Efficient Adaptive Resource Management for Linux Systems

Benjamin Smith
Jack Tan (Faculty Mentor)
Computer Science Department
University of Wisconsin
Eau Claire, WI 54702
{smithbl, tanjs} @ uwec.edu

Abstract

Adaptive resource management is a method of improving the performance of computers by allowing the operating system to change its settings to better fit real use requirements. Any set of tasks will have different resource needs, and settings which will provide the best performance. This research explores ways to enhance the Linux operating system to efficiently analyze system usage in real time, avoiding time-consuming analysis which may yield unusable results.

The prototype created determines performance of the system using existing measures for its analysis and adjustment of parameters. These include memory usage, and time a process spends waiting. As long as performance criteria can be measured and parameters can be adjusted quickly, new performance measures and adjustable parameters can be added and managed by the system. The design allows for turning off analysis when an adequate set of parameters is found, which can then be used without a penalty.

1 Introduction

Several different methods of managing resources are used by an operating system. These methods attempt to efficiently manage the CPU sharing policies, memory management, and input/output. They allow applications to get as much useful work done as possible. Users have varied views of what constitutes good performance from their computers. They have varied needs, and run applications which have different requirements.

Adaptation comes in two forms. First, the system can change its settings and policies to better overall fit the needs of applications which it runs. Second, in this project, different policies can be used for different tasks. This means that what works well for one process may not work well for others. The system attempts to determine what settings and running parameters will work best for each task individually.

2 Background and Terminology

Modern operating systems have the ability to perform multiple tasks as the same time. This is called multitasking. When only one processor is present in a system this is done by dividing the time spent between running tasks. The amount of time spent on each process is called its time slice, or time quantum. Longer time slices reduce the percentage of time that the system spends switching between processes. This is known as a context-switching. Shorter time slices reduce response time. Actions by the user and the reactions from the system are more quickly seen. The illusion of multitasking is more realistic. The amount of time slice each process is given is determined by its priority. This priority is then adjustable by the use of a nice value. In practice, the nice value is rarely used, since no programmer really wants to have their program execute less. The nice value is used in the project for that very reason. The nice value, and its corresponding impact on priority, are already programmed into the Linux operating system, and can be used to influence performance [1, 3].

Virtual memory is used in the Linux operating system to allow more programs to run than otherwise would be possible given the amount of memory installed in a system. Hard drive is used to temporarily store values when another process needs memory. When those values are needed again, they can be brought back into regular memory. This process is called swapping. Swapping is very time consuming, since accessing data from a hard drive is much slower than accessing memory. Therefore, it is advantageous to reduce the amount of time spent swapping as much as possible.

The current Linux scheduler only recomputes priorities one time for each processes tour through the cycle of scheduling. This is one of the features of the Linux kernel. Other operating systems recompute priorities for every process all the time. The highest priority process always runs first. Processes which require immediate attention like a key press are given priority so that they can be taken care of without delay. They also finish quickly and use few resources, so it usually isn't a problem [3].

3 Problem

Measuring performance in real time is difficult, as all measuring requires the use of resources. More accurate measurements consume more resources. In meeting the goal of creating an operating system which adapts itself in real time to better fit user demands, the adaptations themselves require resource use.

Other obstacles exist as well. One important goal of this project is that it avoids changing existing systems which could affect the proper running of existing programs. Programs that currently run correctly before the changes must continue to work correctly after. Programs must not be required to interact with the operating system in a new way. The changes to the operating system should not require new methods of interaction with the OS, or require any new knowledge or techniques used to create software.

Another challenge is that ideal performance measures are difficult at this level. It is hard to determine how well an application is running when only a small slice of what it does is viewable at one time. It is difficult to tell how well a program is using resources when all the operating system has to look at is a few milliseconds of what it accomplished. The kernel presently has no means of measuring how much a given task has accomplished. It only has external indications showing how long it spends waiting for or using resources. This problem is interesting because it offers a simpler solution to present methods of performance analysis. Existing performance analysis is a time consuming process involving creating performance benchmarks, running them, measuring the results, making changes to some running parameters, and then running the benchmarks again. All of this is performed hopefully under controlled conditions to prevent bad measurements which will produce inaccurate conclusions. These controlled conditions almost never occur in actual use. This leads to poor quality results. In addition, this time consuming work must be redone when changes are made to the system which affect performance. Research in this area hopefully will lead to an automation of this process to be used in real-time under actual conditions.

The approach to this problem used in this project involves changing the Linux operating system which already maintains some different measurements which can be used in calculating performance of the system. These measurements are used to calculate performance and this performance is used to evaluate success of running parameters. Different global system contexts produce different running environments for tasks. The same task will run faster or slower depending on the conditions of the overall system when it is run. So, the global environment is kept track of.

The behavior and requirements of certain tasks will be different than other tasks. So, this is measured and kept track of. This allows new processes to use existing high-quality running parameters which were determined when processes with similar behavior were run.

When the global and process specific characteristics are known, a set of data is maintained which contains the average performance of similar processes under similar conditions. This set of data includes running parameters for each group which are

adjusted based on the success or failure of the running parameters used. This allows the system to adapt itself to provide the best parameters.

One limitation on this research is that existing methods work well, leaving little room for improvement. Changes to an operating system allowing adaptation are not the way to achieve significant performance gains. This research measures performance at the scheduler level and very little can be determined about a process when looking at such small fractions of their behavior.

Another limitation on this project is applications themselves. Profiling can determine what portions of code are run most often, and which use the most resources, but would offer no evidence of how an application accomplishes its goals or how efficiently. The focus of this project is on efficient managing of resources as measured by system factors external to the application.

Under certain circumstances, the behavior of the operating system can be significantly improved. When all memory is used by applications the system is forced to use the hard drive to store information temporarily. When processes are swapped in and out of memory frequently, this can have a very significant impact on performance. It is beneficial to swap less frequently, thus reducing the percentage of time spent on this expensive activity. This project keeps track of how much memory is available and how much memory is used by each process. The system then attempts to reduce swapping by influencing the scheduling of processes.

The information stored and maintained by this project can also be of use to developing new systems. Instead of spending time determining good parameters, multiple options can be programmed allowing better solutions to be determined at run time.

4 Methodology

To create a prototype to test whether or not this system is viable, a prototype was made using the following configuration. VirtualPC was determined to be an appropriate platform for making changes to an operating system. The advantages of this system include that an entire running machine state can be saved. It then can be moved to another computer, or previous working systems can be compared with new ones when new tests are developed. One disadvantage of this system is that the simulated environment will affect performance measurements. So far only a slight system-wide slowdown has been observed so far.

The Linux kernel used in this project is the OpenSuse 10.0 open source distribution. This distribution offered good control over installation, and allowed a very minimal setup. The GCC compiler and minimal tools required for programming kernel code were included, in addition to some performance measurement tools.

Several benchmarks were created which allow a wide range of tests to be run. The N-Queens problem is very CPU-bound and requires finding the number of solutions to

fitting N non-attacking queens on an NxN board. The code and memory use in this benchmark are very low. Benchmarks can be run which take anywhere from less than a second, to problems where finding the number of solutions would take years. For a memory intensive benchmark, the Eratosthenes sieve is implemented for finding prime numbers less than a given number. The size of the sieve can be adjusted to use as much or as little memory as desired. Multiple sieves can be run to force the use of virtual memory and the swap file.

5 Data Structure & Storage

The most important data structure used in the prototype contains the performance data arranged into groups of similar process/system environments. This structure is an array of structures, one for each environment grouping. The structures each contain the average performance for processes in that group, the running parameters used, and a value containing the number of times that environment has occurred. This could be useful later in calculations to fill in approximate values to be used as running parameters for environments not yet encountered.

To store the information used for this system, the best placed was determined to be within the Linux scheduler code. This is the natural place since the scheduler is used every time a context switch occurs. The process data structure is already loaded, and values related to the interactivity of each process is already used here for existing operations.

6 Process Structure & Algorithm Design

The main system for storing, calculating, and making use of measurements consists of five steps which take place during every context switch.

Calculating and storing characteristics is done before a process is executed. This is to ensure that the performance measured later is applied to processes in situations similar to those encountered before anything is known about execution. This way when nothing is known about a new process, running parameters will move toward more appropriate for generalized default settings. The characteristics measured are stored in the process data structure itself for later use. These characteristics are then used to determine the appropriate group in which to find its running parameters.

Calculating and applying desired running parameters is done by looking them up in the main data structure. The most significant bits of the environment characteristics are used to calculate the address in the main data structure array which contains the running parameters. Those parameters are changed slightly every time they are used in an attempt to test new values which may prove to work better than the current ones. The new slightly changed running parameters are added to the process data structure for later reference.

The running parameters calculated in the previous step are translated from their single

byte form to whatever meaning they have for execution. In the prototype, this means that the value is divided to fit in the -20 to +19 nice value range used by Linux. In later work, a parameter could be used to determine whether or not to use a particular feature which may prove to perform better in certain environments.

After the process has finished executing for the time being, the performance of its execution is determined. In the prototype, this value is calculated by looking at the number of virtual memory swaps which occurred during execution. More measures can be easily added, but doing so does not necessarily improve the quality of the performance measurement. Only one measure of performance is ultimately calculated in any case, although it may take multiple measures into account.

Using the performance as measured in the previous step, the average performance and running parameters are adjusted. The outgoing process's characteristics are used to find the appropriate entry in the main data structure. The performance measurement is compared with the stored average performance in the appropriate environment. If the new performance is better, the stored parameters are adjusted toward the slightly changed parameters used in the now finished execution. Otherwise, the stored parameters are moved away from the parameters used. This calculation uses the amount of time the execution used to avoid the case where short bursts of execution cause undue changes in the parameters. The time of execution is also used to appropriately adjust the average performance measure stored.

7 Observation and Analysis

Some metrics can be measured regarding system performance at the level of granularity used in this project. The amount of free memory is known, along with the amount of time each process spends waiting. The number of swaps between physical and virtual memory can be counted. When looking from a further out perspective which may include different applications, the observations of performance are less specific, and it becomes more difficult to take advantage of specialized situations. Some measurements do exist however, and can be used to determine a measure of performance.

This project involved the analysis of a number of existing performance measurement tools. Oprofile is a profiling tool which looks at what system events occur while a program is running. Oprofile uses sampling, so for more accurate results it must be run more frequently and use more resources. Sampling will not be used in this project because it is important that all processes are measured for calculating the best running parameters. Top is program which allows a useful general purpose display of CPU and memory use by running processes. The program uses around 5% of the CPU, making it only usable as a guide. Vmstat is quite useful as a very low overhead performance monitor. Using vmstat, it is possible to keep track of CPU, memory, and IO use at intervals as small as one second. Even at this interval, vmstat can be run and provide useful information with no measurable impact on the benchmarks used in this project [2].

Considering some of the benchmarks used for this project, it was observed that running

multiple copies of the same program takes longer than running the same copies run one after another. This is because the multiple copies use up all of the system memory and are forced to use virtual memory, while one copy can fit in memory without a problem. The swapping using the hard drive causes the required time to complete the tasks to take 25 percent longer in some cases. This would indicate that the action of swapping using virtual memory in this case is responsible for the extra time. If the time spent swapping is reduced by using less frequent context switches, less swapping will be done, and overall performance will be increased. In this project, the rate of swapping will be influenced by detecting situations which will create swapping such as low available system memory, or counting the number of swaps. The nice value associated with each process is used in the systems priority calculation, which in turn is used to determine the time slice each process receives. Larger time slices mean fewer context switches and less swapping, making the system faster.

Creating a system which takes advantage of the information found at the time slice level of granularity has some requirements. Environment characteristics need to be grouped to take advantage of similarity between system contexts. For instance, high priority processes being run under low memory conditions are measured separately from low priority processes run with plenty of system memory available.

Characteristics used to determine the system context are broken into two approaches, including global measurements and process specific measurements. These characteristics must measure some value that changes. It is of no use to keep track of values like processor speed or total memory which stay the same throughout the running of the system.

These characteristics, once measured, are stored in the process data structure. This data is then used to determine which group the performance measurements will be stored in. It isn't practical to compare performance of different processes running under different conditions. Overall performance must be a single value. Two or more measures of performance have no meaning, since both could be combined anyway by using a weighted average or some other calculation.

All measurements are stored in a single byte. This is done for a few reasons. First, no floating point is easily available in the kernel. Second, very little precision is needed in most cases. Finally, the more information which can be stored the better, so single bytes work well. More complicated statistical calculations can be done later in user space when time is at a premium, and more sophisticated tools are available.

8 Analysis

The design of the prototype uses only calculations which are O(1), and are also very fast. Adding new performance measures may increase the amount of calculation required, but it will remain a fixed number throughout the running of the system. The changes made for the prototype maintain the O(1) algorithmic complexity which is a major feature in the Linux 2.6 kernel. When adequate running parameters are found, the design allows for

the parameters to be used indefinitely without recalculation. This would require the characteristics to continue to be calculated in order to know which parameters to use. In addition, the characteristic calculations could be turned off if desired, and the data stored could be used to calculate good default running parameters for use by all processes.

9 Conclusion

This project allows for efficient adaptation of resource use policies and parameters at the time slice level. This method could be used to allow an operating system to adapt to the actual use patterns required by application and the user. The measurements, provided efficiently, can be used for any purpose in addition to being used for adaptation. New features could be added to the operating system, and their effectiveness in many environments could be analyzed in comparison to previous solutions without removing the ability to use either. The parameters would not need to be determined through exhaustive benchmarking, but could be derived by the system itself. When it is not known what policy would work better in real use, both could be used and the better solution based on circumstances will be determined. Adding new measurement techniques for performance or characteristics can easily be added to the system and the adaptive system will be able to use them. New running parameters can be added and used without much difficulty.

Existing features of Linux and hardware do not provide features necessary for quality performance measurement and adaptation at the scheduler level. Approximations and guesses based on external factors like memory use are necessary. Significant performance increases should not be expected using adaptation at this level except in certain circumstances where different resource needs can be anticipated based on existing measurements. In addition, Linux and other operating systems already do a good job of managing resource use, so very little overall improvement should be expected.

The method devised in the prototype changes running parameters based on their measured success or failure compared with previous similar situations. Like any system of this type, the results obtained for running parameters may not be optimal. With many combinations of running parameters, not all may be tried by the system. This may lead to the parameters approaching a local solution which may not be the best possible. Finding a better solution may require testing very poor parameters, which may not be an acceptable possibility for the user.

References

[1] Bovet, D., and Cesati, M., *Understanding the Linux Kernel*, 3rd Ed., O'Reilly Media, Sebastopol, CA (2006).

[2] Ezolt, P., *Optimizing Linux Performance,* Pearson Education, Upper Saddle River, NJ (2005).

[3] Love, R., *Linux Kernel Development*, 2nd Ed., Novell Press, Indianapolis, IN (2005).

[4] Silberschatz, A., Galvin, P., Gagne, G., *Operating System Concepts*, John Wiley & Sons, Hoboken, NJ, (2005).
[5] Tanenbaum, A., *Modern Operating Systems,* Prentice Hall, Upper Saddle River, NJ, (2001).