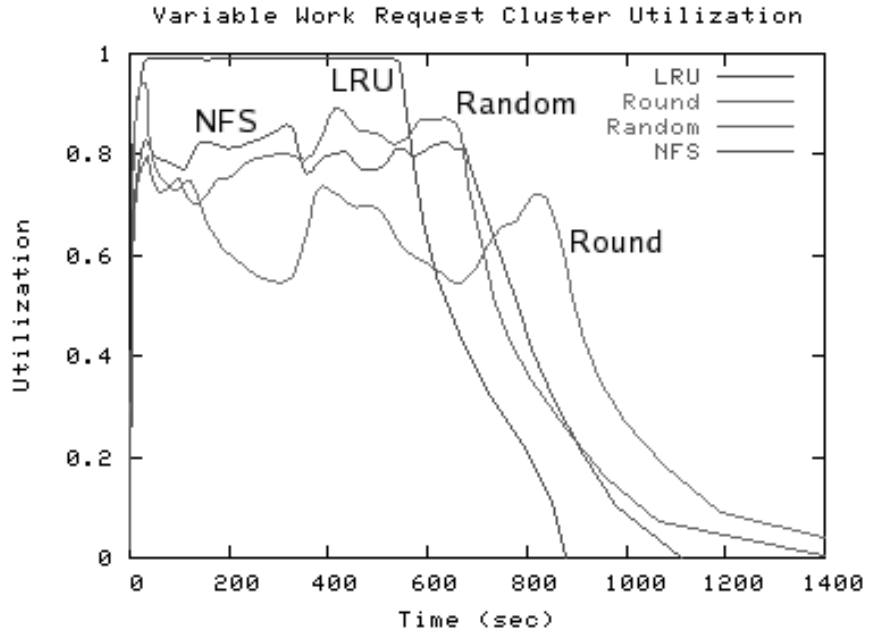

Figure 8

Although difficult to see on this graph, there is an initial period when the first set of work requests are sent out. After this period a plateau is reached which accurately reflects the behavior of the policy. Another period follows this plateau in which the last set of work requests are finishing. By inspecting the plateau we can, in most cases, see a clearer view the performance of the policies as the test progresses.

We can see by the plateau of the LRU policy that its performance is optimal despite variation in job size. Cluster utilization fluctuates between 98% and 100% the length of the test. This minor fluctuation is simply the result of a job ending, slightly lowering the percentage of busy nodes, and then the next job in the sequence immediately being sent out thus raising the percentage back up again.

Had this test been one without variability, the Round policy would have displayed results identical to those of the LRU policy. However, we can see that the policy's simplistic approach makes it a poor choice for environments with variability in job size. It's cluster utilization is actually lower and more turbulent than that of the random selection policies.

The results of the NFS and Random policies look similar here, but the difference is, again, found in the NFS policy's safeguard against migrating to the same node twice in a row. One evidence of this is Random's long tail on the lower right of the graph near the end of the test representing backlog and the effect of migrating multiple jobs to the same node at once when even other busy nodes would have been a better selection.

5.5 Varying Concurrent Work Requests

Another aspect of the system which has remained fixed under testing up to this point is the number of concurrent jobs, or the maximum number of jobs which may be running at once in the system. Thus far this has been set to the number of nodes such that there is one job for each node. While this makes sense for policies such as LRU and Round which traverse a list, it may not be an optimal situation for random selection policies. Conversely, it has not been tested how policies such as LRU perform under such conditions.

Figure 9 shows the results of varying the number of concurrent jobs from 50 to 1,000 in a test with 10,000 total frames, 50 nodes, and all work requests being one frame. The test with 50 concurrent jobs acts as a sanity test and is equivalent to the previous test of 10,000 frames and 50 nodes. Also, the Round policy has been omitted from this test as its behavior here would be equivalent to that of the LRU policy.

One result of this test is that the LRU policy again achieves constant, high performance with a maximum of 98.02% efficiency. An increase in the number of concurrent jobs does not effect the policy's optimal distribution of jobs.

Perhaps the most notable feature of these results is the vast difference in performance between the NFS and Random policies. After 100 concurrent jobs the two policies begin to deviate from one another in performance severely.

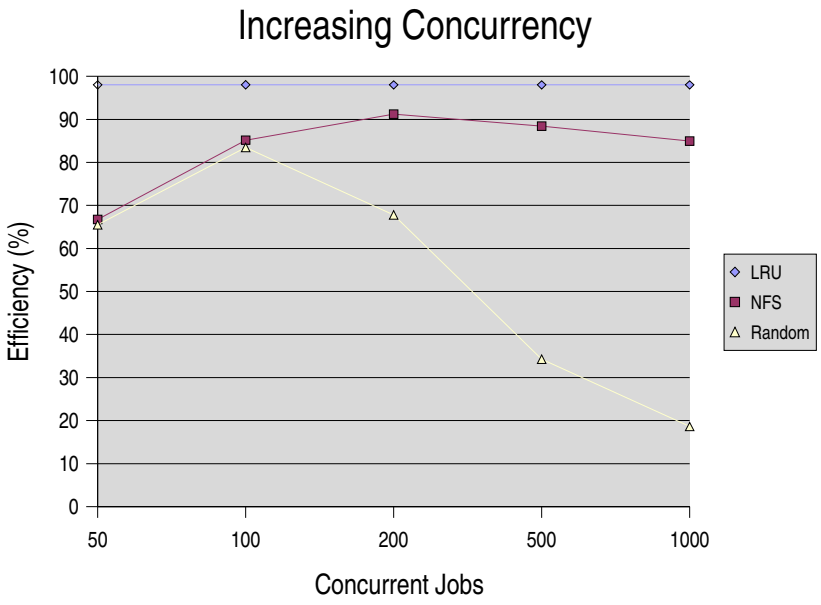

Figure 9

One interesting feature of this graph is that for the NFS policy, as the number of concurrent jobs increases the efficiency increases to a peak and then begins to decrease slowly. It appears that there is a range in which the policy reaches peak performance. To investigate this further, Figure 10 shows the results of a test with 4,000 frames, 50 nodes, and the number of concurrent jobs increasing from 50 to 350 or 1 to 7 times the cluster size. The goals of this test is, using a different number of work requests and nodes than the previous test, to attempt to determine the multiple of the cluster size which results in the highest NFS policy performance. It appears by the graph that the range of 4 to 5 times cluster size provides the highest efficiency. As in the previous graph, the two policies deviate in performance after 100 – 125 concurrent jobs or 2 – 2.5 times cluster size at which point the Random policy reaches its peak performance.

Figure 11 shows the cluster utilization over time for a run of this test with 250 concurrent jobs with the NFS and Random policies. One of the striking results of this test is the high performance of the policies during the plateau. In fact, the NFS and Random policies operated during this period at mean efficiencies of 99.0% and 88.8%, respectively. This gives the NFS policy a performance level comparable to that of the LRU policy. The Random policy also gives a high performance level except for one defect; one may notice that on the right side of the graph, after the NFS policy has completed the test, the Random policy exhibits a lag of over 3,000 seconds. The performance difference between the two policies is largely due to the NFS policy's use of two node lists for lightly loaded and heavily loaded nodes, respectively. Under this high amount of concurrent jobs, it is inevitable that all nodes will become heavily loaded. When this occurs, the NFS policy will continue to distribute work requests over the list of heavily loaded nodes. Because the Random policy does not maintain a list of heavily loaded nodes, however, it attempts to distribute work having an empty list and is forced to run all work requests on the

originating node thus resulting in the unusually large delay.

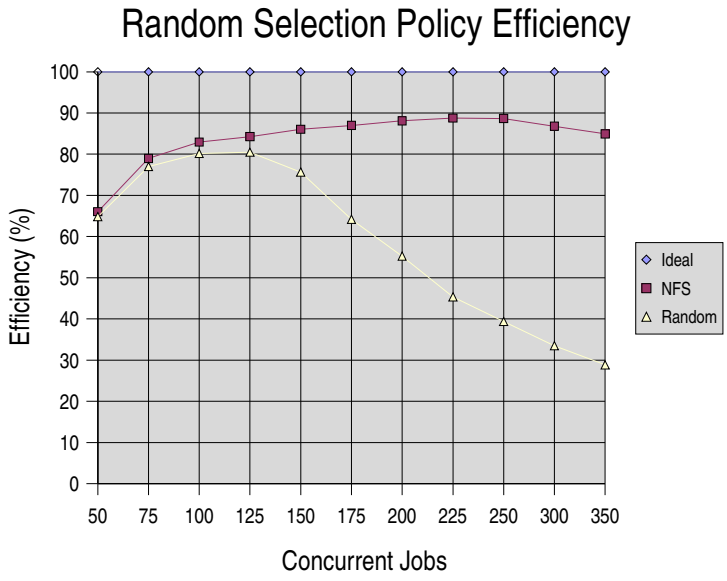

Figure 10

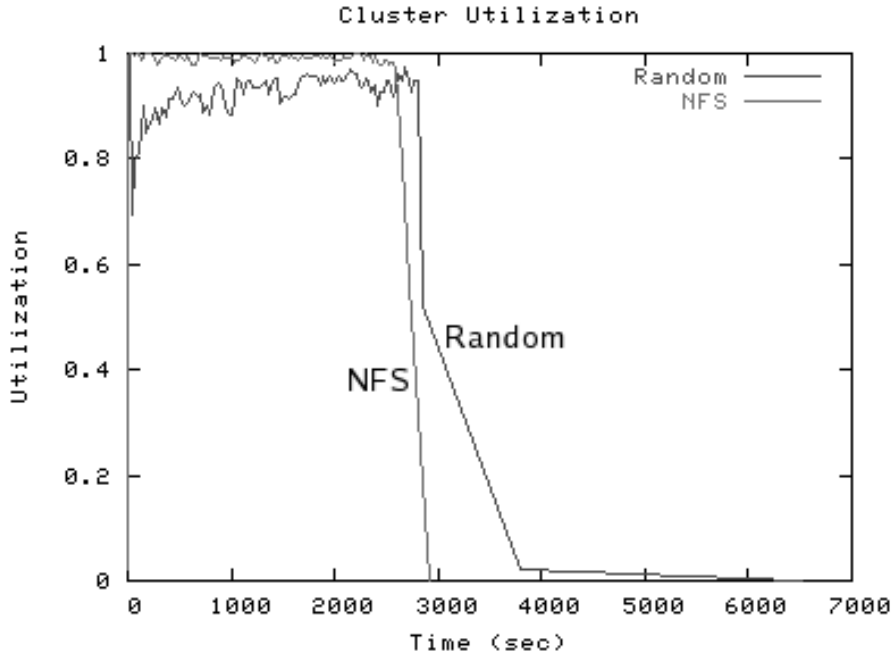

Figure 11

5.6 Improving the NFS Policy

Based on the results in the previous test we created a new policy, NFS2, which modifies the NFS policy by increasing the number of concurrent jobs to four times the cluster size. Figure 12 shows a comparison in steady state performance between the NFS and NFS2 policies under various tests. In each test, the NFS2 policy shows a 20-30% increase in cluster utilization yielding a maximum of 98.93%. Figure 13 shows the overall performance of the policies. By the steady state performance we can see that the NFS2 policy rivals the LRU policy and also provides high performance in cases of multiple originating nodes.

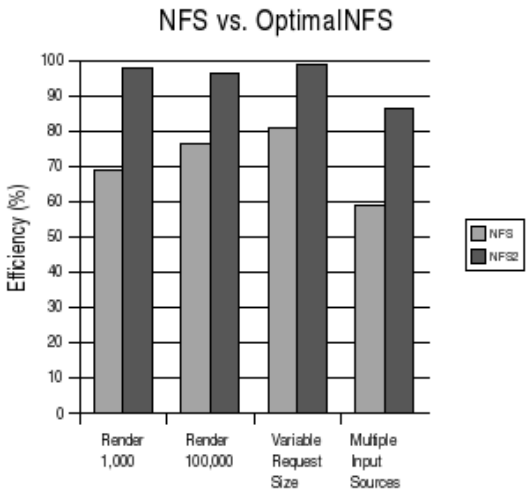

Figure 12

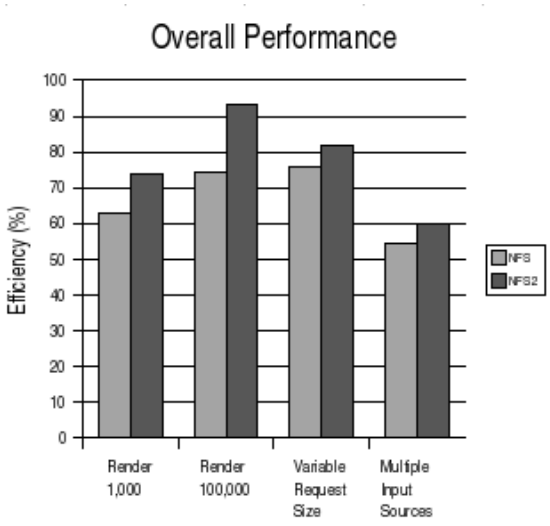

Figure 13

5.7 Performance in a Heterogeneous Environment

The existing system is a homogeneous cluster, each node being identical in hardware and operating system. In all other experiments, this has assumed as the simulation models the existing system. It is possible, however, for the software to run on a heterogeneous cluster and will be simulated in this experiment. The experiment is run using 1,000 frames and 50 nodes.

Heterogeneous may mean that the nodes vary in many different ways such as processor speed or memory or disk size. In this experiment the nodes will be heterogeneous only in that the time a job takes to process is determined by multiplying the job's process time by a multiple from the node it will execute on. The multiple of the node determined

randomly at the beginning of the experiment. There are three different levels of node speed variability:

Level 0: All nodes are the same speed.
Level 1: 80% are the base speed, 20% are 3x faster.
Level 2: 50% are the base speed, 30% are 3x faster, and 20% are 10x faster.

As shown in Figure 14, the results are very similar to those of the variable work request size experiment. The LRU policy performs optimally while the Round Robin policy decreases in performance dramatically. This is the expected result as the Round Robin policy has shown to perform poorly when variance is present. As before, the random selection policies are unaffected by the variance in node speed.

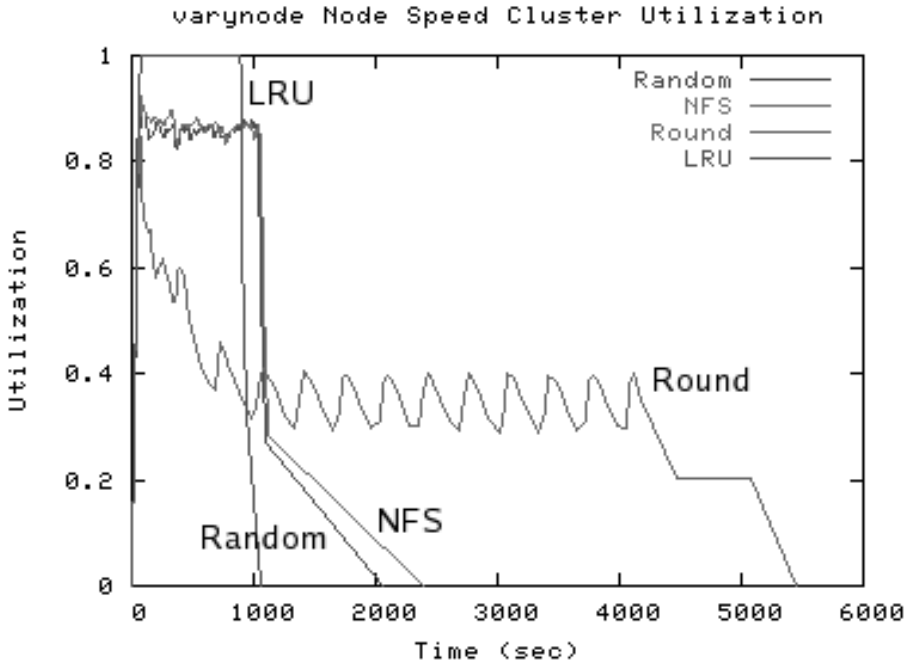

Figure 14

6 Designing Load Balancing Policies

When designing a load balancing policy, many intuitive notions of how a policy should operate are in fact counter-productive and degrade performance. It is often only through implementing an existing policy, such as those presented here, or through research and performance analysis, that any flaws in a design may become apparent. Many of the designs in the current policies exist due to seeing flaws in previously intuitive designs. In this section we will present some examples of designs which by using ideas which were

intuitive appeared to offer high performance but later proved otherwise. We will also present best practices of load balancing policies, or those properties which are present in a "good" policy.

6.1 Evolution of the Policies

The first policy used by PANTS, for several versions in fact, was the Leader policy. More information on the Leader policy and others can be found in the Load Balancing Policies section. Until it was inspected more closely, it was intuitive and assumed to be performing well. Many properties of this policy, however, decreased performance. One of the reasons its poor performance was not evident was because it was only used in clusters of less than ten nodes. Closer examination showed that the presence of a leader node quickly became a bottleneck as cluster size increased. In fact, even in small clusters, a sufficiently high workload could cause the cluster to go into a state of "meltdown" in which performance degrades significantly.

In order to eliminate the leader node, the Random policy was created which distributes the role of the leader node over all nodes in cluster. This helped to improve the stability of system and avoid "meltdown" under high workloads. Even during its development, the Random policy went through several versions with each revision changing something that had previously been thought a good approach. One idea was that since the nodes are classified as heavily and lightly loaded, work should only be distributed if the node starting the work is heavily loaded. If the node is lightly loaded it is acceptable to execute it on that node. This proved to be a very bad policy as in tests such as the bigsum test all work would be executed on the originating node. To fix this, work is always distributed even if the originating node is lightly loaded.

Another design of the Random policy is that work is only distributed over lightly loaded nodes. This is understandably intuitive as heavily loaded nodes are currently busy and so work should only be sent to those that are lightly loaded. Thus the list on each node only contains only lightly loaded nodes. This introduced a problem in situations in which the cluster was under high load. In this case the list would either be very small, such as one or two nodes, or the list would be empty. When the list is empty, the policy would be forced to default to executing all requests on the originating node until another lightly loaded node became available.

As a solution to this problem in the Random policy, the NFS policy was created. The major modification in this policy was its assumption that no matter what the state of the cluster, work requests should always be distributed. One way of implementing this was to maintain lists of both lightly loaded and heavily loaded nodes. If all nodes became heavily loaded, work requests would be distributed over the heavily loaded nodes. This offered significant performance improvement under high load as demonstrated in the

increased job concurrency experiments. Up to this point, all policies had used random node selection but closer work with the design of the policies showed that random selection did not distribute work as evenly as one might expect. In fact, a node would often be selected twice or more in a row. In an attempt to mitigate this the NFS policy included an optimization which prevents a node from being selection twice in a row even if it is the only node in a list.

In another attempt to overcome the problems with random selection, the Round Robin policy was implemented as a move toward a more deterministic policy. It was thought that if each node received the same number of work requests this would result in an even distribution of work and thus optimal performance. Additionally, the policy's design included what was found to be good in previous policies; work requests were always distributed, even over heavily loaded nodes. In most situations the policy showed near-ideal steady state performance. Any variation in the work requests or node speed, however, caused the policy to perform much worse than the random selection policies.

To handle situations in which work request size or node speed vary, the LRU policy was designed to prioritize nodes by the number of work requests currently executing on each. Another problem found in the random selection policies is their dependence on knowing node state. Knowing this required that the load on each node be measured and updated periodically which lead to the lists becoming slightly outdated. The goal of the LRU policy was to have a more accurate picture of the cluster state at any time which led to using the number of jobs on each node rather than each node's load status as the factor in node selection.

By looking back over the evolution of policies supported by PANTS, we can see that designing a good load balancing policy is a non-trivial process. An intuitive design is often not optimal. This difficultly in designing policies can be mitigated, however, through understanding how policy performance can be measured and what behaviors are characteristic of good policies.

6.2 Measuring Efficiency

In order to design a good policy, one that offers near-ideal performance under the circumstances for which it is designed, we must understand how to measure performance and what aspects of a policy's behavior most affect it. Here performance has been measured as either overall performance or steady-state cluster utilization.

Overall performance, or efficiency, is calculated as the ratio of the speedup of the policy and the ideal speedup. For instance, if for a cluster of 100 nodes a policy offered a speed-up of 80x, compared to an ideal speedup of 100x, would be a .80 or 80% efficiency. This serves as a good general measure of performance but does include overhead in calculation

and so varies with the length of the test as noted in the section on simulation test results.

Steady-state cluster utilization is calculated as the ratio of the number of busy nodes to the number of nodes in the cluster. It can be calculated at any time and is useful for providing a more detailed view of a policy's behavior as well as the policy's performance independent of the length of the test. Not only is this a useful metric but actually measures the most important characteristic of policy behavior; how fully the policy utilizes the nodes available. In fact, if one were to look at a graph of cluster utilization over time, take it's integral, and divide by the length of the test one would find that this and overall performance are nearly equal, deviating from one another by 1-2%. Figure 15 shows the a graph of cluster utilization over time. Calculating overall steady-state performance can be accomplished in the same way as overall performance by not including the initial and end periods in the integral. For instance, the LRU policy here shows near-optimal steady-state performance but overall performance, which includes the initial and end periods, would be lower.

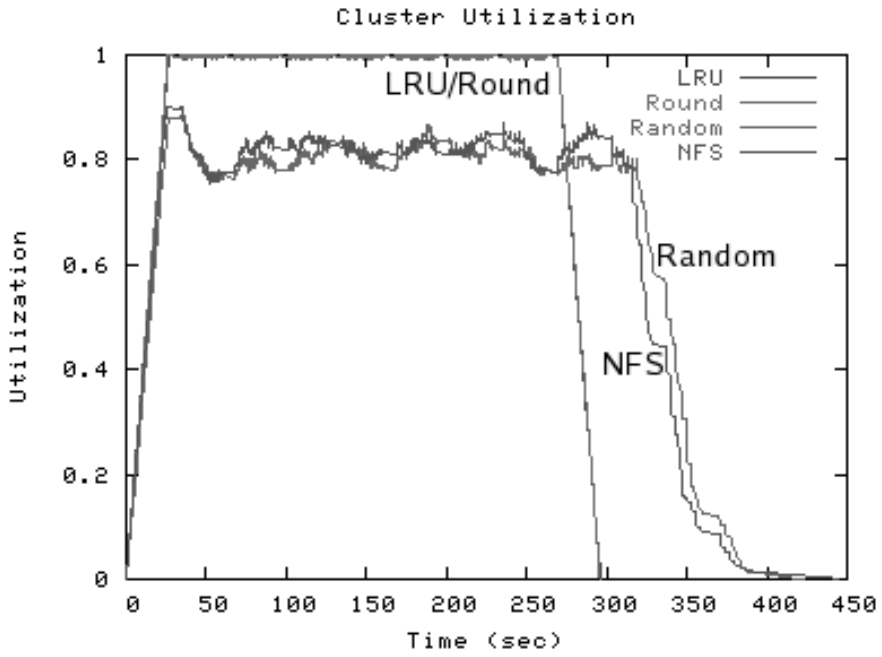

Figure 15

This relationship between cluster utilization and efficiency shows that the key behavior of a policy is node selection. Node selection is what determines the level of cluster utilization and at anything less than 100% utilization, there are idle resources which could have been used.

6.3 Best Practices

In this section we will look at best practices, or general guidelines, in designing load balancing policies. There are behaviors common to all good policies but that can be difficult to identity. We will present these behaviors here and why they are important.

One behavior that a policy should include is to always distribute work. This means that despite cluster state, such as being under high workload, work distribution never defaults to a single node. This behavior should be fairly intuitive as defaulting to one node does not make sense if the goal of a policy is to distribute work load evenly.

The most important behavior to any policy is to avoid "bad choices". While the previous behavior is essential, it is actually a subset of this general behavior. Bad choices occur where a policy selects one node, when another could have processed the work faster. For instance, the LRU policy performs optimally because, by prioritizing nodes by the number of jobs on each, it essentially prioritizes nodes by which nodes can process the next work request fastest. The random selection policies, however, encounter problems due to random selection and periodic node status updates. Random selection leads to nodes being selected twice or more in a row and periodic updates result in incorrect cluster information. These cause the policy to make bad choices which can be seen in their approximately 80% cluster utilization under normal operation. Note that we speak of avoiding bad choices rather than making good choices although these are essentially the same thing. For any node selection there could be many bad choices, selecting a busy node rather than one that is idle, and many good choices, such as selecting an idle node.

6.4 The Challenge of Policy Design

Having identified those behaviors which are necessary for any good load balancing policy, we find that implementing such a policy presents problems which may not be apparent initially. Here we will address the key challenges in designing and implementing a policy.

Consider a hypothetical policy which offers optimal steady-state performance. This mean that for any node selection, the policy never makes a bad choice. Clearly the policy would need to have a correct view of the cluster status at all times. Now begin to consider a possible implementation for such a policy. How do the nodes communicate? Where is cluster information stored? How is this information updated and what does it contain? How is this affected as cluster size increases? One can quickly see that these issues are the reason policy design is non-trivial. Maintaining accurate node information while minimizing overhead is the key challenge in policy design.

One issue in this challenge is the trade-off presented of balancing how much communication takes place. Where this balance falls depends on the intended use of the policy and the environment it is designed for. For instance, in situations where only one originating node is used the LRU policy offers ideal performance with no communication overhead, but in cases where there are many originating nodes it's performance decreases dramatically. The NFS policy, however, offers much higher performance in cases of multiple originating nodes but incurs more communication overhead than the LRU policy. Each policy has its own strengths and weaknesses and intended use.

The design of a good policy involves not only including those behaviors that are best practices, but also considering its intended use and the specific problems introduced by the system it is included in.

7 Conclusions

In order to more thoroughly test the existing system and its load balancing policies we built a simulation of it. We have shown this simulation to be valid and have used it to test system performance in larger clusters as well as various workloads including varying work request size, varying node speed, increasing concurrency, and multiple originating nodes.

We have demonstrated the LRU policy to perform optimally in all tested situations, achieving 99-100% steady-state performance, except when there are many originating nodes which it is not currently designed for. Future work on this could include modifying it to support multiple originating nodes with a mechanism such as node status broadcasting. The Round Robin policy was shown to perform the same as the LRU policy, except for when work request size or node speed variance is present in which cases its performance degrades significantly.

The NFS policy has been shown to perform poorly when all node selections are made in a short period of time. In other situations, however, it achieves approximately 80% steady-state performance. An improved version of the NFS policy, NFS2, has been shown to achieve near-optimal performance in many situations through increasing work request concurrency. The Random policy demonstrates performance slightly lower than the NFS policy in most situations. Its performance degrades dramatically under high concurrency or under high variance in work request size.

By reviewing the development process of the PANTS policies, we have identified best practices in designing policies. Namely, that a policy should always distribute work despite cluster state and always attempt to make a bad choice in node selection. We also identified balancing the need for cluster state information with communication overhead as the key challenge in policy design.

References

Adams, D. (2004), “NFS Load Balancing Policy,” *Independent Study, Worcester Polytechnic Institute Computer Science Department, Worcester, Massachusetts.*

Adams, D. and LaFleur, J. (2004) “Leaderless Load Balancing in a Beowulf Cluster,” *Major Qualifying Project CS-DXF-0401, Worcester Polytechnic Institute Computer Science Department, Worcester, Massachusetts.*

Beowulf Web Site (2005), “Beowulf.org: The Beowulf Cluster Site”, www.beowulf.org, (Accessed 25 March 2005)

Claypool, M. and Finkel, D. (2002), “Transparent Process Migration for Distributed Applications in a Beowulf Cluster”, *Proceedings of the International Networking Conference, pp 423 – 422.*

Colt Web Site (2005), "Colt", hoschek.home.cern.ch/hoschek/colt/, (Accessed 8 April 2005)

Lemaire, M. and Nichols, J. (2002), “Performance Evaluation of Load Sharing Policies on a Beowulf Cluster,” *Major Qualifying Project MQP-MLC-BW01, Worcester Polytechnic Institute Computer Science Department, Worcester, Massachusetts.*

MPI Web Site (2005), “Message Passing Interface”, www-unix.mcs.anl.gov/mpi/, (Accessed 25 March 2005)

openMosix Web Site (2005), “openMosix, an Open Source Linux Cluster Project”, openmosix.sourceforge.net, (Accessed 25 March 2005)

PVM Web Site (2005), “PVM: Parallel Virtual Machine”, www.csm.ornl.gov/pvm/, (Accessed 25 March 2005)