Self-Monitoring and Self-Adapting Operating Systems

Margo Seltzer, Christopher Small
Harvard University

Abstract

Extensible operating systems allow applications to modify kernel behavior by providing mechanisms for application code to run in the kernel address space. Extensibility enables a system to efficiently support a broader class of applications than is currently supported. This paper discusses the key challenge in making extensible systems practical: determining which parts of the system need to be extended and how. The determination of which parts of the system need to be extended requires *self-monitoring*, capturing a significant quantity of data about the performance of the system. Determining how to extend the system requires *self-adaptation*. In this paper, we describe how an extensible operating system (VINO) can use *in situ* simulation to explore the efficacy of policy changes. This automatic exploration is applicable to other extensible operating systems and can make these systems self-adapting to workload demands.

1 Introduction

Today's extensible operating systems allow applications to modify kernel behavior by providing mechanisms for application code to run in the kernel address space. The advantage of this approach is that it provides improved application flexibility and performance; the disadvantages are that buggy or malicious code can jeopardize the integrity of the kernel and a great deal of work is left to the application or application designer. It has been demonstrated that it is feasible to use a few simple mechanisms, such as software fault isolation and transactions, to protect the kernel from errant extensions [15]. However, it is not well understood how to identify those modules most critical to an application's performance and how to replace or modify them to better meet the application's needs.

The ability for applications to modify the kernel is the key to extensible operating systems, but it is also its critical drawback. The power derived from enabling applications to control their own resource allocation and kernel policy can lead to improved performance, increased functionality, or better system integration, but it imposes a tremendous burden on the application developer. The application designer must determine which kernel modules are critical to an application's performance and what modifications to those modules are required.

Determining which parts of the kernel are critical to an application's performance requires an in-depth understanding of the demands that the application places on the operating system. In some application areas, such as database management, these demands are well understood [16]. However, in other application domains or in the case of emerging applications, these demands are not well-understood, and there is no convenient method of obtaining this information.

2 The Need for Data

One of the lessons learned from every major software development project is that there is never enough data about how the system is operating and what is going wrong. Saltzer reminds designers that measurement is more trustworthy than intuition and can reveal problems that users have not yet detected. He encourages designers to focus on understanding the inner workings of the system, rather than relying on the response to changes in workload [13]. Lucas conveys a similar message encouraging system designers to regularly run benchmarks and build in as much instrumentation as possible [11]. While many systems of the 70's heeded these words of wisdom and built in significant performance evaluation tools [8], today's systems show a surprising dearth of native measurement tools.

The 1980's produced a noticeable absence of well-instrumented systems. Today's common research platform, UNIX, initially had very little in the way of performance measurement tools, but now has a standard set of utilities that can provide constant monitoring of system state (e.g., netstat(1), iostat(8), vmstat(8), rstat(1), nfsstat(1), pstat(8), systat(1) [4,5]). These utilities were designed for their output to be read by humans, not processed automatically. However, any system with such a collection of performance monitoring tools could benefit from the off-line analysis of regularly captured output from them.

A second source of readily available data comes from the hardware itself. Several of today's microprocessors contain one or more instrumentation counters that provide invaluable data for performance analysis. For example, the

Pentium processor family contains a 64-bit cycle counter and two 40-bit counters that can be configured to count any one of a number of hardware events such as TLB misses, cache misses, and segment register loads [10]. The Sparc [17] and Alpha [6] microprocessors also have performance counters, although neither has as extensive a set as those available on Pentium processors. Regular monitoring and collection from these hardware counters can also provide a source of information and insight into system behavior.

3 Self-Monitoring

This section introduces methods for making an operating system self-monitoring. The details are presented in the context of the VINO operating system, on which a prototype self-adapting system is being built. However, the principles and approaches are applicable to any number of systems that support extensibility (e.g., SPIN [2]).

VINO is an extensible operating system designed to provide resource-intensive applications greater control over resource management. VINO supports the downloading of kernel extensions (*grafts*), which are written in C++ and protected using software fault isolation. To facilitate graceful recovery from an extension failure, VINO runs each invocation of an extension in the context of a transaction. If the invocation fails or must be aborted (e.g., because it is monopolizing resources), the transaction mechanism undoes all actions taken by the invocation of the extension [15].

The VINO kernel is constructed from a collection of objects and consists of an inner kernel and a set of resources. VINO provides two different modes of extensibility. First, a process can replace the implementation of a member function (method) on an object; this type of extension is used to override default policies, such as cache replacement or read-ahead. Second, a process can register a handler for a given event in the kernel (e.g., the establishment of a connection on a particular TCP port). Extensions of this type are used to construct new kernel-based services such as HTTP and NFS servers.

The VINO approach to self-monitoring and adaptability takes advantage of the extensible architecture of the VINO system providing the following features:

1. Continuous monitoring of the system to construct a database of performance statistics.
2. Correlation of the database by process, process type, and process group.
3. Collection of *traces* and *logs* of process activity.
4. Deriving heuristics and algorithms to improve performance for the observed patterns.
5. *In situ* simulation of new algorithms using logs and traces.
6. Adapting the system according to results of simulation.

These steps are described in more detail in the following sections.

3.1 Monitoring

Each VINO subsystem includes a statistics module that maintains counts of all the important events handled by the subsystem. For example, the transaction system records the number of transactions begun, committed, and aborted, as well as the number of nested transactions begun, committed, and aborted. The locking system maintains statistics about the number of lock requests and the time to obtain and release locks. Each module in the system records the statistics relevant to the module and provides interfaces to access these statistics.

The first step in making VINO self-monitoring is to periodically record the statistics for each of the kernel's modules and accumulate a database of performance activity. In VINO, this mechanism can be provided by constructing an event graft (one that responds to a timer event) that polls the kernel modules and records the statistics it collects. We can factor out the overhead of the measurement thread itself by running the measurement graft in its own thread, and using our normal thread accounting procedures.

3.2 Compiler Profile Output

The second source of system performance information comes from the compiler. Harvard's HUBE project has produced a version of the SUIF compilation system [9] that collects detailed profiling information. By compiling VINO using the SUIF compiler, we can collect detailed statistics concerning code-path coverage and branch prediction accuracy in the kernel. These statistics augment those collected with the measurement thread described in the previous section.

3.3 Tracing and Logging

The measurement thread output and compiler profile output represent static data that characterizes the behavior of the system. The next task is to capture dynamic data about the behavior of the system. We use VINO's grafting architecture to facilitate the collection of this dynamic data.

We attach simple grafts to the inputs and outputs of all modules in the system. On input methods, these grafts

record the incoming request stream and then pass the requests to the original destination module. This record of incoming requests, called a *trace*, captures the workload to a given module and is used to drive that module or its replacement during simulation. Similarly, grafts placed on output methods of a module record the outgoing messages or data before passing them along to the next module. This record of outgoing results, called a *log*, captures the results of a particular module and is used to compare the efficacy of a number of different modules or policy decisions within the module. The set of traces and logs are then available for simulation.

3.4 Simulation

The combination of statically gathered data, traces, and logs creates a complete picture of what the system is being asked to do. The next step is to evaluate the currently implemented policy and determine its efficacy and the potential for other possible policies.

The VINO grafting mechanism provides the ability to perform *in situ* simulation. This is a significant improvement over conventional simulation methodology where entirely separate simulation systems are typically used to evaluate design decisions. Since VINO provides the mechanism to replace a kernel module on a per-process basis, a simulation process can simply replace the module under investigation with an alternate implementation, rerun a trace, and record the result log that would be produced if the system used the alternate implementation. The result logs from multiple simulation runs are then compared to determine which of the simulations produced the best results.

Most modules inside the VINO kernel can be instantiated as simulation modules. Simulation modules are identical to real modules except that they do not modify the global state. Therefore, simulations can run without affecting the rest of the system. Since the simulators and real modules share much of the code, we do not increase code size substantially and our results are realistic. Similar technology has been employed in the design and analysis of file systems, with encouraging results [3].

Modules that support simulation must consist of two logical sets of states: the first set is writable by both the real and simulation instances of the module and is duplicated for each instance of such modules, the second set is writable only by the real instance of the module because the states are shared system-wide. Returning to the example of a buffer cache module, meta-data such as buffer headers falls into the first category, while the actual data falls in the second.

The simulation modules run without affecting the rest of the system and are not affected by other activities in the system. This allows the simulations to be run reproducibly under many different configurations.

4 Self-Adaptation

The four components described in Section 3 provide the framework for building a self-monitoring and adaptable operating system. The statistics gathered through self-monitoring provide invaluable feedback for identifying performance-critical portions of the kernel. More importantly, this data is gathered in the context of actual workloads, so it reflects the demands of the workload in practice, not the demands under some artificial benchmarking workload. Because data is collected via grafts, we have access to a low-level interface that provides more detailed data than is typically available to user-level utilities.

The first step in making a system self-modifying is to endow it with the proper analytic tools to allow it to detect unusual or problematic occurrences. We use two different types of analysis to make system changes. *Online analysis* takes advantage of the data as it is being collected while *off-line analysis* consists of a post-processing phase. In general, the off-line system is responsible for monitoring long-term behavior of the system, identifying common and uncommon profiles, suggesting thresholds to the online system, and evaluating the feasibility of system changes. The online system is responsible for monitoring the current state of the system, posing "questions" to the off-line system, and identifying trouble spots as they arise. As we will show in the following sections, the online and off-line systems work together, each gathering data or performing an analysis task best suited to that system.

4.1 Off-line Analysis

As described in Section 3.1, one part of our measurement system is a kernel thread that polls each of the subsystems at regular intervals and records the state of the performance counters. After each collection, this data is written to a system-wide database of statistics for later processing. The off-line analysis phase uses this database and specific queries from the online system as its input. The off-line analysis system consists of a collection of user-level processes that process system data to accomplish two goals: construct a characterization of the system under normal behavior and detect anomalous behavior and use that anomalous behavior to suggest performance thresholds to the online system. The latter goal depends significantly on the former goal: in order to identify anomalous behavior, we must have an accurate view of normal behavior.

Let us consider the selection of the measurement interval as an example of how the off-line system works and how it can provide useful information to the online system to change how the system behaves. Our normal system characterization is based on time series analyses of resource utilizations. For each resource (e.g., memory, disk, network, processor) we maintain utilization statistics. Initially, the in-kernel thread collects utilization statistics every 100 ms. Periodically (at least once per hour), the data are collected each millisecond. All the data are written to the system wide database. Each night, the off-line system performs variance analysis. First, it examines the one-ms data and determines if the default measurement interval is sufficiently short (i.e., that it does not fail to capture important utilization patterns). If the current measurement interval is determined to be appropriate, the off-line system performs variance analysis on the normal intervals, to determine if the measurement interval can be increased without loss of information. For each resource, the off-line system feeds this interval information back to the measurement thread, which modifies its behavior according to the recommendation of the off-line system. In this way, the online and off-line systems interact to allow the measurement thread to run as infrequently as possible without loss of important information.

The next task of the off-line system is to examine the day's resource usage profile, identifying any periods of anomalous behavior and informing the online system of suggested thresholds for resource utilization. We have analyzed the behavior of the systems that comprise our central computing facility and found that, in general, system usage follows a regular pattern over the course of a week [12]. Therefore, in analyzing our system profiles, we compare a day's profile to reference profiles constructed from profiles of the previous day, the same day of the previous week, and the same day of the previous month. We use these three profiles to generate a reference profile for the day and then compare the daily profile to the reference profile, looking for periods of abnormal utilization. If we detect abnormally high utilization, we trigger detailed analysis.

In order to perform detailed profile analysis, we decompose the daily profile into per-process profiles. Using these per-process profiles, we determine if the anomaly is caused by a single process, a small group of processes, or by an overall increase in system load. All the results of the profile analysis are then fed back into the off-line system, which derives expected loads and thresholds for the next day. The expected loads are based on the analysis of the current day's profile and the previous week's and month's profile. The thresholds are based on the observed variances in the reference profiles. Finally, the next day's thresholds and expected profiles are fed back into the online system.

4.2 Online Analysis

The online system is responsible for monitoring the instantaneous resource utilization and the rate of change in utilization. In addition, it maintains efficiency statistics such as: hit rates for all the cached objects in the system, contention rates in the locking system, disk queue lengths, and voluntary and involuntary context switch counts. Using the expected behavior and thresholds presented by the off-line system, the online system is responsible for detecting "red flag" conditions; cases where resource utilization is outside the expected variance and still climbing. When this condition is detected, the online system has two options: it can select an adaptation heuristic or it can install the appropriate trace-generating graft for the resource in question, producing data for the off-line system.

The former approach is effective when, as system designers, we have an *a priori*

understanding of the types of algorithmic changes that might be beneficial. These techniques are similar to other dynamic operating system algorithms, such as multi-level feedback scheduling and working set virtual memory management. We discuss these heuristics in Section 5.

The latter approach is the method of last resort, in which we make an effort to develop new algorithms to improve the performance of a particular workload. When the system detects a red-flag condition and has no heuristic to improve it, it generates a trace of the poorly performing module and then signals the off-line system to analyze the trace. The off-line system computes the optimal behavior of the system under the imposed load and compares the optimal behavior to the actual behavior. If the actual behavior is within an acceptable margin of the optimal, then the off-line system concludes that the module in question cannot significantly improve performance alone, and it invokes detailed system analysis to identify other suspect modules.

If, however, the current algorithm is significantly less effective than is optimally possible, our goal is for the off-line system to suggest new heuristics to the online system. Not surprisingly, this is one of the key areas for future research in the development of this system.

5 Adaptation Heuristics

The typical goal of adaptation is to decrease the latency of an application. As a rule, latency is caused by an application waiting for the availability of some resource. Waiting can be caused by the application blocking on user or other input, which is outside the control of the system, or it can be caused by the application blocking on a resource that is unavailable due to an operating system policy deci-

sion. We call the former form of blocking *compulsory* and the latter form *needless*. The goal of adaptation heuristics is to reduce needless blocking.

In the rest of this section we discuss some causes of needless blocking, heuristics for identifying the causes, and methods for decreasing the amount of needless blocking in the system.

5.1 Paging

Techniques for dealing with high paging overhead have been known for decades. In general, if an application is paging, it is assumed that the working set of the application is larger than the number of physical pages assigned to it and that it should be given more physical memory. When allocating additional pages to one application reduces the number of pages for some other application below its working set size, some application is swapped out in order to increase the amount of physical memory available.

Similar techniques can be used in our environment. If an application is paging heavily, we generate a trace of the pages faulted in by the application and the value of the program counter for each page fault. We can collect more detailed traces by unmapping all pages in the address space, generating a full trace of page accesses. Although this technique adds considerable overhead, it provides the complete page access history.

Once we have page access traces, we look for simple, well-known patterns: linear memory traversal or correlation between function calls and page references. In the former case, we perform simple prefetching, while in the latter case, we perform slightly more complex prefetching, faulting in appropriate data pages when the application enters the function for which the data will be referenced.

5.2 Disk Wait

We can make similar system modifications to alleviate disk waiting. When we detect an application that is spending time waiting for disk I/O, we generate the trace of disk block requests, capturing both those that were satisfied in the cache and those that required I/O operations. As in the case of page faults, we then look for common patterns.

If the application is performing a linear pass over a file (which is the most common case [1]), our normal read-ahead policy should already be performing aggressive read-ahead. If the application is spending less time processing each page than it takes the system to read the page from disk, read-ahead can reduce, but not eliminate, the amount of time the application spends waiting for the disk. In this case, we have found a compulsory component of the disk waiting time that can not be removed.

If the application appears to be randomly accessing the file, we compare the traces of multiple runs of the application to determine if the blocks of the file are accessed in a repeatable order. If so, we graft an application-specific read-ahead policy that knows this ordering. Other patterns for which we can easily construct application-specific read-ahead policies include strided reads, common for scientific applications, and clustered reads, where a seek is followed by a fixed-size sequential read.

5.3 CPU-Bound Processes

Even if an application is simply CPU bound, there are techniques we can use to improve its performance. Using SUIF, we can gather information about branch mispredictions. With on-chip counters, such as those on the Pentium, we can gather information about L1 cache misses, code and data TLB misses, and branch target buffer hits and misses.

If we find that the application is suffering from a large number of branch mispredictions or poor code layout, we request recompilation of the poorly performing kernel functions in the context of the particular application [14]. We then install this recompiled kernel segment as an application-specific graft.

5.4 Interrupt Latency

While the time spent waiting for an interrupt to take place is *compulsory*, the time between the occurrence of the interrupt and when the application runs is *needless*. Previous studies of latency point out that slow systems irritate users [7]. By reducing this latency we can improve not only overall application performance, but also overall user satisfaction.

In order to measure this latency, we time-stamp interrupts as they arrive, and compute the difference between this time and when the process to which the interrupt is delivered is scheduled. Our goal is to detect interrupt handling latencies or variance across latencies that are perceptible to users. When we detect such cases, we try to determine the cause and either modify the system behavior appropriately or notify the application of potential areas for improvement.

For example, if a process blocks too long behind higher priority processes, we recommend raising the application's priority. If we find that the process typically faults immediately upon being scheduled, this is a sign that the event handling code is being paged out. In this case, we pin the code pages of the event handler into memory. If the process is not yet ready for the event (i.e., a mouse interrupt arrives before the process performs a select() on the mouse

device), we recommend that the application be restructured to check for events more frequently.

5.5 Lock Contention

If a lock is highly contended (i.e., the queue of processes waiting for the lock is long), there is a problem with the structure of the applications using the lock. When we detect processes waiting on unusually long lock queues, we decrease the granularity of the locked resource (if possible), to reduce contention. Note that this reduction in granularity is frequently possible when the resource in question is a kernel resource whose structure we understand, but may not be possible if the resource is an application resource. In the latter case, we signal the application that contention on the particular resource is abnormally high.

6 Summary

In summary, we can construct a framework in which a system becomes self-monitoring, so that it may adapt to changes in the workload it supports, providing improved service and functionality. We accomplish this by:

1. grafting the necessary components into the VINO system to make it self monitoring,
2. designing and developing a system to collect and analyze performance data,
3. developing heuristics and algorithms for adapting the system to changes in workload, and
4. using *in situ* simulation to compare competing policy implementations.

7 References

[1] Baker, M., Hartman, J., Kupfer, M., Shirriff, K., Ousterhout, J., "Measurements of a Distributed File System," *Proceedings of the 13th SOSP*, Pacific Grove, CA, Oct 1991, pp. 198–212.

[2] Bershad, B., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., Chambers, C., "Extensibility, Safety, and Performance in the SPIN Operating System," *Proceedings of the 15th Symposium on Operating System Principles*, Copper Mountain, CO, Dec. 1995, 267–284.

[3] Bosch, P., Mullender, S., "Cut-and-Past file-systems: Integrating Simulators and Filesystems," *Proceedings of the 1996 USENIX Technical Conference*, Jan 1996, pp. 307–318.

[4] Computer Systems Research Group, University of California, Berkeley, *4.4BSD System Manager's Manual*, O'Reilly and Associates, 1-56592-080-5, 1994.

[5] Computer Systems Research Group, University of California, Berkeley, *4.4BSD User's Reference Manual*, O'Reilly and Associates, 1-56592-075-9, 1994.

[6] Digital Semiconductor, *Alpha 21164 Microprocessor, Hardware Reference Manual*, Order Number: EC-QAEQB-TE. Digital Equipment Corporation, Maynard, MA 1995.

[7] Endo, Y., Wang, Z., Chen, J., Seltzer, M., "Using Latency to Evaluate Interactive System Performance," *Proceedings of the 2nd OSDI*, Seattle WA, October 1996.

[8] Ferrari, D., *Computer Systems Performance Evaluation*, Prentice Hall, Englewood Cliffs NJ, 1978, 44–64.

[9] Hall, M. W., Anderson, J. M., Amarasinghe, S. P., Murphy, B. R., Liao, S. W., Bugnion, E., Lam, M. S., "Getting Performance out of Multiprocessors with the SUIF Compiler," *IEEE Computer*, December 1996.

[10] Intel Corporation, *Pentium Processor Family Developer's Manual. Volume 3: Architecture and Programming Manual*, Intel Corporation, 1995.

[11] Lucas, Henry C., "Performance Evaluation and Monitoring," ACM Computing Surveys, Vol. 3, No. 3, September 1971, pp. 79–91.

[12] Park, L., "Development of a Systematic Approach to Bottleneck Identification in UNIX systems." Harvard University Computer Systems Technical Report, TR–05–97, April 1997.

[13] Saltzer, J.H. and Gintell, J.W., "The Instrumentation of Multics," *Communications of the ACM*, Vol. 13, No. 8, August 1970, pp. 495–500.

[14] Seltzer, M., Small, C., Smith, M., "Symbiotic Systems Software," *Proceedings of the Workshop on Compiler Support for Systems Software* (WCSSS '96).

[15] Seltzer, M., Endo, Y., Small, C., Smith, K., "Dealing with Disaster: Surviving Misbehaving Kernel Extensions," *Proceedings of the Second Symposium on Operating System Design and Implementation*, Seattle, WA, October 1996.

[16] Stonebraker, M., "Operating Support for Database Management," *Communications of the ACM 24*, 7, July 1981, 412–418.

[17] Sun Inc., *STP1021 SuperSPARC II Addendum for use with SuperSPARC and MultiCache Controller User's Manual*, Revision 1.2.1, Sparc Technology Business, A Sun Microsystems, Inc. Business, Mountain View, CA, 1994.