Self-Monitoring in VINO

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Citable link</td>
<td><a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:23574268">http://nrs.harvard.edu/urn-3:HUL.InstRepos:23574268</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at <a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA">http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA</a></td>
</tr>
</tbody>
</table>
Title : Self-Monitoring in VINO

Author : Hany S. Saleeb

TR Report # : TR - 03 -99
Self-Monitoring in VINO

Abstract

Computer system performance is a measure of how well the operating system shares hardware and software resources among the various applications that are running on it. The goal of performance monitoring in the VINO extensible operating system is to make recommendations for improving application performance. This is accomplished by collecting system data through a monitoring agent, automatically identifying conditions causing performance degradation, and presenting evidence to support its conclusions. Once the operating system is well monitored, it may be able to tune itself to improve system performance.

Within the framework of VINO, I describe a system that monitors itself and gathers information about its performance. I show that system monitoring is advantageous for two reasons. First, by utilizing sample user applications, it is capable of warning designers of application bottlenecks and system degradation. Second, the operating system can dynamically self-adapt its own kernel behavior and policies after monitoring access patterns. Thus, the monitoring system aids the application designer in defining performance limitations and adapts kernel policies to improve overall system performance.

1 Introduction

With the rapid emergence and short life cycle of developing new applications, system designers may produce software with little or no understanding of its performance. As a result, applications may not perform optimally due to bottlenecks within the operating system. Engineers, for example, may develop software without realizing that there exists resource contention due to disk bandwidth or excessive memory paging. It is true, however, that some application vendors, such as those in databases have spent a considerable amount of time and effort gathering statistical system data to achieve peak performance, but the vast majority of communities have not. This is especially true for the new generation of applications based on the Web and networking. If such applications were capable of collecting, analyzing and understanding their use of the underlying system, performance may improve.

Determining which parts of the kernel, if any, are critical to an application's performance requires a comprehensive understanding of the resource requirements placed on the operating system. There are some existing statistical monitoring tools presently available in most operating systems that produce valuable data, such as the number of context switches and the bandwidth for I/O requests. Each one, however, is obtained by a different system command and is just a listing of statistics without any context or advice. Allowing the operating system to automatically gather its own performance statistics and analyze its behavior is advantageous [Sel97], because the operating system can investigate a much larger spectrum of system statistics at a faster rate than any human user. This allows the operating system to not only gather and present key statistical metrics but also advise a user on a correct course of action to improve system performance. As a framework, VINO examines the feasibility of constructing such a system that monitors itself and provides application designers with guidance on their application's use of the operating system.
The goal of the automatic performance monitoring mechanism in VINO is to make recommendations for improving application performance. VINO periodically collects and analyzes the collected data. The monitoring module differentiates among CPU-, memory-, management and disk I/O-bound applications and quantifies the system requirements and demands. VINO, then, presents the user a simple report on resource and time utilizations. For example, a monte carlo simulation stresses the operating system in a different manner than a database or a large matrix multiplication operation and, thus, the report pinpoints different bottlenecks in the underlying system. Having identified a potential system bottleneck, the report advises the user to modify specific functions or modules within the kernel via VINO's extensibility mechanism. It is, of course, possible that there is not a single bottleneck in the system and, thus, the report may not advise any extension.

The monitoring system described here is particularly advantageous for VINO for two reasons. First, it is capable of alerting designers to application bottlenecks and sources of system degradation. Second, as the operating system is extensible, when such bottlenecks are discovered, it is possible that the system can automatically download extensions to help relieve the bottlenecks.

Section 2 discusses related research and commercial projects in monitoring system behavior and optimizing techniques by a cooperation between the compiler and operating system. It also addresses adaptation techniques in real-time systems and operating systems. Section 3 describes the design of the monitoring module in VINO. This includes a survey of the performance metrics, classification of bottlenecks and how the application designer is notified about the system’s performance. Section 4 describes a number of user applications and how the system aids the designer in understanding where the application spends most of its time. Section 5 gives a simple example of the kernel’s ability to self-tune itself after monitoring itself and learning about the application’s access patterns. Section 6 gives a conclusion and future work that needs to be done to make the system more intelligent and robust.

2 Related Work

Self-monitoring in VINO is at the intersection of two areas of interest within the computer science research community. Section 2.1 discusses past work in monitoring application performance. Previous research partially focussed on optimizing executables, profiling machine-level instructions or developing specific dynamic modules such as a scheduler that may adapt its behavior depending on application requirements. Section 2.2 discusses extensible operating systems as VINO is an example of one.

2.1 Performance Analysis

An early system that monitors system resources is Multics [Sal70]. Originally developed to allow the sharing of services in a controlled manner, it uses a number of measurement techniques to understand system resource utilization. The metering capability records the time spent executing certain supervisor modules. For each module, the metering system records the number of times the module is invoked and the total execution time accumulated. The Multics designers regarded statistical system data gathering as a vital component because it reveals
performance problems that application designers or users may not detect.

Interest in the detection of system bottlenecks and system degradation for optimization purposes has produced a number of systems that continuously collect performance data on user and kernel activities. Morph [Mor96], for example, is a compilation environment that allows executable programs to be targeted to a specific machine. It defines the compilation such that the executable can be organized to monitor the system's configuration and accumulate knowledge about the application requirements. Such monitoring allows the use of feedback-based optimizations which can improve performance. Unlike the proposed VINO performance analysis monitoring based on an x86 platform to improve kernel performance, Morph provides a framework for compiler optimizations and is only available on Digital Alpha machines running Digital UNIX 4.0.

The Digital Continuous Profiling Infrastructure (DCPI) [Wei97], developed at Digital Equipment Corporation, continuously profiles program executables. It uses the profiles to identify and understand performance problems and to drive profile-based optimizations. The system builds up a database on disk of profile files for each executable image that has run. The profiling system periodically samples the hardware performance counters on the Alpha processors to produce sample counts for every instruction in every executable that is run. Instructions that take longer because of memory-access stalls, or branch mispredicts will accumulate more samples over time. Thus, the sample counts can be used to understand in minute detail where time is being spent. Even though the granularity of profiling in DCPI is quite fine compared to VINO, the goal of collecting statistical application profiles for performance analysis and automated optimization is similar. Its detailed instruction level execution profiles for user-land optimizations are based on DEC Alpha UNIX and NT systems.

The Synthetics operating system [Pu95] developed at the Oregon Graduate Institute aims to build toolkits based on the HP-UX operating system to support specialization of system code and demonstrate its use in real commercial applications. Unlike VINO's monitoring during execution time, specialization in Synthetics takes advantage of circumstances during load, run and compile times to make system optimizations. For example, the error-correcting and data-compression protocols necessary on a low-quality low-bandwidth wireless connection are unnecessary on a high-quality high-bandwidth fibre-optic network connection. Synthetics has been successful in specializing communications protocols, maximal bandwidth and data quality for a variety of applications such as a distributed multimedia system [Lin96].

The popularity of system monitoring is evident even in commercial products. IBM’s System Performance Monitor/2 (SPM/2) is an integrated package of powerful facilities allowing a user to monitor resources such as CPU, RAM, or disk on the local or remote IBM OS/2 2.x workstations and servers (http://www.provantage.com/). SPM/2 can display resource utilization, or record comprehensive data in log files, from which a user can produce detailed reports profiling system performance. In the realm of databases, DELPHI is a Real-time Performance Monitor for ORACLE, which provides comprehensive statistical and accounting data about ORACLE performance (http://www.blenheimintl.co.uk/delphi.htm). DELPHI combines information from the ORACLE dynamic performance tables, information from the IBM MVS operating system, and transparent instrumentation of the target database. DELPHI supplements
the ORACLE SQL*DBA utility, giving database administrators and system programmers concise, usable information about the performance of their database system. IBM’s DB2 Performance Monitor (DB2 PM) is a tool for analyzing, controlling, and tuning the performance of DB2 and DB2 applications (http://www.software.ibm.com/data/dbtools/). It includes a real-time online monitor, a history facility to view events that happened only recently, a wide variety of reports for in-depth analysis, and an explanation feature to analyze and optimize SQL statements.

Once a system is successful in monitoring its own performance, it is able to adapt its behavior and functionality to improve performance. VINO’s long term goal is to use the detailed profile information generated by the monitoring system to drive automatic optimizations and transformations of programs. With similar goals in real-time systems, Nett [Net98] develops a scheduler that tries to detect situations where an adaptation is needed by observing the behavior of an application and triggering adaptations when application defined constraints are violated by the current execution. Constraints include CPU utilization, waiting time for a task to complete and the latency for an I/O request. The real-time system is divided into four components: a distributed real-time application, a monitoring component that observes the behavior of the application, a resource manager that controls the execution of the application based on data collected by the monitor and an adaptation manager that uses the input from the monitor to implement long-term adaptation strategies. Adaptations may be performed on the real-time system itself or in the application level. Adaptive systems have also been implemented in the realms of distributed audio players [Lin96] through the use of a toolkit-based software feedback for client/server synchronization, dynamic control and system adaptiveness.

Adaptation requires systems to learn either via computational or neural-network means. Learning can be accomplished inductively and deductively [Win92]. Inductive learning is when the system boots up without any knowledge and learns everything from experience. It does not have any biases or assumptions. Deductive learning is rule based where the system starts off with some pre-specified observations and knowledge and then learns based on its assumptions. For example, the observation that a low cache hit ratio may be a symptom of a bad page eviction policy is a rule that is pre-defined in the system. Most systems are predominantly deductive in nature and in that way make decisions based on some prior understanding [Nak97]. Self-adaptation in an operating system is a combination of both types of learning but mostly deductive. This is true for two reasons. First, there are some inter-module relationships that are valid in any operating system and there is no reason for the computational learning system not to be aware of them. Second, even if one were to allow the inductive learning to take its course of action, a lot of time and effort may be required to achieve the same result had a set of deductive rules been pre-defined.

### 2.2 Extensible Systems

Adaptation may be accommodated easily in the realm of extensible operating systems because of their ability to alter kernel functionality. An example of such an operating system is VINO [Sma94], which allows user code to be loaded into the kernel. Thus, a user application, for example, may change kernel functionality by defining an updated version of a kernel module and dynamically loading it into the kernel. Extensions, called grafts in VINO, are compiled separately and loaded within the kernel at run-time. Monitoring is performed by defining statistical metrics
into the kernel code and periodically collecting system data from the operating system’s modules. (This is the topic of Section 3.)

There are a number of other extensible operating systems presently in use. The Exokernel [Eng95] allows the management of policy decisions to be made by the applications themselves. The kernel is only responsible for the protection of system resources such as physical pages and disk blocks. Using user space libraries, applications are able to manage critical resources. Through the libraries, the applications manage resources as if they were directly controlling the hardware components. Monitoring and adaptation may be accomplished by defining performance metrics in the libraries utilized by the applications. In this way, resource demands and access patterns may be collected to analyze system performance. Then, adaptation of kernel behavior may be achieved via’s Exokernel’s extensibility mechanism.

3 Monitoring in VINO

3.1 Introduction
Allowing the operating system to gather system performance statistics and analyze its own behavior automatically is invaluable. An on-line system can investigate a much larger spectrum of system statistics at a faster rate than any human user. The automatic monitoring mechanism within the VINO extensible operating system analyzes system-wide statistics in order to warn designers and application writers of performance bottlenecks, as well as to advise them of ways of alleviating such resource contentions.

Figure 1 shows the overall architecture of the monitoring module. An application running on VINO utilizes system resources. At the end of the execution, a user may request a report of the application's system performance.

![Figure 1: VINO Monitoring Architecture](image)

Figure 2 shows the design of the monitoring module. It consists of a monitoring thread that periodically queries sub-modules of the operating system such as the file system and virtual memory. A copy of the collected system statistics for each module is kept on disk for future use.
Section 3.2 describes the statistics gathered within VINO. Section 3.3 explains the methodology by which the collected data is analyzed and how different application bottlenecks such as I/O and CPU are identified. Using a sample user application execution, a generated report presents system statistics, access patterns and advice on improving performance. Section 4 describes the off-line monitoring module which aids the on-line module in detecting system trouble spots.

3.2 Data Collection

VINO monitoring collects, searches and analyzes a great quantity of information. This process of data analysis is divided into two major components:

a) Data Collection Phase - Every 2 seconds\(^1\), a clock interrupted measurement thread collects statistics and data from every major module in the system such as the file system, virtual memory and system call modules.

b) Representation Phase - Upon request, the system processes the data from the previous phase into a simple and concise report\(^2\) about system performance and resource utilization. This helps the application designer understand the reason(s) for performance degradation or system bottlenecks if any even exist. The report also includes suggestions to improve overall system performance.

The monitoring module periodically gathers performance metrics in three different categories. First, each thread has its own set of performance metrics that allows it to isolate the effects of its own behavior from other threads. Second, running applications may have a number of child processes or threads executing concurrently. Thus, a collective set of performance metrics

---

1. An interval of 2 seconds produces an overhead of less than 1%. Users, thus, should not perceive a significant slowdown in performance. This is described further in the overhead section of 4.3.
2. Sample user reports are given later in this section.
allows each application to analyze its own resource demands. This is especially important when having two concurrent applications that stress the operating system in very different manners. For example, a CPU intensive simulation exhibits different system behavior than an I/O bound database. Thus, separating the gathered performance statistics on a per-application basis allows each executing application to understand its own resource needs. Third, system wide statistics allow the understanding of the overall state of the computer system irrespective of the different resource demands of each running application.

Table 1 shows a partial listing of the present VINO metrics that are collected periodically. Specifically, for the purposes of this research, statistics only reflect the Virtual Memory, File System and System Call modules.

<table>
<thead>
<tr>
<th>Table 1: VINO Sensor Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Memory Allocations</td>
</tr>
<tr>
<td>Virtual Memory Allocation Time</td>
</tr>
<tr>
<td>Virtual Memory Deallocation</td>
</tr>
<tr>
<td>Virtual Memory Deallocation time</td>
</tr>
<tr>
<td>Virtual Memory Pages Read</td>
</tr>
<tr>
<td>Virtual Memory Pages Freed</td>
</tr>
<tr>
<td>Virtual Memory Paging Started</td>
</tr>
<tr>
<td>Virtual Memory Pages Active</td>
</tr>
<tr>
<td>Virtual Memory Pages Inactive</td>
</tr>
<tr>
<td>Number of System Call Invocations for each type</td>
</tr>
</tbody>
</table>

Every monitoring interval, the monitoring module collects statistical information to aid in pinpointing fluctuations and bottlenecks within the system. The gathered performance data is kept in memory and periodically written to disk. The monitoring subsystem keeps a directory on disk of the statistics for each module. For example, a monitoring file for the Virtual Memory (VM) module indexed by the interval timestamp and process (pid) lists the performance metrics of the VM module. The monitoring thread places a new entry containing the timestamp of the interval followed by all the VM related performance metrics in the appropriate file.

From the basic sensored metrics, other useful statistical measures arise. These include the mean time for memory allocation, standard deviation of the time to allocate memory, the mean time for a memory deallocation, the standard deviation for a memory deallocation, and the cache miss ratio of pages not found in physical memory. VINO, presently, splits the total execution time used by an application into user and system time by a novel approach. Every time a context
switch occurs the return program counter (PC) address is used to differentiate between code that was executing in user versus system address space. If the PC were in user space then the previous clock interval contributes to the total user-space time execution; otherwise the interval allots to the kernel time utilization.

Regular samples of sensed performance metrics help to detect trends or patterns in system behavior. For example, the monitoring module may sample a downward trend in the number of page faults over a period of time. Over the course of an execution, this reflects that an application requests' for data are satisfied from memory and do not incur as much disk I/O. This type of analysis is more insightful than a single value for the cache hit ratio or the total number of page faults because it spans the total execution time of an application. Thus, it extracts the demand for I/O as early periods where a large number of pages are requested from disk and later periods with few or no page faults. Trends may be categorized into three different groups: upward, downward and none. Trend extraction needs to be robust enough to handle statistical noise which is common in any real system. For example, a downward trend of sampled system-wide page faults may have some upward bursts but have an overall downward pattern (i.e. 10,9,8,9,7,5,3,3,1). There are many ways [Bow79, Lar86] to extract trends in time-series analysis such as utilizing linear regression in combination with a T-test but this work uses correlation because of its simplicity, robustness and not requiring the use of any probability distribution tables that are common for most statistical analysis.

### 3.3 Monitoring Analysis

There are two goals in the data analysis phase. First, the monitoring module identifies applications’ access patterns and habits. For example, an I/O dependent read-only sequential access pattern has different performance optimizations than a CPU intensive read-write access pattern. Second, once high latency operating system modules are pinpointed, a user report suggests specific enhancements and optimizations.

#### 3.3.1 Classification Module

Bottlenecks and system degradation, if any, are a reflection of the demands made by an application on the operating system and hardware. Classifying such resource demands requires an understanding of access patterns. One way is to provide the operating system with access pattern information via user supplied hints or qualitative access pattern descriptions [Cao94, Pu95]. Unfortunately, this approach requires ongoing programmer effort to reconcile the hints with code evolution. Inaccurate hints can cause performance problems if the system selects policies that are suitable for the hints, but unsuitable for the actual access pattern. Another approach, taken in VINO, is to automatically classify access patterns during program execution. This approach requires no programmer intervention and is robust enough to handle dynamically changing or data-dependent access patterns. A classifier module (CM) observes the application level access stream and generates qualitative descriptions. These descriptions, combined with qualitative statistics are used to detect performance degradation and suggest optimizations. This approach is similar to the one taken by Mowry [Mow96] in improving the performance of large, memory-intensive, scientific applications by enhancing paging in the virtual memory. [Mow96] develops an automatic mechanism for communication between the operating system and the
The compiler analyzes future access patterns to predict future page faults and the operating system prefetches the needed pages in order to minimize I/O time.

The goal of the analysis phase is to understand the system demands of the application. First, we identify whether the application in question is CPU-, Memory- or I/O intensive applications. Other useful classifications include whether the access pattern is read-only, write-only or read-write, whether the requests are predominantly sequential in nature or not, whether page evictions have started, the rate of virtual memory paging and whether there is an upward or downward trend in the time spent in I/O or the number of page faults.

When a user prompts VINO to produce a report of the applications's execution, VINO identifies the interval of time that the application was executing from the statistics files. The report, then, bases its results on statistical metrics within the span of execution.

Since we must first identify whether an application is I/O-, memory-, or CPU bound, we first examine the time distribution between kernel execution and user-level execution. The more time is spent in system or kernel mode then an application is I/O or Memory intensive; otherwise, it is CPU intensive.

Given that an application is CPU intensive, paging may have occurred. If there are page faults and virtual memory activity combined with little or no file system requests, the application may be memory bound. This is a situation where the paging is not caused by file system requests but by bringing pieces of the address space. In this situation, a better page evictor may help improve performance. If there is no paging or virtual memory activity such as in a simulation or scientific computation, then the application is CPU-bound. In this state, there is not much that can be done other than utilizing a faster processor or computer. A possible alternative is compiler or executable optimizations as in Morph or DCPI.

On the other hand, an application that is memory or I/O intensive has many variables and system metrics for consideration. If a large percentage of the read and/or write requests are sequential, the application most likely reads from or writes to a file with a great amount of consistency and predictability. Therefore, a more aggressive read-ahead policy within the operating system may be advantageous. A non-sequential access pattern with a downward trend in page faults and I/O time may reflect an application with a lot of pages from disk in the beginning but as the used pages become available in memory, the need to page fault and incur disk I/O time diminishes. In such a situation, a better prefetch mechanism, if possible, to predict future requests may be advantageous. If there is page eviction activity combined with a non-sequential access pattern, a better page evictor policy may be advantageous. Lastly, an access pattern that may only be categorized as non-sequential leads to an uncertain method of improvement. This is a situation where the user may not be aided in any specific way.

To determine if an access pattern is read-only, the CM inspects the number of read and write requests made to the file system. If any of the requests is a write, the pattern is not classified as read-only. Write-only classification is analogous to read-only classification. An access pattern is read-write if there exists the possibility of both reads and writes. It is true that this rather simple classification is unforgiving because a single write to an otherwise read-only file is classified as
An access pattern is defined sequential by computing the probability that two consecutive read requests are made to two sequential blocks of a file. The file system keeps a count of the total number of read requests made as well as the count of the number that are sequential. If the probability that sequential read requests is higher than some threshold, the access pattern is classified as sequential. Presently, there are different thresholds for sequential classification, 60%, 80% and 90%. This allows the classifier to have a richer analysis of sequentially. For example, an analysis of an application with a sequential access ratio of 98% warrants a different type of advice than an application with one that is 61%. Sequential access ratios of less than 60% deem the application to have a non-sequential access pattern. Ideally, the threshold values are updated off-line from a Markov Model simulator trained from application access patterns (Discussed further in Section 3.4).

### 3.3.2 A Sample User Application and Report Output

As a concrete example of an application, consider a user who executes a monte carlo simulation of a sinusoidal integration on a 133Mhz pentium with 32MB of RAM running the VINO 0.40 kernel. Once the application completes, the user requests a report via the classify() command in VINO. The resulting report is:

- Total Execution Time = 40 seconds
- Total Number of Read Requests = 7
- Total Number of Read Bytes = 57342
- Total Number of Sequential Read Requests = 2
- Total Number of Write Requests = 13
- Total Number of Write Bytes = 5412
- Total Number of Sequential Write Requests = 5
- Page Evictions Started = False
- Total Number of Pages Freed = 0
- Total Number of VM Allocations = 0
- Total Number of VM Deallocations = 0
- Percentage of time spent in Kernel = 0.5
- Percentage of time spent in User-land = 99.5
- Trend In Page Faults = None
- Trend in I/O = None

Analysis:
- It seems the execution is vastly in user-space (99.5% of execution time).
- There were no page evictions, so paging is not an issue.
- There are no trends in I/O or Page Faults.
- The accesses do not seem to be sequential in nature.
- You are CPU bound and a faster processor will help.

The report attempts to give a summary of the resource requirements of the application. Then, it gives some insight into the access patterns, trends and whether the application is I/O, Memory or
CPU bound. More examples are given in Chapter 4.

3.4 Off-line Monitoring Module
Threshold values for triggering system warnings have default values that are defined by the kernel developer. However, allowing the operating system to dynamically classify access patterns and thresholds is vital. This may be accomplished via an off-line feedback neural network or a Hidden Markov Model system. The network will be made up of a number of nodes which are connected together via communication channels that carry data. Each channel will have a weight associated with it and the nodes will produce outputs that are computed from a weighted sum of the inputs. Given a detailed system resource utilization, the off-line module will be capable of obtaining new thresholds for a variety of components such as the threshold for the probability of sequential reads.

Currently, the collection of statistical data throughout the system is performed at regular pre-defined intervals. However, it is possible that some modules require a finer grain of monitoring. This conclusion may be reached by an off-line module which will use statistical techniques for variance analysis to decide if the collection interval ought to be decreased [Sel97]. It also may decide that the interval can be increased without loss of information.

4 VINO as a Performance Bottleneck Detector
In section 3, the monte carlo simulation was an example of a user application. The report showed that it was CPU bound and there was not much that could be done by the operating system to improve performance. This chapter gives examples where the operating system aids in improving system performance. First, utilizing applications such as matrix multiplication, copying a file and randomly reading from a large file, the VINO monitoring module pinpoints various bottlenecks depending on system demands. Then, the application designer utilizes the advice of the report to improve performance. Second, the Lempel-Ziv Compression scheme is an example of a common application in networking and systems. VINO analyzes its resource requirements and outputs its findings. Finally, section 4.3 describes the overhead of monitoring in VINO.

4.1 More Sample User Applications
Three different applications stress VINO in different ways: copying a large file, multiplying two matrices and accessing a file randomly. With each one, a different report results with different analysis and advice. All these applications ran on a 100Mhz pentium computer with 32MB of RAM running the VINO 0.40 kernel.

First, a user copies a 6.5 megabyte file (i.e "cp file another-file"). The resulting report is:

Total Execution Time = 15 seconds
Total Number of Read Requests = 106
Total Number of Sequential Read Requests = 97

1. This section is not yet implemented
Total Number of Read/Write Bytes (I/O)= 6590092
Total Number of Bytes Requested by User = 6562816
Total Number of Write Requests = 102
Total Number of Write Bytes = 6590092
Total Number of Sequential Write Requests = 93
Page Evictions Started = False
Total Number of Pages Freed = 0
Total Number of VM Allocations = 0
Total Number of VM Deallocations = 0
Percentage of time spent in Kernel = 100.0
Percentage of time spent in User-land = 0.0
Trend In Page Faults = None
Trend in I/O = None

Analysis:
- It seems the execution is vastly in kernel-space (100.0% of execution time).
- There were no page evictions, so paging is not an issue.
- There are no trends in I/O or Page Faults.
- The accesses seem to be sequential in nature (95.0%)
- You are I/O bound and a more aggressive read-ahead policy will help.

Taking the report’s suggestion, a user dynamically loads a graft within VINO. This graft represents the open_file_o::compute_ra method. After noticing a sequential pattern, the read ahead method, by default, prefetches DEFAULT_RA_AMT, 32K, of the file in anticipation of another request. \( off \) is the offset in the file and \( count \) is the amount of bytes requested.

    if (off == last_off) {
        // This read was sequential. Do some read-ahead
        ra_end = round_page (off + count + DEFAULT_RA_AMT)
        ...
    }

The open_file_o::compute_ra graft changes the prefetch amount to:

    if (off == last_off) {
        // This read was sequential. Do some read-ahead
        ra_end = round_page (off + count + 2*DEFAULT_RA_AMT)
        ...
    }

After the graft is downloaded in VINO, the copy only required 13.8 seconds. This is a 1.2 second or 8% improvement. The performance improvement is important for two reasons. First, the monitoring module successfully pinpointed a system bottleneck and recommended a course of action. Second, VINO’s extensibility capability allowed an application designer to dynamically load an alternative read-ahead mechanism and improve the application’s execution time.

A second benchmark is an application that reads random locations from a 21.2 megabyte file.
The application runs for 5 minutes and then quits. The main part of the applications is:

```c
for (loops = 0; ; loops++) {
    for (i = 0; i < 4096; i++) {
        if (i % check_interval == 0) {
            gettimeofday(&time, (struct timezone *) 0);
            time_elapsed =
                (time.tv_sec + time.tv_usec/1e6 - start_time);
            if (time_elapsed > end_time) exit(0);
        }

        where = (random() % 4096) * 4096;
        lseek(fd, where, 0);
        p = buffer + ((i % 1024) * 4096);
        read(fd, p, 4096);
    }
}
```

The resulting user report is:

- Total Execution Time = 300 seconds
- Total Number of Page Faults = 4108
- Total Number of Read Requests = 75744
- Total Number of Sequential Read/Write Requests = 29
- Total Number of Read Bytes (I/O) = 16901120
- Total Number of Bytes Requested by User= 321758792
- Total Number of Write Requests = 0
- Total Number of Write Bytes = 0
- Total Number of Sequential Write Requests = 0
- Page Evictions Started = False
- Total Number of Pages Freed = 0
- Total Number of VM Allocations = 0
- Total Number of VM Deallocations = 0
- Percentage of time spent in Kernel = 98.1
- Percentage of time spent in User-land = 1.9
- Trend In Page Faults = Downward
- Trend in I/O = Downward

Analysis:
- It seems the execution is vastly in kernel-space (98.1% of execution time).
- There were no page evictions, so paging is not an issue.
- There are downward correlation trends in I/O (-0.94) and Page Faults (-0.96).
- The accesses do not seem to be sequential in nature.
- You are I/O bound and accesses seem to be resolved more often in memory as the application runs longer.
- Not much help can be advised.

An application with random access patterns is difficult to improve. While the VINO monitoring module does not yet extrapolate complicated access patterns such as non-sequential equally
spaced memory accesses, an application designer could better understand the behavior of his or her application. Then, a better read-ahead or prefetch mechanism may foresee future accesses to disk or memory and, thus, improve system performance.

The third benchmark is a matrix multiplication application that multiplies two matrices of long integers each with a dimension, DIM, of 1024x1024 and stores the result in a third matrix. In order to induce memory paging, 21.6MB of the available 32MB are reserved for malloc() by using the global MALLOC_RESERVE variable in VINO. The core of the matrix multiplication application is the following:

```c
for (i = 0; i < Dim; i++)
    for (j = 0; j < Dim; j++)
        for (k = 0; k < Dim; k++)
            C[i][j] += A[i][k] * B[j][k];
```

The user report is:

- Total Execution Time = 1884 seconds
- Total Number of Read Requests = 13
- Total Number of Sequential Read Requests = 2
- Total Number of Read/Write Bytes (I/O) = 12488400
- Total Number of Bytes Requested by User = 829440
- Total Number of Write Requests = 2026
- Total Number of Write Bytes = 12010
- Total Number of Sequential Write Requests = 1824
- Page Evictions Started = True
- Total Number of Pages Freed = 8274
- Total Number of VM Allocations = 0
- Total Number of VM Deallocations = 0
- Percentage of time spent in Kernel = 4.1
- Percentage of time spent in User-land = 95.9
- Trend In Page Faults = None
- Trend in I/O = None

Analysis:
- It seems the execution is vastly in user-space (95.9% of execution time).
- There were page evictions.
- There are no trends in I/O or Page Faults.
- The accesses do not seem to be sequential in nature.
- You are Memory bound.
- Low filesystem activity but mostly write-only
- Consistent Virtual Memory activity of pages freed, activated and de-activated.
- Virtual Memory pattern with little or no file system demands.
- A better page eviction policy may aid in improving memory accesses.

The performance of matrix multiplication is not optimal. Each matrix is stored as a 2-dimensional array. Since the matrices are 1024x1024 and each element is a long integer (8
bytes), each row occupies 1024 * 8 = 8192 bytes or two complete pages. But when the matrices are multiplied, the rows of matrix A are multiplied with the columns of matrix B. These columns span different pages so as the program iterates down column, new pages must be constantly loaded into memory. Also, since the results of the multiplication are written to the contents of the third array, a lot of write faults result if its pages are chosen for eviction. As an application designer may notice, a simple alternate implementation of the matrix multiplication is to multiply one matrix with the transpose of the other. Since the elements of a row all fit on one page, there should be better locality in data references and, thus, better performance. The matrix multiplication is altered to the following:

```c
for (i = 0; i< Dim; i++)
    for (j = 0; j < Dim; j++)
        for (k = 0; k < Dim; k++)
            C[i][j] += A[i][k] * B[k][j];
```

The user report is:

- Total Execution Time = 645 seconds
- Total Number of Read Requests = 24
- Total Number of Sequential Read Requests = 6
- Total Number of Read/Write Bytes (I/O) = 6875900
- Total Number of Bytes Requested by User= 870400
- Total Number of Write Requests = 678
- Total Number of Write Bytes = 12010
- Total Number of Sequential Write Requests = 546
- Page Evictions Started = True
- Total Number of Pages Freed = 11059
- Total Number of VM Allocations = 0
- Total Number of VM Deallocations = 0
- Percentage of time spent in Kernel = 14.1
- Percentage of time spent in User-land = 85.9
- Trend In Page Faults = None
- Trend in I/O = None

Analysis:
- It seems the execution is vastly in user-space (85.9% of execution time).
- There were page evictions.
- There are no trends in I/O or Page Faults.
- The accesses do not seem to be sequential in nature.
- You are Memory bound.
- Low filesystem activity but mostly write-only
- Consistent Virtual Memory activity of pages freed, activated and de-activated.
- Virtual Memory pattern with little or no file system demands.
- A better page eviction policy may aid in improving memory accesses.

The new report states that memory is still a problem because of the large number of memory accesses. As seen in Table 2, comparison of the two matrix multiplication implementations
shows substantial performance gains. The total execution time has dropped by approximately 65% and the amount of bytes read from disk dropped significantly by 45%.

**Table 2: Performance Improvement in Matrix Multiplication**

<table>
<thead>
<tr>
<th>Matrix Multiplication</th>
<th>Total Time (secs)</th>
<th>Time in Kernel (%)</th>
<th>Time in User-Space (%)</th>
<th>Bytes Read from Disk (I/O)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old Implementation</td>
<td>1885</td>
<td>4.1</td>
<td>95.9</td>
<td>12488400</td>
</tr>
<tr>
<td>New Implementation</td>
<td>645</td>
<td>14.2</td>
<td>85.8</td>
<td>6875900</td>
</tr>
</tbody>
</table>

**4.2 The Lempel-Ziv Compression**

A common system utility that is popular in networking is compression. Specifically, the Lempel-Ziv compression replaces strings of characters with single codes. It does not do any analysis of the incoming text. Instead, it just adds every new string of characters it sees to a table of strings. Compression occurs when a single code is output instead of a string of characters.

The code that the LZW algorithm outputs can be of any arbitrary length, but it must have more bits in it than a single character. The first 256 codes (when using eight bit characters) are by default assigned to the standard character set. The remaining codes are assigned to strings as the algorithm proceeds. The sample program runs with 12 bit codes. This means codes 0-255 refer to individual bytes, while codes 256-4095 refer to substrings.

The Lempel-Ziv compression algorithm in its simplest form looks like:

**Routine LZW_COMPRESS**

```plaintext
STRING = get input character
WHILE there are still input characters DO
    CHARACTER = get input character
    IF STRING+CHARACTER is in the string table then
        STRING = STRING+character
    ELSE
        output the code for STRING
        add STRING+CHARACTER to the string table
        STRING = CHARACTER
    END of IF
END of WHILE
output the code for STRING
```

A quick examination of the algorithm shows that the algorithm always tries to output codes for strings that are already known. And each time a new code is output, a new string is added to the string table.
The companion algorithm for compression is the decompression algorithm. It needs to take the stream of codes output from the compression algorithm, and use them to exactly recreate the input stream. One reason for the efficiency of the LZW algorithm is that it does not need to pass the string table to the decompression code. The table can be built exactly as it was during compression, using the input stream as data. This is possible because the compression algorithm always outputs the STRING and CHARACTER components of a code before it uses it in the output stream. This means that the compressed data is not burdened with carrying a large string translation table.

```
Routine LZW_DECOMPRESS

Read OLD_CODE
output OLD_CODE
WHILE there are still input characters DO
    Read NEW_CODE
    STRING = get translation of NEW_CODE
    output STRING
    CHARACTER = first character in STRING
    add OLD_CODE + CHARACTER to the translation table
    OLD_CODE = NEW_CODE
END of WHILE
```

A simple application that compresses and then decompresses a user-specified file (13.9 MB) executes on VINO.

The user report is:

Total Execution Time = 268 seconds
Total Number of Read Requests = 3834
Total Number of Sequential Read Requests = 3822
Total Number of Read/Write Bytes (I/O) = 64500126
Total Number of Bytes Requested by User= 27624256
Total Number of Write Requests = 4012
Total Number of Write Bytes = 34083012
Total Number of Sequential Write Requests = 3975
Page Evictions Started = False
Total Number of Pages Freed = 0
Total Number of VM Allocations = 0
Total Number of VM Deallocations = 0
Percentage of time spent in Kernel = 83.4
Percentage of time spent in User-land = 16.6
Trend In Page Faults = Downward
Trend in I/O = None

Analysis:
    - It seems the execution is vastly in kernel-space (83.4% of execution time).
    - There were no page evictions.
- There is a downward correlation trend in page faults (-0.64).
- The accesses seem to be sequential in nature (99.0%).
- You are I/O bound.
- Read-Write filesystem activity
- You are I/O bound and a more aggressive read-ahead policy may help.

An application designer may easily implement a more aggressive read-ahead policy as utilized int the matrix multiplication example. An alternatice is a a slight change in the implementation. In the original code, I have used code sizes of 12 bits. In a 12 bit code program, there are potentially 4096 strings in the string table. Each and every time a new character is read in, the string table has to be searched for a match. If a match is not found, then a new string has to be added to the table. With large files to compress, a larger string table is advantageous. With 14 or 15 bit codes, compression may improve. Table 3 shows a performance comparison of the original implementation, grafted read-ahead VINO and the altered compression algorithm.

<table>
<thead>
<tr>
<th>LZW Implementation</th>
<th>Execution Time (secs)</th>
<th>Size of compressed file (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>268</td>
<td>3.06</td>
</tr>
<tr>
<td>Grafted Read-Ahead</td>
<td>262</td>
<td>3.06</td>
</tr>
<tr>
<td>15-bit Code size</td>
<td>264</td>
<td>2.5</td>
</tr>
</tbody>
</table>

It can be seen from Table 3 that there is a trade-off between time performance and the size of the compressed file. A slightly faster alternative to the original implementation may not render an optimally efficient utilization of disk. But either the graft or larger code size improves overall system time utilization.

4.3 Overhead
In order to determine the overhead of monitoring, VINO reads a 6 megabyte file sequentially with and without monitoring. Using a 100Mhz Pentium system with 32MB of physical RAM, 62 intervals of system performance statistics are collected. The latency incurred from collecting the 20 intervals of statistics is 400 milli-seconds resulting in an overhead of 20 milliseconds per interval. This includes the extra overhead for collecting the monitoring data, updating the appropriate files and 2 context switches. One context switch to initiate the monitoring agent and the other to return to the executing application. The 1% overhead achieved is low enough that application performance won't be affected greatly. Smaller overhead can easily be achieved if the monitoring agent is triggered within a clock interrupt to avoid the 2 unnecessary context switches.

5 VINO as an Adaptive Self-Tuning System
Quantitative and qualitative data make possible dynamic selection and configuration of kernel policies based on access pattern characteristics. By choosing policies to match each
application’s needs, the operating system can provide higher performance than by enforcing a single system-wide policy.

VINO continuously monitors and classifies the behavior of many modules such as the file system. This is crucial for adaptive on-line functionality. For example, when an access pattern is classified as sequential and read-only, a new kernel policy might prefetch more aggressively. Likewise, when the access pattern is regularly strided, a new policy might prefetch anticipated blocks on the stride size and adjust the cache size to increase the hit ratio.

To illustrate how dynamic adaptation can be a performance advantage, a simple benchmark that reads the first half of a 40 MB file sequentially and the second half randomly runs on VINO. For both access patterns, the request size is 4096 bytes which is equivalent to VINO’s page size.

The number of read requests and the percentage which are sequential are the important performance metrics. If the percentage of sequential reads is greater than 75%, a more aggressive read-ahead policy is used. Normally, read-ahead prefetches the next page. A simple more aggressive read-ahead policy can obtain the next 4 pages. Of course, excessive read-ahead could be just as bad. When the CM analyzes the performance metrics and access patterns, a change in the read-ahead kernel policy using VINO’s grafting capability may be performed. An updated read-ahead policy is dynamically loaded into the kernel.

Table 4 shows the time in seconds for the two different components of the user access pattern.

<table>
<thead>
<tr>
<th>Access Pattern</th>
<th>Default</th>
<th>Adaptive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential Reads</td>
<td>126</td>
<td>95</td>
</tr>
<tr>
<td>Random Reads</td>
<td>248</td>
<td>249</td>
</tr>
<tr>
<td>Total Time</td>
<td>374</td>
<td>345</td>
</tr>
</tbody>
</table>

As can be seen, a benefit is obtained by realizing that the user access pattern is sequential and thus invoking a new read-ahead policy. The time for the adaptive components include the times required to load the new policy into the kernel. That is why the random reads using the adaptive technique is slightly longer than the default. The first few intervals of statistics are still assuming an aggressive read-ahead policy and therefore a lot of time is wasted bringing sequential pages into memory without much gain to the overall performance of the system. It takes a bit of time to realize that this is an incorrect assumption and unload the graft. Unloading the graft takes approximately 90ms.

---

1. The choice of 4 pages is somewhat chosen arbitrarily to compromise between reading ahead too many and too little pages.
Even though the example is relatively simple, the results of the benchmark demonstrate that automatic classification and adaptive systems have the potential to provide better performance over a wider range of access patterns.

6 Conclusion

Through the use of available products, it was shown that the monitoring mechanism is capable of providing useful and valid performance analysis via its collected statistics. The classification methodology allows simple self-tuning and adaptive functionality using VINO’s extensibility mechanism.

VINO’s monitoring module is a piece of the larger Self-Monitoring and Self-Adaptation project [Sel97], which is exploring techniques to substantially improve the performance of large customer programs running on Intel based systems. The existing profiling tools provide detailed information about the behavior of a system, allowing developers and users to determine where time is being spent and to use that insight to improve performance. A secondary goal is to use the detailed profile information generated by the monitoring system to drive automatic optimizations and transformations of programs.

7 Bibliography


