Project Number: MLC-BW01

Performance Evaluation of Load Sharing Policies on a Beowulf Cluster

An Major Qualifying Project

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Computer Science

by

Marc Lemaire

James Nichols

April 25, 2002
Approved:

Mark Claypool

Abstract

Load sharing in a Beowulf cluster can be done transparently by PANTS. However, PANTS distributes load using only a measurement of CPU usage to index load. While CPU usage is the typical metric used in load distribution, other system resources such as disk and memory can become loaded and be a performance bottleneck. We examine PANTS in the context of load distribution algorithms, build new load indices, develop benchmarks, and evaluate performance. We find our load indices can reduce compile time of the Linux kernel by 1/2 of the time of the original PANTS indices.

Contents

1 Introduction

A Beowulf cluster is a distributed system consisting of inexpensive computers networked together cheaply, usually via ethernet. It is often desirable in such systems to distribute workload throughout the cluster.

Ideally, the distribution of workload tries to share load equally among all the machines in the cluster, decreasing response times and increasing overall throughput. A drawback of workload distribution in clusters has historically been the need for expertise in both designing and implementing applications to make good use of the distribution of workload.

PANTS Application Node Transparency System is a load distribution system which strives to remove the need for expertise required by other load distribution mechanisms [BG]. This is achieved by adding a layer of transparency to the load distribution mechanism. PANTS intercepts `execve()` system calls under Linux and transparently shares the process with other machines in the cluster also running the PANTS load sharing daemon. PANTS also implements a multicast messaging policy which minimizes messages to nodes busy with computations and is also fault tolerant to machines failing in the cluster.

PANTS uses the `/proc` filesystem to obtain a count of jiffies (1/100ths of a second) that the CPU spent processing in user mode in the past 5 seconds. PANTS then calculates the total jiffies and finds the percentage of the user mode jiffies to the total. If this percentage if over 95% then PANTS considers the node loaded or "busy"; otherwise it is considered "free". In the current PANTS implementation, any process or workload which does not generate CPU load will not be shared amongst other machines in the cluster.

Early results from the implementors of PANTS showed a near linear speedup for computationally intensive applications [Moy]. Unfortunately, programs that impart load on the CPU of a machine are not the only applications that are desirable to run on Beowulf clusters. An application could read or write many files to disk imparting load on the I/O subsystem, maintain large data structures loading memory, or cause system events such as interrupts and context switches.

Similarly, load measurement does not have to be based on CPU usage. Load measurement could also be based on I/O in terms of number of blocks read and written to the disks, memory load in terms of total pages read and written per second, context switches per second, or interrupts per second. Load can also be measured as a mix of all of these criteria. Compiling the Linux kernel is a typical example of an application which imparts significant load on the I/O resources of a system. Compiling the kernel requires some CPU resources, but does not impart significant load on the CPU. When just looking at CPU usage as the only measurement of load, a system compiling the kernel would not "load" the system, while there would be a performance benefit if the load measurements included I/O use.

The goal of this report is to examine PANTS in the context of load distribution algorithms, to devise new methods of capturing load metrics, and also to measure the performance of new metrics and policies we have devised. We add new ways to measure load in PANTS such as I/O usage, memory usage, context switches, and interrupts, as well as improving the way CPU usage is measured. To test the new measures of load we build a micro benchmark for testing each new measure of load and a macro benchmark for testing with different mixes of load. To better evaluate the performance of our new load indices we also used a real-world application as a benchmark: a distributed compilation of the Linux kernel.

This report commences as follows: Chapter 2 is an organization and presentation of work related to our project. This includes an overview of Beowulf clusters and software tools as well as different load indices and policies. The current configuration of our WPI cluster is also discussed, specifically the hardware, operating system, and applications associated with our project. Chapter 3 presents our methodology and approach including: improvements to the PANTS implementation, load metrics we devised, and discussion of the benchmarks we created to measure performance. Chapter 4 contains analysis of our results.

2 Related Work

To make our design choices clear, we first discuss the background surrounding our project. After talking about Beowulf clusters in general, and the PANTS system in particular, we cover the various indices and policies involved in the measurement and management of load. We also specify the configuration of our hardware and software, and mention the applications involved in our Beowulf cluster in some detail so that we may see how and why the load sharing policy can be improved.

2.1 Load

In a system consisting of a typical network of workstations it has been shown in simulated and analytical studies that there are many idle workstations at any time [ML87]. It is desirable to distribute load from busy workstations to idle ones, increasing processor utilization and performance [dS96].

A Beowulf cluster is designed to facilitate sharing of load amongst a number of connected nodes, instead of processing entire tasks from only the node that started them. Clusters usually have two types of nodes, those that typically interact with users, and those that can only be used through their network interfaces (`ssh`,`rlogin`). The nodes without monitors are idle, while the nodes with monitors often have high load in response to user interaction. Load distribution is achieved in WPI's Beowulf cluster through the use of the PANTS system, which implements a load distribution algorithm.

2.1.1 Load Distribution Algorithms

[RR96] suggests several objectives for a load distribution algorithm, of relevance to us are: minimization of task average response time, balanced distribution of load, and minimization of machine idle time. Load distribution algorithms strive to meet each of these objectives in different ways by balancing the inherent tradeoffs.

[Moy] suggests that load balancing is impractical because it may needlessly transfer tasks just to achieve balance. Those transfers are often very costly and degrade the performance of the system. This is supported in work done by Eager, Lazowska, and Zahorjan, authors of a seminal work in load sharing [ELZ86].

PANTS attempts to distribute work to idle processors away from busy processors to minimize average task response times [FW95, Moy, DHV00]. This defines PANTS as a load sharing algorithm instead of a load balancing algorithm. In other words, the PANTS algorithm makes no attempt to balance the load among machines. PANTS simply attempts to keep idle machines busy by transferring tasks from busy machines.

[dS96] splits load distribution algorithms into four policy components. What, when, and where information is collected is the initial part of the algorithm and comprises the "information policy". The "transfer policy" uses this information to decide when a processor should send (or accept) tasks to (or from) the other processors in the system. The "selection policy" chooses a task. The "location policy" determines which machine to transfer the task to.

The PANTS load sharing algorithm can be split into the component policies above:

Information policy: The PANTS client daemon, PANTSD, runs on each node of the cluster. This daemon monitors the load at the node by periodically (in past implementation, 5 seconds) checking the CPU usage though the `/proc` filesystem. If this load is under the static threshold set at runtime, the machine is considered "free", and the daemon sends a multicast message to the leader multicast address. This message says that the node is available to accept tasks. If the load is over this threshold it again sends a message to the leader stating it is "busy" and can't accept any tasks. The leader subscribing to the leader multicast address is a PANTS daemon running on a node in the cluster (this is determined when PANTS is started. It is fault tolerant and handles losing the leader - see [DHV00]). This leader maintains a list of "free" nodes which are willing to accept tasks.

Transfer/Selection policy: When a process is started in PANTS , the PREX mechanism first checks to see if it is migratable. This is done by checking a flag set in the binary of the executable. If the executable if flagged as non-migratable it is executed normally. Otherwise, PREX communicates with the local PANTS daemon, and if the last load measurement indicates the node is free the process is executed normally. If the node is busy, the process is selected for migration and an attempt is made to transfer the process.

Location policy: Once a process has been selected for migration PANTSD sends a message to the leader multicast address asking for the address of a free node. The leader chooses a random node from the free list it has been maintaining and returns the address to the requesting daemon. The leader then removes the node from the free list, as the node will soon be busy. If there are no nodes in the free list a blank address is returned. PANTSD then gives the address to PREX, which either sends the process to the free node or executes it locally if there were no free nodes in the list.

The important work done by PANTS is in the information policy. All of the past work done on WPI's Beowulf cluster suggests as future work modifying the information policy, specifically the tuning of load variables [DHV00]. Indeed, these load variables, or more generally, this quantification of load plays a pivotal role in the information policy of any load distribution algorithm. Measured load along with several thresholds determines the state of a node: either free and able to accept tasks, or busy and unable to accept tasks. The choice of those metrics and thresholds are fundamental to the performance of the algorithm [dS96].

2.1.2 Load and Load Indices

Load is the demand for services or performance made on a machine or system. From an operating systems perspective load is the demand or usage of some system resource. The most common resources are the CPU, system memory, disk access, disk space, network bandwidth, and attached devices such as printers, etc.

In the context of a load distribution algorithm, load is indexed and measured for use in the transfer policy. This load metric is then used to determine if a machine is "free" or "busy". In other words, the load metric is used to decide if the machine should attempt to lessen it's load by transferring tasks, or take on more load by accepting tasks from other machines in the cluster.

A load index can be comprised of a number of things: CPU queue length, CPU usage, average response time, I/O queue length, I/O service time, I/O blocks read/written, memory page-fault rate, idle process run time, CPU load average, or demand on some other system-specific resource [Moy, dS96].

PANTS monitors the load at a node by periodically checking the CPU usage through the `/proc` file system. The number of jiffies that the CPU spent processing user, system, nice, and idle processes in the last 5 seconds are obtained. The user, system, and nice totals are summed and divided by the total, obtaining a percentage which is then compared against a static threshold set at compile time.

2.2 Current Configuration

This section describes the current configuration of hardware and software. This will allow our results to be understood in context, and also allow them to be reproduced more exactly in the future. In addition to base hardware and software, our cluster has some software that should be explained, concerned with the workings of PANTS. This section is supplemented by the logs of configuration for each node, found in Appendix B.

2.2.1 Hardware

The current PANTS Beowulf cluster (hereafter referred to as 'the PANTS cluster') is composed of seven 600MHz Alpha machines. The physical memory ranges from 512MB on the file server to 64MB on several of the node machines, and all machines run at least 128MB of additional disk swap space. Each machine is equipped with a PCI Ultra-Wide SCSI controller and UW SCSI hard drives, as well a 100Base-T network card. The network is arranged in a star topology, with one machine providing a gateway to the outside world through an additional network card and IP-Masquerading software. The hostnames of the nodes are Pants, Beowulf, Moscow, Rome, Paris, London, and Shanghai.

2.2.2 Software

All of the machines in the cluster are running RedHat Linux 7.1, the latest release to support the Alpha architecture. The Linux kernel is version 2.4.18, each kernel image is compiled with NFS support, and the build 'flavor' is specified for each system's particular motherboard and processor type. The cluster shares a common `/home` directory which is shared via NFS from the file server. This common file system is necessary to facilitate rsh execution of child processes. The NFS server is the fastest of the machines and it's kernel image has a larger maximum read/write block size than default. The read/write block size settings on remote machines are also increased, for better performance. To see a listing of all the processes running on each system in the cluster see Appendix B.

2.2.3 PANTS

Each machine runs the PANTS daemon, which provides the bulk of the PANTS distributed functionality. This daemon controls the availability of a node to accept processes, the list of currently free nodes, and the allocation of free nodes to processes which request them.

The PANTS daemon has two procedures for measuring load and determining its availability, both of which examine the current CPU load. The first method uses the `sysinfo()` system call to query the last one minute of CPU load, which returns a status of 'available', 'unavailable', or 'overloaded'. This method is not in use by default. The second method, which presently is in use, examines the CPU jiffies as reported in the `/proc/stat` file. This measurement provides the number of jiffies (1/100th of a second) spent by the CPU processing in user mode in the immediate past. The threshold for 'unavailable' status is set at 95% CPU utilization.

2.2.4 PREX

PREX (PANTS Remote Execution) is the means by which processes are migrated. This library receives all requests for new processes and examines the requested binary to determine if it has been flagged migratable. If so, PREX requests a free node from the PANTS daemon, then executes the binary on that node via rsh. For this system to work the binary must have the same absolute path on both the host machine and the remote node. For the current cluster setup this means the binary must be within the `/home` directory structure, which is shared via NFS.

2.2.5 Past Benchmarks

The group that implemented PANTS developed a simple benchmark to test functionality. This test consists of an application which performs summations by splitting a large sum into multiple chunks and exec'ing a process for each chunk. PANTS then sends processes away from the originating machine, resulting in a near linear speedup for this specific application when the sum was sufficiently large enough. They found a 1 second delay of execution for processes that were migrated [DHV00].

3 Methodology and Approach

Our methodology consists of improving the PANTS implementation by adding configuration options, and signal handling capabilities. The implementation modifications facilitate improvements in the load measurement techniques. These improvements include: snap shot load measurements, configurable exponentially weighted averaging, metric weighting, and adjustable sampling frequency settings. We are now able to accurately measure the load on the system by looking at CPU usage, I/O usage, context switching, and memory usage. We have developed benchmarks to load these components, an overall macro-benchmark, and a real world application. The purpose of these benchmarks is to improve the performance of the PANTS demon under various loads.

3.1 Improved PANTS Implementation

We made several improvements to the PANTS implementation. These changes include adding a PANTS configuration file, signal/interrupt handling, and logging. These improvements are necessary to make PANTS more robust, configurable, and easier to evaluate in terms of performance. Since remote shell (RSH) is used by PANTS to remotely execute processes, we have also profiled the performance of RSH to establish the cost of migration in the PANTS system.

3.1.1 Configuration file

In the prior PANTS implementation many variables were static, coded in the main header file for the PANTS daemon. Included in these variables were multicast addresses, port numbers, and “timeout” settings. These timeout settings are in place to control the granularity of the load measurements. The daemon takes a load measurement, sleeps for the timeout period and then takes another measurement. Also hard coded into the implementation is the threshold for determining if a machine was busy based upon CPU readings being greater or less than this threshold. In order to evaluate the performance of the PANTS system it is crucial to us to have fine control over these values, preferably without a recompile of the PANTS code.

To enable this control we have implemented configuration file parsing, (`/etc/*.conf` files.) A user can enter textual tokens into the configuration file, for example: `log_metrics no` disables the logging of the load metrics [also see: Section 3.1.3]. A simple parser was constructed, and the daemon reads the configuration file on startup.

A user can also cause the daemon to re-read the file by sending the daemon a 'SIGUSR1' signal. Support has also been added later for configuring our new load metrics, their weighting, and also the exponential weighted averaging [see: Section 3.2]. For details on the valid options in the configuration file see Appendix A.

3.1.2 Signal/Interrupt Handling

The prior PANTS implementation only responded to termination signals, either through a Ctrl-C key press or a kill system call if the daemon was not invoked by a user inside a shell. We added support for SIGUSR1 signals, which causes the daemon to reread the configuration file [see: Section 3.1.1], SIGHUP signals, which cause the daemon to completely restart, SIGUSR2 signals, which toggle the logging of metrics [see: Section 3.1.3], and SIGTRAP signals, which toggle additional logging to standard error.

3.1.3 Logging

Logging of daemon information and load data is very important in terms of performance evaluation. PANTS printed to standard output some information such as availability, multicast messages it received, etc. Unfortunately, this functionality was limited if the daemon was started at boot time. There was also too much data being output by the daemon to make this feature useful if started from a shell.

We improved PANTS by adding compatibility with the standard syslog daemon which comes with every Linux system. The PANTS daemon now logs to a separate log file in `/var/log/`, making data collection and analysis easier. The daemon has multiple levels of logging: error logging, load metric logging, and all purpose logging, is configurable through the configuration file [see: Section 3.1.1], and can also be controlled through the use of signals [see: Section 3.1.2].

3.1.4 Profiling RSH

When a process is going to be remotely executed on another machine, PANTS makes use of Remote Shell, or RSH [see: Section 2.2.2]. The original implementors of PANTS developed a simple benchmark [see: Section 2.2.5] and held that performance was only increased when a sum of substantial size was calculated [DHV00].

They claimed that this was due to the fact that a small summation does not experience any speedup under PANTS because there was significant overhead (1 second) in executing a process via RSH. A 1 second overhead could add up if there was large number of short process migrated by PANTS.

We devised an experiment to profile the performance of RSH remote execution. The experiment program that we wrote remotely executed commands through RSH on random nodes in the cluster and measured the time each took to execute. After one thousand executions of the same command were executed the average time per execution was calculated. The experiment program also ran the command one thousand times on the local machine using the `system()` call, also calculating the average time per execution.

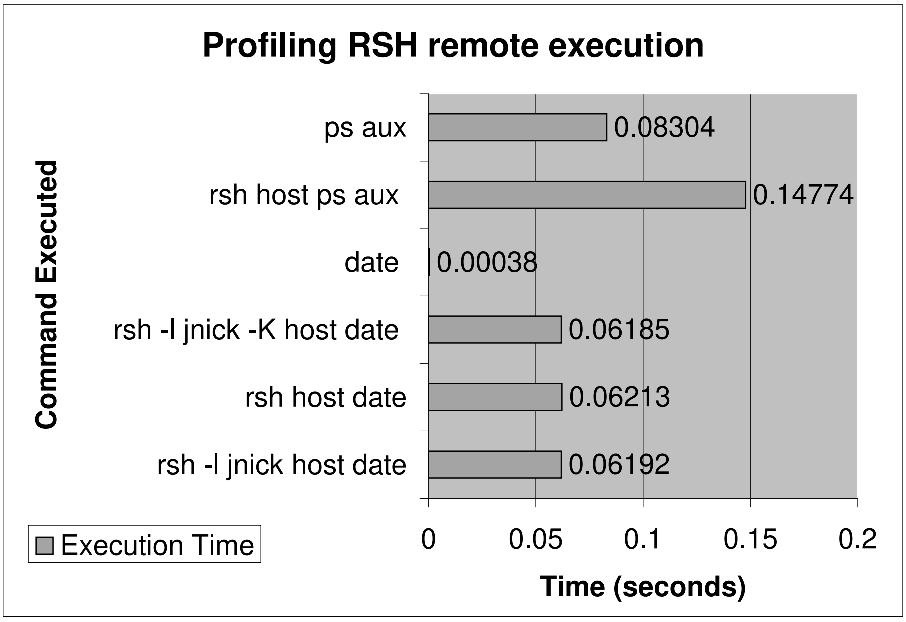

Figure 1: Remote Execution

In comparing the average execution times of `ps aux` and `rsh host ps aux` it is evident that remote execution through RSH takes approximately 0.06 seconds longer then local execution. This is confirmed by comparing the average execution times of `date` and `rsh -l jnick -K host date`. Also profiled were different command line options given to RSH, including the login name and enablement of Kereberos authentication, which has no significant effect on the average execution time. We concluded that the performance impact of RSH was small and that there was no reason modify the remote execution mechanism in PANTS.

3.2 Improved Load Measurement Technique

The original PANTS daemon measured load by simply looking at the percentage of CPU jiffies spent processing in user mode over a period of time. While this can be a valuable measurement, several improvements were made in this implementation to facilitate a more complete measurement of load.

3.2.1 Modularity

A primary change to the original PANTS implementation was the modularization of the load measurement code. While the original CPU measurements were only two procedures, they were contained in the primary source files of the daemon. In order to supplement this code it was first extracted and placed in its own module outside the existing source files. This allowed easy modifications of the entire load measurement system with few alterations to the rest of the daemon, and greatly increased the overall clarity of the source code. A substantial number of constants were introduced into this module to increase the readability of the load functions. The module communicates its status to the daemon by returning a value to indicate its availability.

3.2.2 Snapshot Load Information

The load information used by the daemon is read directly from the /proc/stat file produced by the proc filesystem. This file provides direct access to kernel data structures which store accounting information used to make load decisions. By maintaining a series of moments of information from this file, we can compute a windowed average of the activity of a node. This series is constantly being updated, to provide a system of snapshots, employed to maintain a history of load over several intervals. These snapshots record certain system accounting variables at a point in time and are time stamped as they are read from `/proc/stat`.

3.2.3 Configurable Metric Weighting and Sampling Frequency Settings

This series of snapshots provides several advantages. Primarily it allows exponential load weighting where the measurement of load on the system at any point in time can be weighted slightly against past performance. System resources used during each interval are calculated at each request for system load, and the previous intervals are multiplied by a fraction. By increasing this fraction the system can flatten statistical "spikes" in the load metrics. The system is currently configured to record the past two measurements. The weighting values for these past measurements are configurable from the main PANTS configuration file and allow easy modification of both the weight of previous load measurements and the number of past measurements examined. The interval between snapshots is also controlled in the configuration file and can be changed during execution. Time stamping each individual snapshot allows for modification of the time interval during execution by providing an accurate interval measurement between each snapshot. All metrics measured are effected equally by any exponential weighting.

3.3 Load Metrics

The actual load variables read from `/proc/stat` include CPU usage, I/O measurement, number of context switches, memory pages read and written, and overall interrupts generated on the system. These numbers are read from `/proc/stat` into a data structure and are then time stamped. See the manual page for the `/proc` filesystem for more information about the values in `/proc/stat`. Each metric has a threshold value that is used to determine the availability of the node. If the measurement produced by the metric is over this threshold the node is considered loaded and will return an unavailable status to the leader node. The node will not be listed as available until the loads are under the thresholds, and the leader node is then informed of the node's availability.

3.3.1 CPU

The CPU value is read as a series of snapshot readings which represent the overall number of jiffies (1/100ths of a second) that the CPU spent processing in user, nice, system, and idle modes since bootup. While the default PANTS load sharing policy looks exclusively at jiffies spent processing in user mode, our policy totals all jiffies spent in user, system, and nice mode between snapshots and divides by the total jiffies over that same interval. If this percentage is over the 95% threshold setting for CPU load metric the node is loaded.

3.3.2 I/O

Input and output are measured in terms of number of blocks read and written to disk. This value is divided by the interval time to produce blocks/second, which is them compared against the I/O threshold value to determine if the node is loaded.

3.3.3 Context Switching

Context switches are measured in terms of total switches performed by the system since bootup. Total switches for an interval are calculated by taking the difference of total switches in two snapshots. This number is then divided by the time interval between the snapshots to produce context switches per second. If this number is above the context switching threshold the node is considered loaded. A significant number of context switches occur while the system is completely unloaded; this must be taken into account when setting the context switching threshold.

3.3.4 Memory Usage

Memory is measured in terms of total pages read and written. For the purpose of this benchmark there is no difference between a page write and a page read, instead they are added together to produce total memory page operations. The difference in total page operations between two snapshots is divided by the time interval to produce page operations per second over the interval. If this value is larger than the memory threshold the node is considered loaded.

3.3.5 Interrupts

Interrupts are measured as total interrupts generated since bootup. The difference between the total interrupts measured in two snapshots is divided by the time interval between the snapshots to produce interrupts generated per second. If this value is larger than the interrupts threshold the system is considered loaded. A significant number of interrupts are thrown in normal unloaded system execution which must be observed when setting the interrupt threshold. This is usually around 100,000 interrupts per second on our cluster.

3.4 Micro-Benchmark

A refined understanding of the way that PANTS shares load is made possible by tests with fine control over the CPU cycles used, interrupts thrown, disk usage, memory usage, and context switching. The micro benchmarks that have been created for this purpose allow us to test specific actions of PANTS in controlled situations. For any given test, different amounts of different kinds of work can be assigned to the system, and the sharing of work can be studied for tuning and analysis purposes.

The manner by which the micro benchmark programs implement this control is by creating a task that consists mainly of one of the types of load we have measured, and reporting how fast the assigned tasks were completed. A system that shares the load better will complete the tasks faster. The types of load created and measured by the benchmarks are CPU usage, I/O to the disk drive or drives, and memory usage. Because PANTS works by relocating processes before executing them, these benchmarks start a number of processes on one node, and then allow pants to remotely execute them. Each benchmark test begins with several minutes of idle time to allow the systems to reach their baseline measurements.

When the actual benchmark tests begin, they write a time stamp to syslog to mark the start of their execution. The controlling process will then begin spawning a number of identical child processes at fixed intervals. These child processes do the actual work of the benchmark, and are migratable and therefore can be passed to a remote node for execution. When all the child processes are done, the syslog of the parent is again time stamped to mark the end of execution.

3.4.1 CPU Load

The CPU load process consists of a parent process that spawns four child processes along five minute intervals. Each of these child processes performs a large number of floating point operations (FLOPs). Flops were chosen for this benchmark because they are frequently used in some distributed computing. This benchmark shows system load conditions under a system that is loaded primarily with CPU intensive processes, while ignoring other metrics.

3.4.2 I/O Load

The I/O test is designed to load the disks of a machine while leaving the CPU relatively free. A copy of a large directory structure (384MB including files and directories) was placed on the local hard disk of each machine. The parent process then spawned four child processes along five minute intervals. Each child process copied the directory structure to a new location on the local hard disk. Many small files were involved requiring more writes than one large sequential file of the same size because new files and directories must be written as well as the actual data of the files. The file system used was the Linux ext2fs.

3.4.3 Memory Usage

The memory usage test consisted of a parent process that spawned four child processes along five minute intervals. Each of these child processes used the `malloc()` function to allocate 100MB of memory, and then it used the `mmap()` function to map an existing section of memory into this allocated area. After the memory was allocated and mapped it was released using the `free()` function. This method was repeated ten times in each child process, creating a rise and fall of virtual memory. None of the systems involved started the test with 100MB of free physical memory, so each machine must page memory in and out of its swap space during each cycle to provide enough virtual memory to fit the allocated structure.

3.5 Macro-Benchmark

Our micro-benchmark successfully loads individual components in the system. Most applications don't load a single aspect of a system, but instead load several. We developed a "Macro Benchmark" which loads multiple system resources, specifically memory and I/O access.

Similar to the micro-benchmarks this benchmark is highly tunable. It takes as command line parameters a number of processes to exec and an integer load factor. The optimum number of processes to spawn is the same as the number of nodes in the cluster. The number of processes are spawned, each set an alarm via the `setitimer()` system call to 100*load factor, and begin loading the system. This loading is achieved by opening a 130 megabyte file, reading strings out of it via `scanf()`, and outputting those same strings to another file via `printf()`. The alarm goes off after the specified time, giving a load in relation to time proportional to the parameter load factor.

By reading the large file many I/O operations are requested, along with a sizable amount of memory usage (enough to exhaust the physical memory and cause page thrashing), while CPU usage is kept relatively low. If PANTS were to quantify load based only upon a CPU metric, as it formerly did, processes like this would receive no load sharing benefit.

3.6 Real-world application

To further evaluate the performance of PANTS we chose to use a real-world application as a benchmark. The application is a distributed compilation of the Linux kernel, which is executed by the standard Linux program `make`. This distributed compilation exemplifies an application which imparts significant load on the I/O and memory resource with lower load on the CPU.

To make the compilation distributed via PANTS we had to make modifications to the compiling process. First, we marked the gcc compiler binary as migratable, which allows PREX to remotely execute gcc on remote nodes. Second, we modified the standard Makefile included with the Linux source tree to use a script program (my_gcc) as the compiler. This script program simply made any file references passed from make into absolute paths. Relative paths to files are not translated properly when sent to remote nodes, as the working directory is where the binary is located on the filesystem.

The my_gcc script then uses an `execve()` of the real gcc giving it the absolute file names it determined as arguments. PREX then intercepts this `execve()` call, checks and sees that gcc is migratable, and then hands the process to the PANTS system. We also modified `ld`, the link editor program used by make. We modified it to wait until all the files were present on the NFS mount before it attempted to link them. This added robustness to the compilation, but in practical use we feel it is unnecessary as a properly tuned NFS server can keep up with the demand placed on it during the compile.

The Linux kernel source tree was located on the NFS mount and all output files were sent to this same location. This made all of the files available to all the nodes. The build was started from any node in the cluster by simply typing `make vmlinux` from the NFS mounted Linux kernel directory.

4 Results

Results are based off the log files produced by the PANTS daemon. These log files contained metric information for CPU, memory, disk, context switching, and interrupts at five second intervals. We measured the load on an idle machine and obtained the following baseline measurements for each metric: CPU load at 0%, I/O load at 250 blocks/sec, memory load at 0 blocks/sec, interrupts at 103,000 interrupts/sec, and context switches at 950 switches/sec.

4.1 Micro benchmark results

The micro benchmarks were run on unloaded systems, only running basic system services. These services included the inetd daemon, which is responsible for rsh, the system logging deamon, crond, swapd, and a few other services. To see a listing of all the processes running on each system in the cluster see Appendix B.

4.1.1 CPU benchmark

The CPU benchmark displayed the typical behavior of the PANTSD system. As each node surpassed 95% CPU utilization, it was removed from the list of free nodes and did not take any more processes. The total load of the benchmark was divided evenly between all four nodes. See Figure 2 for the average load cluster wide for each metric during the benchmark.

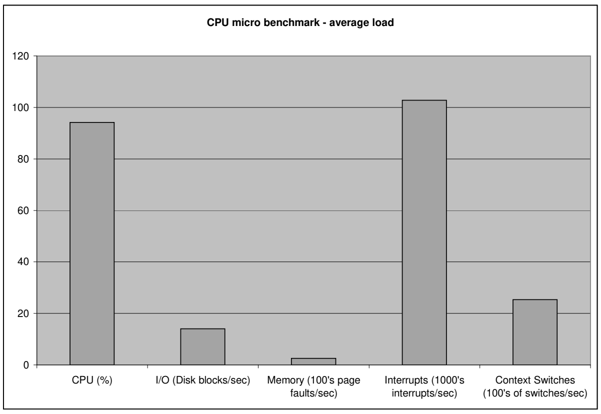

Figure 2: CPU micro benchmark

4.1.2 Memory benchmark

The memory benchmark was first run with only the CPU metric enabled, with the CPU threshold at 90%. The parent node never became unavailable, as CPU utilization stayed around 7%. However, memory usage on the parent node was extremely heavy, and the machine became unresponsive to user control for most of the test. The total runtime for this test was 1 hour, 33 minutes and 43 seconds. See Figure 3 for the average load cluster wide for each metric during the benchmark. While the memory load is high, the I/O is also quite loaded. This is because the memory metric is page faults per second and page faults by their nature effect the load on the I/O subsystem. The standard deviation for the memory metric in this benchmark is approximately 12,000 page faults/sec, which is expected as no load is being distributed.

The memory metric was then enabled for the second run, with the threshold set at 10000 pages/second. After the first process was spawned the parent node (London) showed 50000 pages/second, switched to the unavailable state and was removed from the free node

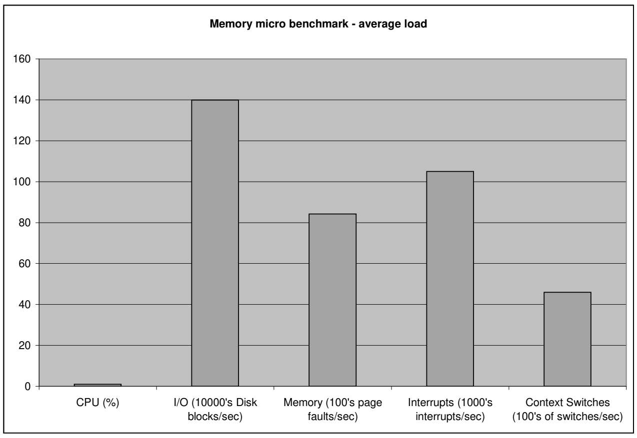

Figure 3: Memory micro benchmark with PANTS default policy

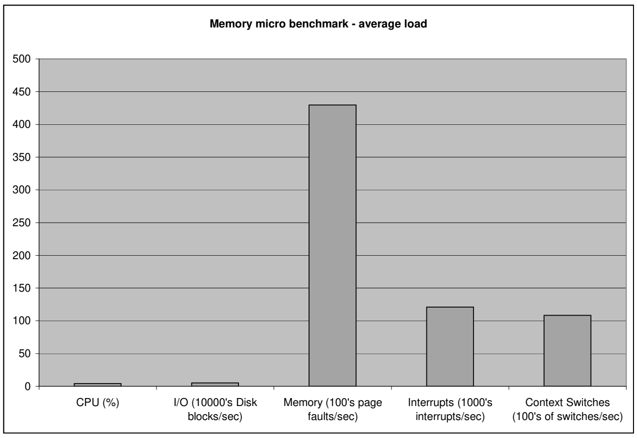

Figure 4: Memory micro benchmark with PANTS new policy

list. The second process was then passed to Rome, which immediately became unavailable when it displayed 100000 pages/second. The third process was passed to London, and the fourth to the leader node, Paris. Total execution time for this run was 10 minutes and 53 seconds. The average load for each metric cluster wide is shown in Figure 4. The memory load average is increased as the previously idle nodes are now participating, and the standard deviation is much lower at 575 page faults/sec. The load on the I/O system is much lower for two reasons: the load on memory is not as high so not as many disk accesses are necessary to swap pages, and some machines in the cluster have more memory than others and not as many swaps are needed are required on these machines.

4.1.3 I/O benchmark

The disk benchmark was first run with only the CPU metric enabled, with the CPU threshold at 90%. CPU utilization stayed between 4% and 8% for the duration of the run. The parent machines was fairly responsive to user input until the second process was spawned, at which time it stopped responding to other commands until the test

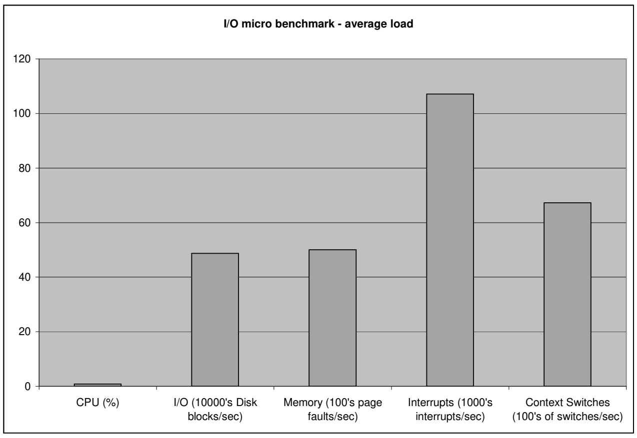

Figure 5: I/O micro benchmark with PANTS default policy

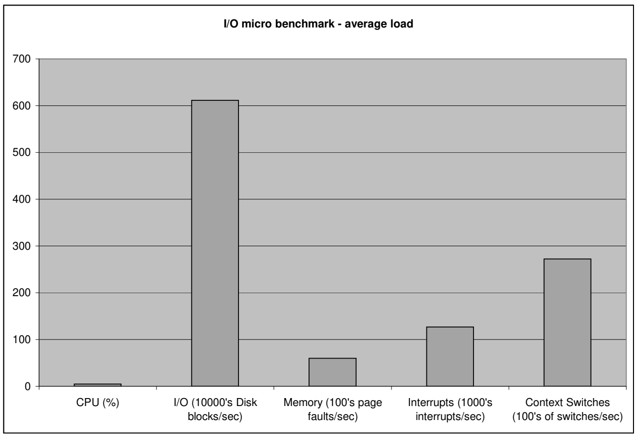

Figure 6: I/O micro benchmark with PANTS new policy

completed. The average load for each metric cluster wide is shown in Figure 5. While the average for I/O load is low, the standard deviation is very large.

For the second run the disk threshold was set to 10000 blocks/second in addition to the CPU threshold at 90%. Immediately after the first process was spawned the parent node displayed 20000 blocks/second and because unavailable. The second process was passed to Beowulf, which immediately displayed over 100000 blocks/second and also immediately becomes unavailable. The third process was passed to Rome, and the fourth process to the leader, Paris. Cluster wide load averages can be seen in Figure 6. The average is considerably increased, but the standard deviation is greatly decreased, as this load is more evenly distributed.

4.2 Linux Kernel Compile as a Real-World Benchmark

Our performance evaluation of the distributed compilation of the Linux kernel involved several steps. We obtained timing measurements for several variations of the compilation for comparison purposes. Our first variation was a compile were the files where stored on

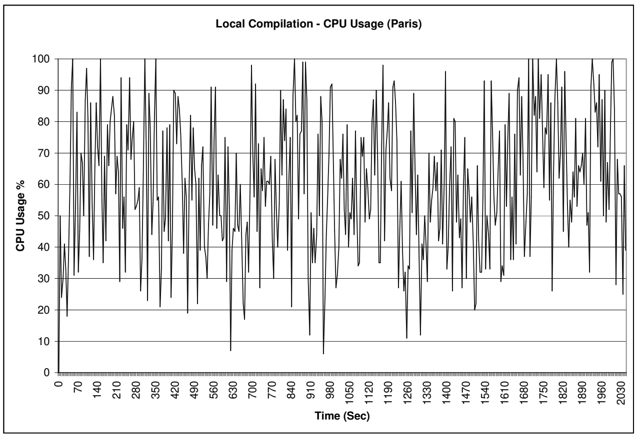

Figure 7: CPU Usage: Local

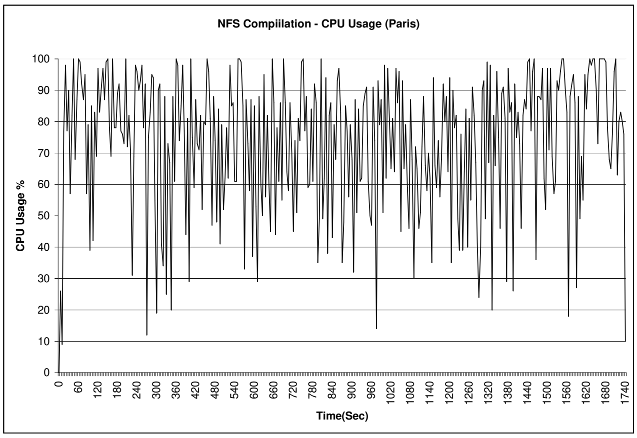

Figure 8: CPU Usage: NFS

the local hard disc and no PANTS load distribution. The second was a compile of the kernel tree which was stored on the NFS mounted disc. The next three were all compiles where the source was mounted over NFS with PANTS running. In the first of these three the gcc binary was not migrateable, in the second PANTS used the default load metrics, and finally PANTS used our new load metrics and policy.

The Linux kernel version was 2.4.18, the build was configured to compile a kernel image identical to those on which the machines in the cluster currently ran. Overall, 432 files were compiled, with the mean source file size being 19KB. Five compile times were averaged to obtain the results shown.

Figure 7 shows the CPU usage for a local disc compilation, which shows considerable load being imparted on the CPU. However, when the kernel source code is accessed via NFS, some delay is induced and subsequently the processor is not loaded, as can be seen in Figure 8.

The compilations with PANTS running but migration disabled was achieved by flagging the gcc binary as not migratable. This evaluation was done to give baseline results and

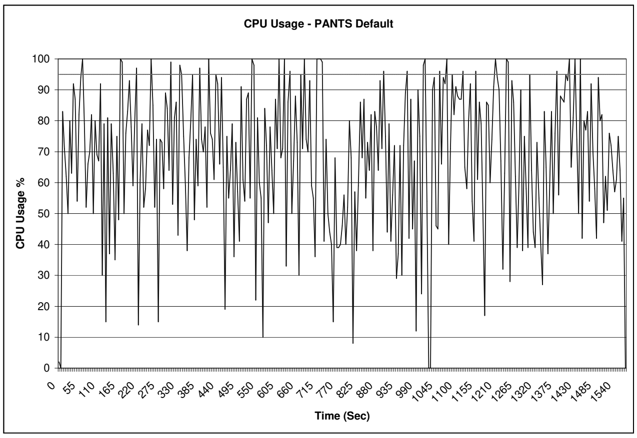

Figure 9: CPU Usage: PANTS default

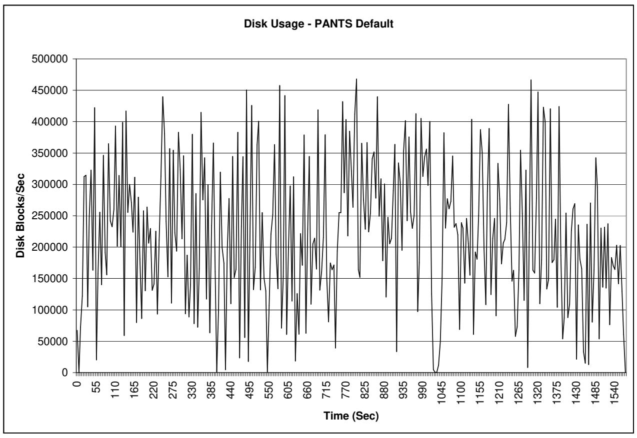

Figure 10: Disk Usage: PANTS default

an idea of the overhead involved with running the PANTS daemon. We conclude that there was very little overhead incurred: the compile with PANTS running took 5 seconds longer out of a total of 1520 seconds than an NFS compile. This overhead is the slight delay induced by prex's checking the gcc binary for migratability, which was done at least 432 times during the course of the compile. Additional checks were made of the make binary, and the utilities that make uses, such as ld and ar.

The next evaluations were with PANTS running using the default load metrics. Specifically, as noted in section 2.2.3, the CPU metric was the only one being used and the threshold was set at 95%. As shown in Figure 9, the CPU usage during a compile rarely goes above 95%, and only at these times are process migrated. This lack of migration yields no throughput increase or load distribution throughout the cluster. In Figure 10, it is clear that the disk is being heavily loaded. In addition to the disk usage, the memory is being loaded and there is also a large number of context switches and interrupts generated during the compilation.

4.2.1 Establishing thresholds

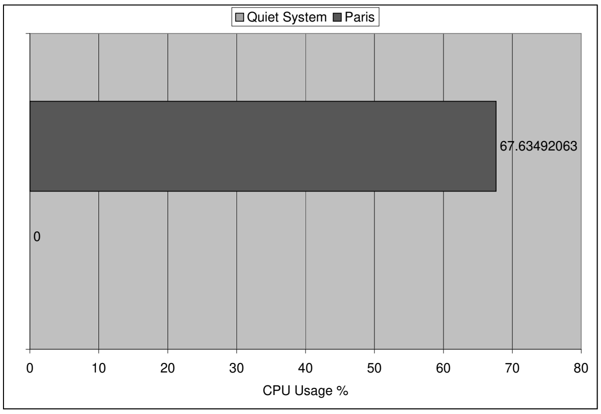

Figure 11: CPU

In order to evaluate the performance of our new load sharing policies, we first had to establish thresholds for each load metric. Figures 11 - 15 compare the average load on the originating node when using the default load sharing policies with the load of a quiet, unloaded system. The unloaded measurements were obtained by running the PANTS daemon and measuring the load over a one hour long period of time. Particularly for the context switching and interrupt metrics it was important to determine the baseline for these metrics on an unloaded system. We then took an iterative approach, moving the threshold from this baseline towards the average load we obtained with the default polices. If these thresholds were too low, the remote nodes would become too loaded too quickly and most of the load would stay on the originating node. Eventually we obtained the following thresholds: CPU Usage at 95%, I/O at 1000 blocks/second, memory at 4000 pages/second, interrupts at 115000/second, and context switches at 6000/second.

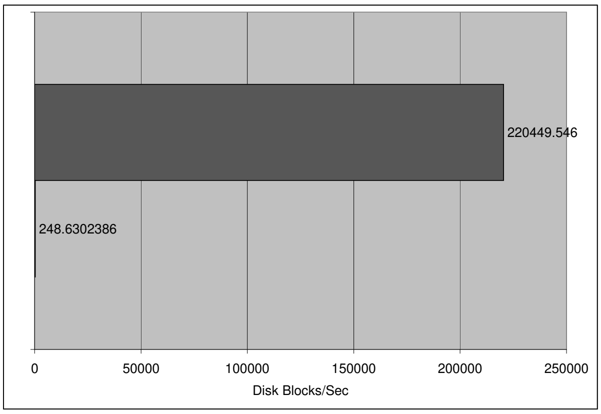

Figure 12: I/O

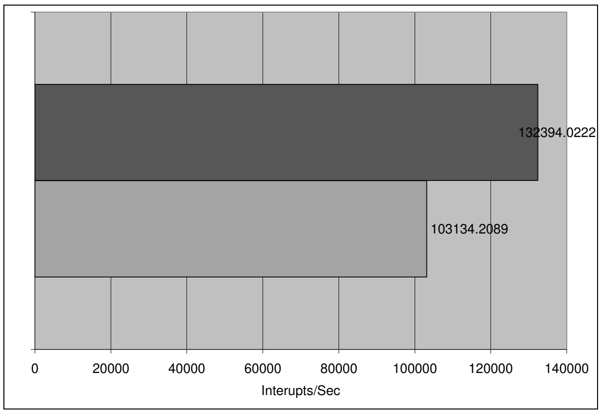

Figure 14: Interrupts

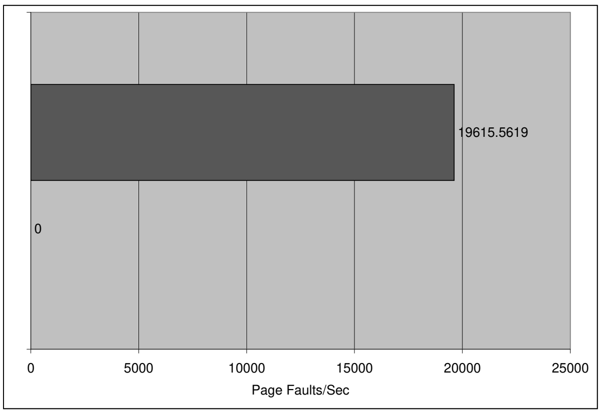

Figure 13: Memory

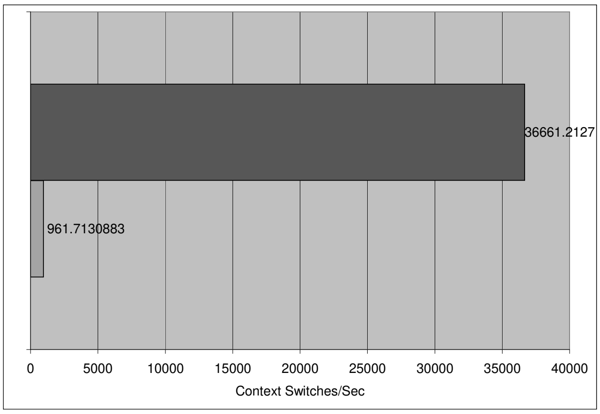

Figure 15: Context Switches

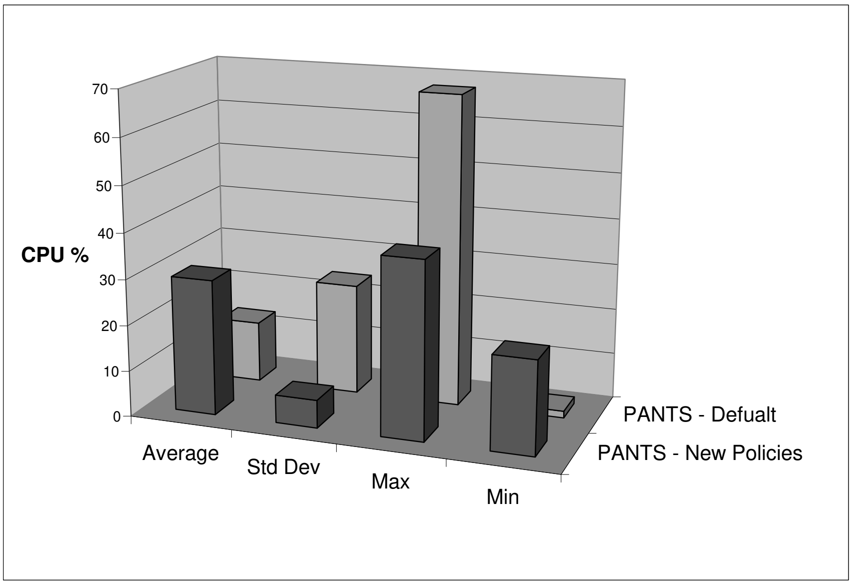

Figure 16: Cluster CPU Load Average

4.2.2 Results of new load metrics and policies

Using our new load metrics and policies the throughput and load distribution is dramatically increased. Figures 16 - 20 show a system level perspective of the results of our new metrics and policies. These figures compare the average load for each metric using the PANTS default policy, and the new policy and metrics we created. The standard deviation is decreased considerably, and the maximum and minimum values are brought closer to the average.

Several metrics show interesting behavior: the I/O and memory load average went down using our new policy. This may seem counter intuitive, but can easily be explained by the fact that some machines in the cluster have more RAM then others. This means that these machines don't have to use swap space and very few page faults are generated.

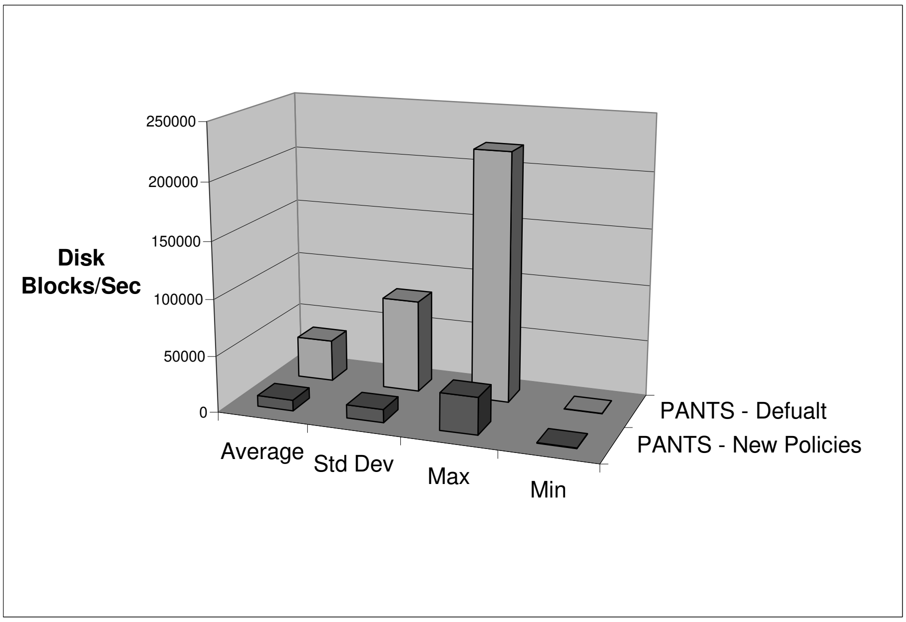

Figure 17: Cluster I/O Load Average

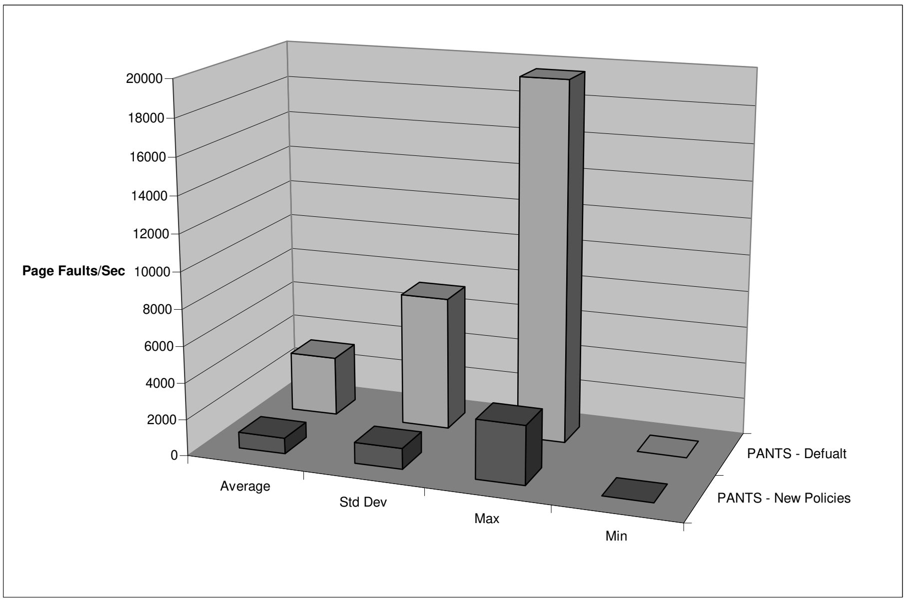

Figure 18: Cluster Memory Load Average

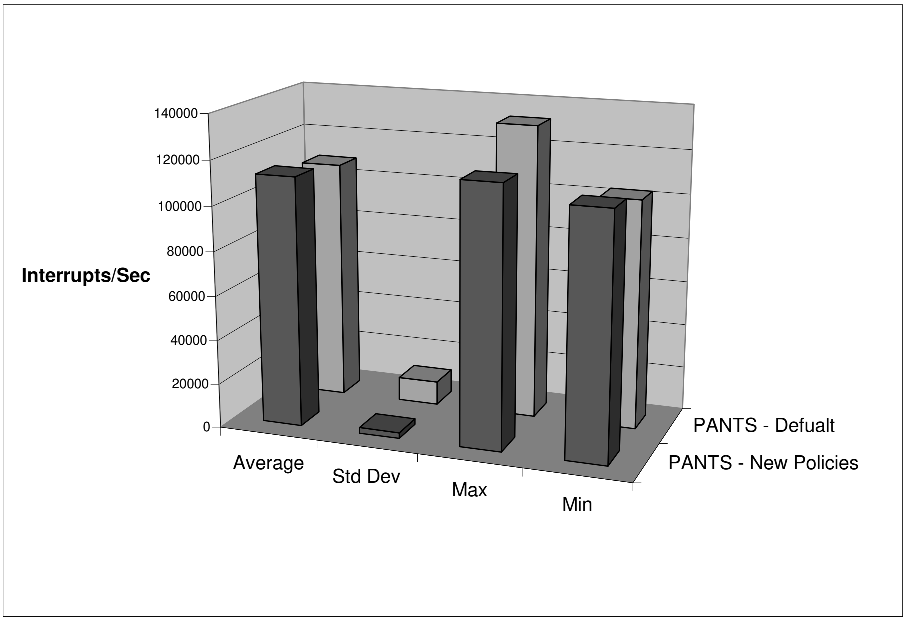

Figure 19: Cluster Interrupts Load Average

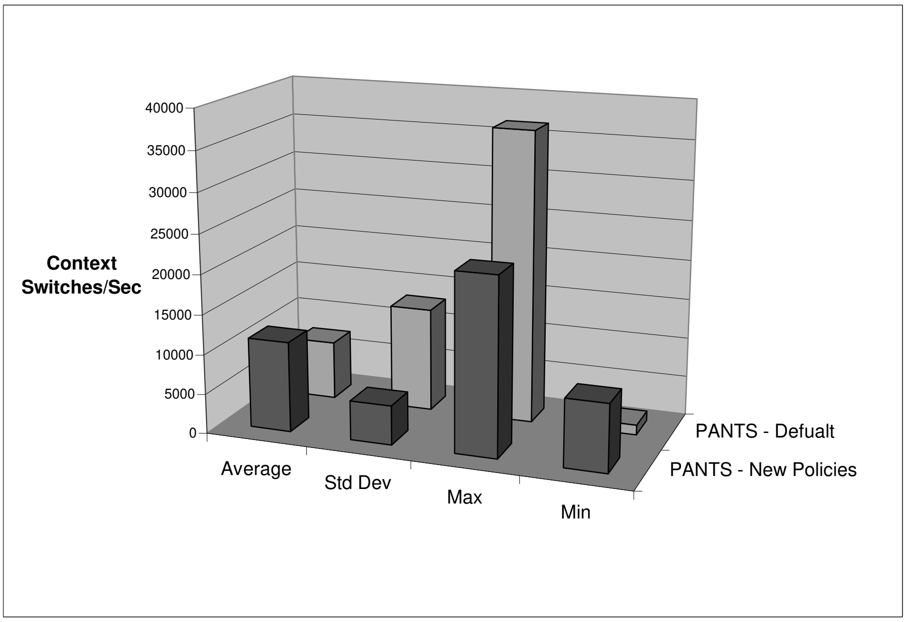

Figure 20: Cluster Context Switches Load Average

There still remains some amount of variance in the load between different machines in the cluster using our new policy. This is due to the fact that there is some load which cannot be migrated from the originating node. This load includes that of running the `make` program and managing over 400 `rsh` remote executions.

In Figure 21, we compare the average time it took for each compilation method. From a user's perspective, our new metrics and polices result in a 54% reduction in the compilation time over the default policy, that just measured the load on the CPU. Furthermore, we decrease the time of the local disc compilation by 56%, and the NFS compilation by 56%. Figure 21 summarizes the total time for these experiments.

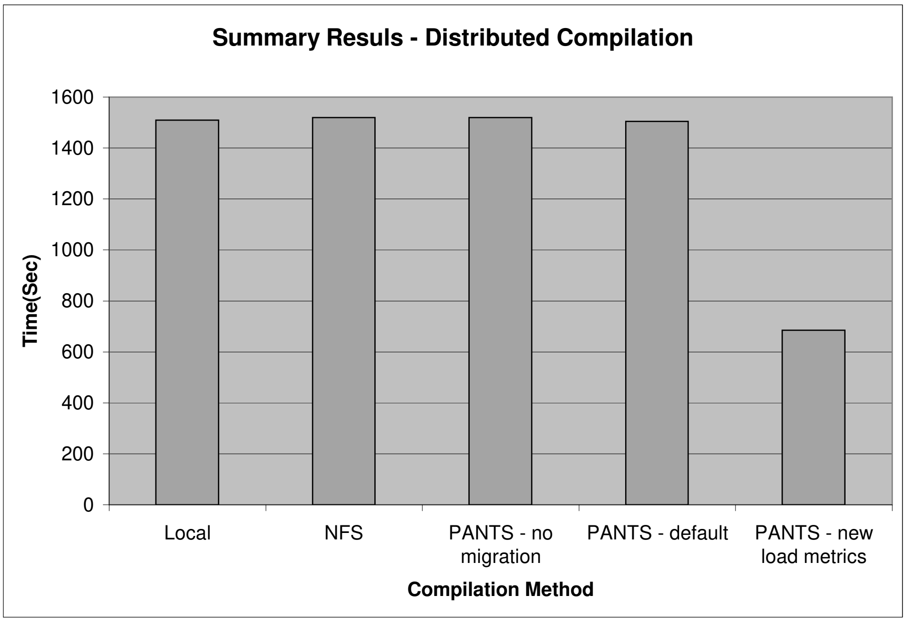

Figure 21: Summary of Linux kernel compilation Application Results

5 Conclusions

A Beowulf cluster is a distributed system consisting of inexpensive computers networked together cheaply, usually via ethernet. It is often desirable in such systems to distribute workload throughout the cluster.

Ideally, the distribution of workload tries to share load equally among all the machines in the cluster, decreasing response times and increasing overall throughput. PANTS Application Node Transparency System [DHV00] is a load distribution system which strives to remove the need for expertise required by other load distribution mechanisms [BG].

By default, PANTS used the `/proc` filesystem to obtain a count of jiffies (1/100ths of a second) that the CPU spent processing in user mode in the past 5 seconds. PANTS then calculated the total jiffies and found the percentage of the user mode jiffies to the total. If this percentage was over 95% then PANTS considered the node loaded or “busy”; otherwise it was considered “free”. Any workload that did not generate CPU load would not be shared amongst other machines in the cluster.

However, programs that impart load on the CPU of a machine are not the only applications that are desirable to run on Beowulf clusters. An application could read or write many files to disk imparting load on the I/O subsystem, maintain large data structures loading memory, or cause system events such as interrupts and context switches. We implemented ways to measure these additional types of load and included them in the PANTS load sharing policy.

We developed micro benchmarks to test our new load metrics and to get a better understanding for how the system was behaving when loaded. We then developed a macro benchmark, which was more of a realistic mix of load. Finally, we chose a real world application as a benchmark.

For our real world application we chose to evaluate the performance a distributed compilation of the Linux kernel with PANTS. When using the default PANTS load metric and policy, only examining CPU load, we found that there was very little process migration. Subsequently, there was no increase in throughput and no sharing of load throughout the cluster.

Using our new metrics and policy we achieve better throughput, decreasing the length of the compile by more than 50% from the default policy. From a system level perspective, we lowered the standard deviation from the average cluster-wide load for each of the metrics considerably sharing load very well amongst the nodes in the cluster. Clearly, including I/O, memory, context switches, and interrupt load metrics has many benefits when used in load distribution.

6 Future Work

Preemptive migration is the term used describe the act of stopping a process that has started execution, moving it to another machine, and resuming the execution. Some load distribution algorithms make use of preemptive migration to allow overloaded nodes to send processes currently running to other nodes which were busy when the execution started but have become free. PANTS may benefit from using preemptive migration, but it has not yet been implemented because most preemptive migration techniques are architecture bound. In the latest development versions of the Linux kernel (2.5.*) there is some support for preemption, perhaps this mechanism could be utilized by PANTS.

Including a network component of the load metrics may be beneficial to PANTS. The current metrics may overlap network usage somewhat, as context switches and interrupts are caused by network activity. This metric could be as simple as parsing information retrieved by the `ifconfig` command, or slightly more elegant by making use of the `/proc/net` information.

Perhaps the most benefit can be found in the use of adaptive thresholds. Currently, the thresholds used by PANTS are static, although they can be modified without restarting the daemon. If the thresholds were adaptive PANTS might be able to respond more appropriately when the load on the system fluctuates.

References

[BG] Ewing Lusk Bill Gropp. The Message Passing Interface (MPI) standard. Online at: http://www-unix.mcs.anl.gov/mpi/index.html.

[DHV00] K. Dickson, C. Homic, and S. Bryan Villamin. Putting PANTS On Linux: Transparent Load Sharing In A Beowulf Cluster. *Major Qualifying Project CS-DXF-9918*, May 2000.

[dS96] L. P. P. dos Santos. Load Distribution: a Survey. Technical Report UM/DI/TR/96/03, Universidade do Minho, 1996.

[ELZ86] D. Eager, E. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, 15(5), May 1986.

[ELZ88] D. Eager, E. Lazowska, and J. Zahorjan. The limited performance benefits of migrating active process for load sharing. *SIGMETRICS*, May 1988.

[FW95] David Finkel and Craig E. Wills. Scalable Approaches to Load Sharing in the Presense of Multicasting. *Computer Communications*, 8(9), September 1995.

[ML87] M. W. Mutka and M. Livny. Profiling workstations' available capacity for remote execution. In *Performance '87, Proceedings of the 12th IFIP WG 7.3 Symposium on Computer Performance*, 1987.

[Moy] J. Moyer. PANTS Application Node Transparency System. Online at: http://segfault.dhs.org/ProcessMigration.

[PVM] PVM. Parallel Virtual Machine. Online at: http://www.epm.ornl.gov/pvm/pvm_home.html.

[RR96] R. Riedl and L. Richter. *Classification of load distribution algorithms*. IEEE Computer Society Press, Braga, Portugal, January 1996.

[SKS92] N. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *IEEE Computer*, December 1992.

[STW01] Michal Szelag, David Terry, and Jennifer Waite. Integrating Distributed Inter-Process Communication with PANTS on a Bewoulf cluster. *Major Qualifying Project CS-DXF-0021*, March 2001.

A PANTSD Configuration File Options

By default the PANTS daemon reads configuration information from `/etc/pantsd.conf`. When starting pantsd you can supply a pathname to a config file for pants to use (example: `/home/jnick/my_conf_pantsd.conf`). The daemon will insert default values if an option is not specified in this file.

Here is the format information for the config file, with what the key is for the config file, the name of the member in the options structure it represents, and the default value.

``multi_leader_address", PANTS_MULTI_LEADER, 224.10.0.1

``multi_available_address", PANTS_MULTI_AVAIL, 224.10.0.2

``query_port", PANTS_QUERY_PORT, 0x9F

``transfer_port", PANTS_TXFER_PORT, 0x9F7

``highwater_mark", PANTS_HWM, 10

``lowwater_mark", PANTS_LWM, 2

``leader_timeout", PANTS_LDR_TIMEOUT_SEC, 5

``leader_timeout_usec", PANTS_LDR_TIMEOUT_USEC, 0

``interval", PANTS_INTERVAL_SEC, 5

``interval_usec", PANTS_INTERVAL_USEC, 0

``forced_update_count", PANTS_FORCED_UPDATE_COUNT, 6

``local_port", PANTSD_LOCAL_PORT, 0x9F8

``max_free_node_list", PANTS_LISTMAX, 128

``threshold", THRESHOLD, 95

``weight_0", WEIGHT_0, 0

``weight_1", WEIGHT_1, 0

``weight_2", WEIGHT_2, 0

``log_stderror", LOG_STDERROR, ``no"

``log_metrics", LOG_METRICS, ``yes"

Lines starting with # are treated as comments and are ignored.

The following is an example of valid pantsd.conf file:

```
# Set threshold

threshold 75

# Log to standard error

log_stderror yes
```

PANTSD responds to the following signals:

SIGINT - terminates the deamon

SIGHUP - restarts the deamon, causing the config file to be re-read, and all load measurements to be reset

SIGUSR1 - causes the daemon to re-read the config file

SIGUSR2 - toggles whether or not the daemon logs load metrics

SIGTRAP - toggles whether of not the daemon also logs to stderr

B Process listing

```
USER        PID %CPU %MEM   VSZ  RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.2  1784  336 ?        S    Apr14   0:03 init
root          2  0.0  0.0     0    0 ?        SW   Apr14   0:00 [keventd]
root          3  0.0  0.0     0    0 ?        SWN  Apr14   0:00 [ksoftirqd_CP
root          4  0.0  0.0     0    0 ?        SW   Apr14   0:02 [kswapd]
root          5  0.0  0.0     0    0 ?        SW   Apr14   0:00 [bdflush]
root          6  0.0  0.0     0    0 ?        SW   Apr14   0:00 [kupdated]
root          7  0.0  0.0     0    0 ?        SW<  Apr14   0:00 [mdrecoveryd]
root        390  0.0  0.2  1968  336 ?        S    Apr14   0:04 syslogd -m 0
root        395  0.0  0.0  1864   96 ?        S    Apr14   0:00 klogd -2
rpc         409  0.0  0.0  2136   56 ?        S    Apr14   0:00 portmap
rpcuser     427  0.0  0.0  2256   88 ?        S    Apr14   0:00 rpc.statd
root        500  0.0  0.0     0    0 ?        SW   Apr14   0:00 [rpciod]
root        501  0.0  0.0     0    0 ?        SW   Apr14   0:00 [lockd]
root        551  0.0  0.1  2296  144 ?        S    Apr14   0:00 /usr/sbin/aut
daemon      563  0.0  0.1  1920  160 ?        S    Apr14   0:00 /usr/sbin/atd
root        585  0.0  0.8  3848 1048 ?        S    Apr14   0:00 xinetd -staya
root        625  0.0  0.8  8768 1104 ?        S    Apr14   0:00 sendmail: acc
root        638  0.0  0.2  1832  352 ?        S    Apr14   0:00 gpm -t ps/2 -
root        650  0.0  0.3  2248  440 ?        S    Apr14   0:00 crond
xfs         686  0.0  0.2  6440  352 ?        S    Apr14   0:00 xfs -droppriv
root        712  0.0  0.1  1736  232 tty1     S    Apr14   0:00 /sbin/mingett
root        713  0.0  0.1  1736  232 tty2     S    Apr14   0:00 /sbin/mingett
root        714  0.0  0.1  1736  232 tty3     S    Apr14   0:00 /sbin/mingett
root        715  0.0  0.1  1736  232 tty4     S    Apr14   0:00 /sbin/mingett
root        716  0.0  0.1  1736  232 tty5     S    Apr14   0:00 /sbin/mingett
root        717  0.0  0.1  1736  232 tty6     S    Apr14   0:00 /sbin/mingett
root      12617  0.0  1.5  4232 1920 ?        S    01:04   0:00 in.rshd
jnick     12618  0.0  0.9  4136 1240 ?        R    01:04   0:00 ps aux
```