Windows Performance Monitoring and Data Reduction Using WatchTower

Michael W. Knop [1,2] Jennifer M. Schopf [1,2] Peter A. Dinda [1]

knop@cs.northwestern.edu jms@mcs.anl.gov pdinda@cs.northwestern.edu

ABSTRACT

We describe and evaluate WatchTower, a set of library routines that simplifies the collection of performance data for the monitoring of Windows NT/2000. WatchTower has an overhead similar to that of existing tools but is more easily embedded into other applications. More important, we show how data reduction techniques can be used to diminish the volume of performance data gathered to only that which is useful; while still capturing the overall behavior of the computer.

Keywords

Performance monitoring, data reduction, Windows NT/2000

1. INTRODUCTION

There is a growing need for systems that can automatically detect performance bottlenecks or more serious problems and then dynamically adapt their execution to fix themselves. The first step toward this kind of fault-tolerant and adaptive computing is performance monitoring. While the many UNIX variants have been studied extensively in this respect, Windows research in this area remains in its infancy. For the Windows operating systems to become viable options in autonomic computing, we must begin with the subject of performance monitoring.

To this end we have developed *WatchTower*, a set of library routines that simplifies the collection of Windows performance data. WatchTower provides easy access to the performance counters that the Windows NT/2000 operating systems manage and is easily embedded into monitoring software. Our application based on WatchTower has comparable overhead to Microsoft's *Perfmon* [11] running in background mode.

Arguably, the amount of Windows NT/2000 performance data that is available can be overwhelming. Blindly monitoring all the performance counters Windows makes accessible could result in over 170 Mb of data per day per machine (monitoring at a rate of 1 Hz). Our experiments indicate that only a fraction of the entire collection of counters needs to be examined, while still retaining most of the useful performance information about the machine being monitored. To show this, we have employed a data reduction technique to create a subset of counters that preserves the machine's behavior. The reduction in the volume of information benefits both the archiving of performance data and its real-time analysis. As the number of machines being monitored increases, the benefits of data reduction become more apparent.

The performance data gathered by WatchTower allows us to study user and machine activity over time across any number of computers running Windows NT/2000. We envision a variety of scenarios for using this data:

Adaptive Computing: Traces gathered can serve as the starting point for real-time adaptive components for building resilient, distributed, and parallel applications [14].

Fault Tolerance: The traces can be used in the diagnosis and healing process for autonomic computing solutions where the complexity of the system is too great for user troubleshooting [7].

Intrusion Detection: What can be classified as unusual or insecure behavior can be detected by using our data analysis [8].

Scheduling: These traces are an accessible form of variability information for cluster scheduling techniques [18], real-time scheduling [2], adaptation [3], or resource management [12].

Platform Profiling: We can compare logs of UNIX system usage with those created under Windows to determine whether the same theories of UNIX user/application habits apply to Windows. Likewise, these logs can be used for building machine signatures [16] to characterize machine performance against standard benchmarks.

The remainder of this paper is structured as follows: In the next section we describe performance monitoring in Windows and WatchTower. Section 3 explains the data reduction technique we used on the logs gathered by WatchTower, and Section 4 discusses the results of that technique. Section 5 reviews related work. Finally, in Section 6 we draw conclusions and describe future work.

---

[1]Northwestern University, Department of Computer Science, 1890 Maple Avenue, Evanston, IL 60201

[2]Argonne National Laboratory, Mathematics and Computer Science Division, 9700 South Cass Avenue, Argonne, IL 60439

Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN), New York City, June 23, 2002.

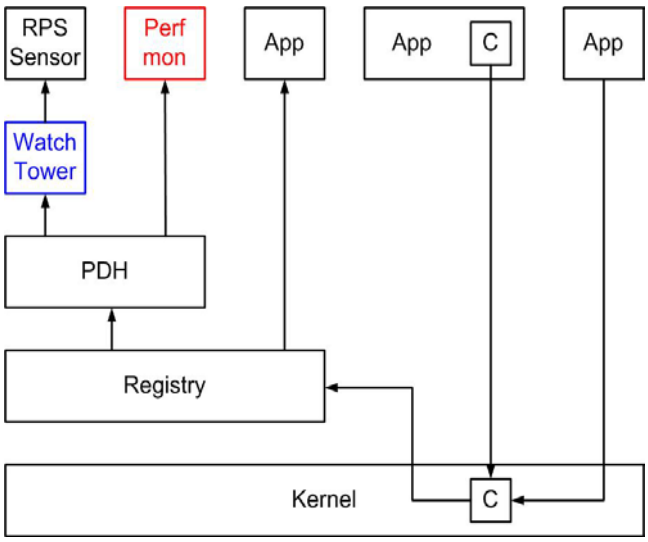

Figure 1: Windows performance monitoring structure.

2. WATCHTOWER

WatchTower is a set of C++ classes that greatly simplifies the collection of performance data for the Windows NT/2000 operating systems. The routines allow monitoring to be done unobtrusively (as a service, no user interaction needed) or at the command line and have several options for logging (to console, file, or streaming). We have created applications based on the WatchTower library, including a service that logs to a local disk and a command-line implementation for RPS [4]. The WatchTower library code had many influences [10][19][20] and has been used in functioning applications for the past year.

2.1 Performance Monitoring in Windows

Windows NT and 2000 contain a measurement infrastructure that is made visible to applications by means of performance counters that together reflect the current state of the system. The exact number of counters varies depending on the system's configuration; for a computer with a complex arrangement, this number is roughly one thousand counters. Counters are arranged in a simple two-level hierarchy. The first level of the hierarchy is the physical or system *performance object*, while the second is the *performance counter*, which is an attribute of the object. Examples of objects include Processor and Memory, example counters of those objects include `%User Time` and `Available Bytes`, respectively. Some objects, such as `Processor` or `PhysicalDisk`, may have multiple instances, like a dual-processor node or a computer with many hard drives.

Performance counters are stored and updated in the Windows registry and can be accessed by a registry API. Working directly with the registry is complex, however, so Microsoft provides a more abstract API called the Performance Data Helper (PDH) [10]. This higher-level API handles the accessing of the counters in the registry and converting their raw values into numbers with appropriate units and scale. PDH is the basis for both Perfmon (discussed in the next section) and WatchTower. Figure 1 shows an overview of how these parts interact.

The first layer in the figure is the operating system kernel. The second layer is a system tool layer, meaning the registry is separate from the kernel yet tightly coupled to it. The platform library layer (PDH) is next, followed by the application library (WatchTower), and finally the application layer. The main source of performance counters is the kernel (gathered from the OS and running processes), but counters can also be specially built into applications. The kernel gathers performance data from its various sources and updates the registry periodically, which in turn makes the counters available to applications using the registry API and Performance Data Helper API.

We emphasize that Windows performance counters should not be confused with hardware counters, such as those found in most major microprocessors. Our focus is on the performance data provided by the Windows NT/2000 operating systems, not the hardware. Thus our approach is most similar to the HPVM performance monitor [17], rather than PAPI [1].

2.2 PDH and Perfmon

Using PDH in an application is easier than using the registry API, as mentioned above. But there is still a good deal of overhead in setting up a trace in PDH. One must get the counters into a query, collect the data in a periodic fashion, correlate the data, and so forth, all while checking for numerous possible errors. Moreover, the only output function given writes to a file on disk without buffering. WatchTower, on the other hand, has been written to automate the common groundwork to start a trace. The only input needed is a list of counters (which it will verify exist), a measurement rate, and where to send the output. Simple output functions are provided (for example, streaming to console), and hooks are available to add application-specific output functions. Thus, programmers do not need to concern themselves with the dense PDH API; instead they need work only within the straightforward WatchTower interface.

The most identifiable PDH-based application is Microsoft's Performance Monitor, also referred to as Perfmon [11]. Perfmon can operate entirely in the background, hidden from the interactive user. It is configured through an HTML-like script. However, Perfmon has several deficiencies that limit its long-term logging capability and usefulness, namely granularity, over-writing of files, and adaptability. These significantly affect its ability to provide adequate logging in a high performance cluster environment.

The finest measurement granularity Perfmon supports is one second, which is inadequate for many uses of logging data in a high performance system. In contrast, WatchTower supports a granularity of 10 ms (limited by the Windows timer granularity) and a peak rate of approximately 16 Hz (depending on how many counters are being monitored). Perfmon also overwrites the last log file after (even graceful) reboots. This drawback makes Perfmon undesirable for collecting long-term traces of machine behavior. WatchTower avoids this problem (when writing to a file) by starting a new log file every hour and after system startup. Finally, Perfmon is difficult to incorporate into other systems or to extend with new functionality (for example, streaming output). WatchTower is a simple set of C++ routines and thus can be embedded into other programs trivially.

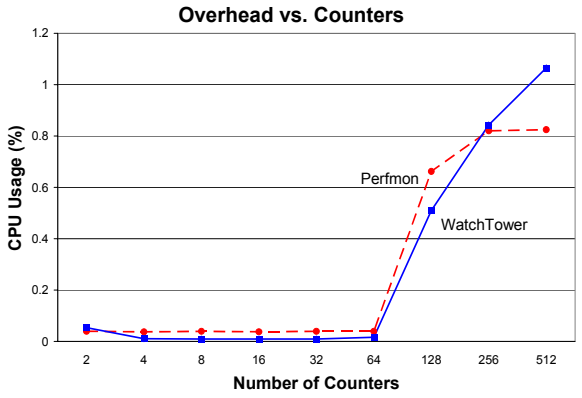

Figure 2: Overhead vs. counters at 1 Hz for Perfmon and WatchTower.

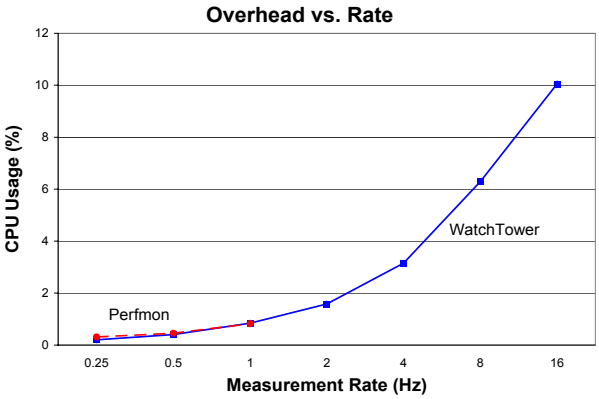

Figure 3: Overhead vs. rate with 256 counters for Perfmon and WatchTower. Note that the Perfmon line does not extend beyond 1 Hz.

2.3 Overhead

WatchTower's overhead is similar to that of Perfmon. Figure 2 shows the overhead (as a percentage of CPU time in use while the machine was idle) of Perfmon and a WatchTower application as a function of the number of counters, while Figure 3 shows the overhead as a function of the measurement rate. In this example, both tools are logging to disk in exactly the same way (a function of PDH). One can see that WatchTower taxes the CPU in a similar fashion as Perfmon. While Perfmon cannot function at rates greater than 1 Hz, however, WatchTower can monitor at a peak rate of 16 Hz (on a dual 500 MHz Pentium III machine) when charged with 256 counters. Further, the memory footprint of WatchTower is 15% smaller than Perfmon.

3. DATA REDUCTION

The volume of data WatchTower is capable of accumulating can be daunting. Logging all possible counters on a typical machine at 1 Hz generates between 43 and 86 million values in a single day. In the worst case, one machine could log over 2 Gb in a week, assuming a 64-bit representation for each counter value. This value would grow two to three times larger if logged in ASCII text (as is standard for PDH). A cluster of machines being monitored in this way would generate data that would make storage and analysis difficult, especially real-time analysis.

Our approach to making sense of this large amount of data is to treat it as a dimensionality reduction (DR) problem, where each counter corresponds to a dimension. DR techniques, of which many exist in statistics and machine learning literature [13], reduce high-dimensionality data into a smaller number of dimensions, while retaining a majority of the information. We focus on the question of which subset of the counters captures the interesting dynamics of the system. To find a compact representation, we currently employ correlation elimination.

We use correlation elimination (CE) to select a relevant, statistically interesting subset of counters. CE begins by computing a correlation coefficient matrix. For such a matrix M of size nxn, $M(i,j)$ corresponds to the correlation coefficient between counters i and j. A correlation coefficient is a gauge of the strength of the linear relationship between two variables, where uncorrelated variables have a value near zero, and highly correlated variables have a value near one (thus, $M(i,i)$ will be exactly one). Next, CE applies a K-Nearest Neighbors (KNN) clustering algorithm, using as input the correlation coefficient matrix computed previously. This groups the counters into clusters where all members have a certain positive threshold correlation coefficient or higher to every other counter in the cluster.

Data reduction happens when we choose to keep only one counter from each KNN cluster, discarding the rest. The reasoning behind this trimming is that since all the counters in a particular cluster are all capturing similar information, only one needs to be kept to describe the phenomenon. In this manner, redundant counters are identified and no longer have to be monitored to accurately describe the behavior of a machine.

4. EVALUATION

Our evaluation of the data gathered using a WatchTower application is based on applying the technique described above, offline, to sets of logs from six desktop PCs over a three-day time span. Each set represents between 26 and 115 hours of data collection (depending on work habits and whether the machine was shut down while not being actively used) of 12 performance objects (206 counters) at a rate of 1 Hz on a Windows 2000 machine, where no two machines had the exact same configuration. These computers were used by a diverse (in terms of activities engaged in during the experiment) group of graduate students as well as a professor and an administrative assistant.

The logs were preprocessed and broken up into a directory-file structure to better facilitate analysis. That statistical analysis consisted of three steps, each step being applied to each of the six machines' 12 performance objects. The steps are presented graphically in Figure 4. To more easily explain each step, we will use the example of the `Cache` performance object from the computer Oaklodge.

4.1 Explanation by Example

The `Cache` performance object has 27 counters, and for Oaklodge we collected 263,003 samples (per counter). In the first

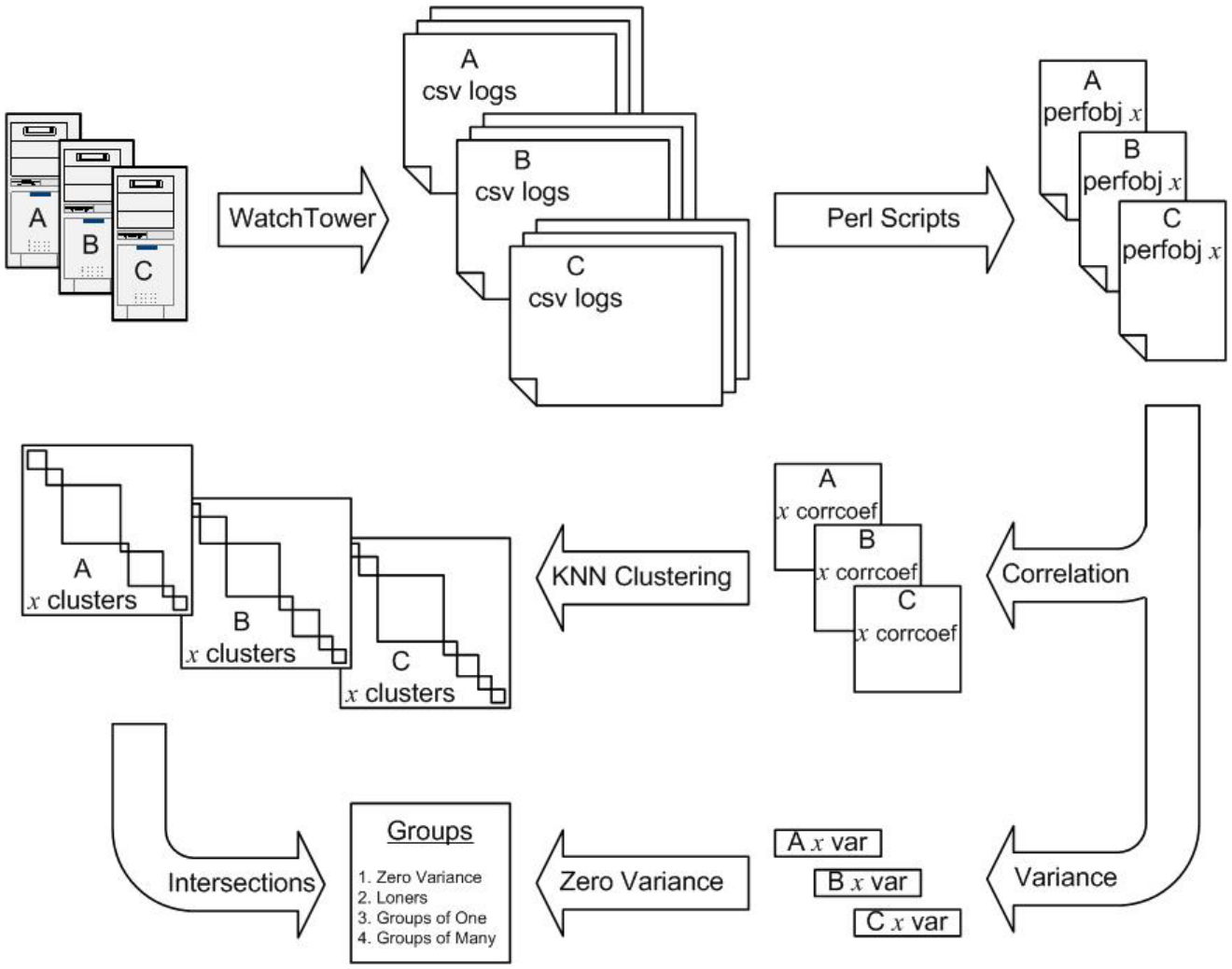

Figure 4: Graphical representation of the current correlation elimination data reduction process.

step of our analysis, we calculated the variance of each counter. Those counters with a variance of zero were set aside because their values did not change throughout the experiment.

In this example, Oaklodge's `Cache` had 9 counters with a variance of zero. The next step of the analysis consisted of applying CE to the performance object's remaining counters. We used a threshold correlation coefficient of 0.8 and for Oaklodge came up with 12 clusters for `Cache`'s 18 remaining counters. Before throwing out any counters though, we applied the first two steps of this process to the `Cache` objects of the other machines in the experiment. In the final step, for each counter in `Cache`, we took the intersection of the clusters from each machine that contained the counter. Also in this step, we took the intersection of the zero variance counters of each machine from the first stage.

The results of the above steps are classes of counters within their performance objects. We note that these classes hold across all the machines since strict intersections were used. We have identified four classes: zero variance, loner, group of one, and group of many. *Zero variances* are, of course, those counters that have a variance of zero. They represent an immediate data reduction because their presence has no statistical significance in the behavior of the machine. Counters that never had a correlation of 0.8 or greater to any other counter are classified as *loners*. These counters are interesting because they characterize some function of the machine that no other counter is tracking. For a counter to be classified as a *group of one*, it must have had correlations of 0.8 or greater to other counters, but these correlations did not hold between all the machines.

The last category a counter can be in is a *group of many*. This category consists of groups of counters that survived the cluster intersections. These counters also represent data reduction. If all the counters in a group have correlations of 0.8 or greater to each other, CE states that only one counter from that group is needed to describe the behavior of the entire group. Thus, we can throw out all but one of the counters from a group of many. The total reduction of data is the combination of those counters thrown out by CE and the zero variance counters.

Figure 5: Performance reduction results. The Lose column is the number of counters that can be thrown out from the total. The last column, Reduction, is the percentage that the Lose number represents of the total. The bottom three rows show the breakdown of reduction across the non-networking objects and the only-networking objects, as well as the total reduction across all objects.

Performance Objects	Loners	Groups of One	Groups of Many	Zero Var	Lose (cntrs)	Total (cntrs)	Reduction
Cache	33%	30%	22%	15%	7	27	26%
Memory	7%	59%	31%	3%	6	29	21%
Objects	17%	50%	33%	0%	1	6	17%
Paging File	50%	50%	0%	0%	0	4	0%
PhysicalDisk (total)	14%	57%	29%	0%	3	21	14%
Processor (total)	30%	40%	20%	10%	2	10	20%
System	29%	47%	12%	12%	3	17	18%
ICMP	4%	30%	7%	59%	17	27	63%
IP	12%	24%	24%	41%	10	17	59%
Network Interface	12%	21%	29%	38%	20	34	59%
TCP	22%	44%	33%	0%	2	9	22%
UDP	40%	0%	60%	0%	2	5	40%
				non Net	22	114	19%
				only Net	51	92	55%
				Total	73	206	35%

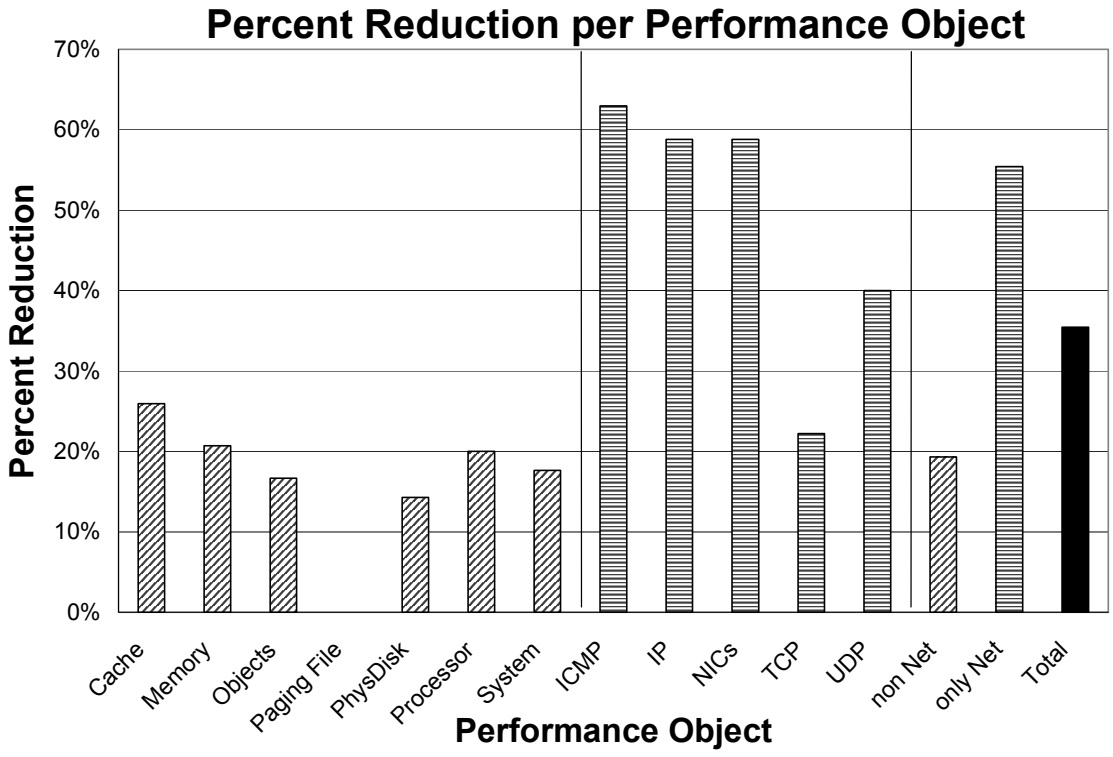

Figure 6: Percent reduction per performance object presented graphically.

4.2 Irregularities among Performance Objects

Unfortunately, not all performance objects were as easy to compare as `Cache`. In particular, three performance objects needed special consideration. The `Processor` and `PhysicalDisk` objects had varying number of instances among machines. This is a consequence of single vs. dual processor and differing number of disks and disk partitions per computer. In these two cases, we used only the counters from the instance `_Total`. These counters represent the total of all the other instances of the performance object, normalized if a percentage value.

The other irregularity was in the `Network Interface` object. One of the six computers in the experiment was a laptop, and its user swapped a wireless network card in and out several times during the experiment, causing the object to have a different number of counters across time. Without a `_Total` instance in the `Network Interface` performance object, we decided to exclude this computer from the `Network Interface` analysis. Thus the results for that specific performance object comprise only five computers. We believe these irregularities do not significantly affect our experimental results.

4.3 Experimental Results

The results of applying the process described above to our experimental data are summarized in Figures 5 and 6. The immediate observation is that 35% of the 206 counters examined can be done away with by CE and zero variance. The next observation is that the performance objects dealing with networking have a much higher rate of reduction than those that do not. Network objects have an overall reduction rate of 55%; non-network objects have a reduction rate of 19%. While network counters represent 45% of the total counters, they make up 70% of the total savings.

Intuitively, the reason for the greater reduction rate for network performance objects lies in their built-in redundancy and echoing effect. Consider the following three counters from the TCP object that survived as a group across all machines: `\Segments/sec`, `\Segments Received/sec`, and `\Segments Sent/sec`. Since `\Segments/sec` is just the sum of `\Segments Received/sec` and `\Segments Sent/sec`, it is redundant to track. Also, segments sent prompt segments to be received, producing an echo and thus a correlation in the counters. Similar phenomena appear in objects pertaining to the disk and processor but do not generate as strong as an effect as do network performance objects.

5. RELATED WORK

Several Windows-specific monitoring systems exist. Closest to our work is HPVM's monitor [17], which is explicitly targeted at Windows NT clusters. Unlike WatchTower, the HPVM monitor provides a complete monitoring infrastructure including communication and high precision clocks. In contrast, WatchTower library routines provide simple sensor capabilities to be used in monitoring tools. NSClient [15] exposes Windows performance counters as a plug-in to the NetSaint [6] monitoring system. None of tools mentioned include a notion of data reduction to capture only the important dynamics in the data.

Performance monitoring and prediction systems such as Remos [9], RPS [4][5], and NWS [21] have limited or nonexistent Windows support. This is not because the code is difficult to port, but rather because of the different nature of sensors on Windows verses UNIX, and the lack of the ability to easily embed Perfmon-like tools. WatchTower provides a simple interface to Windows performance data and has been used to create an RPS sensor. Additionally, such systems would benefit from a reduction in the data sent to them from a sensor.

6. CONCLUSIONS

The experimental results presented here are encouraging. They show that performance data from Windows NT/2000 machines can be reduced while still capturing the overall behavior of the computer. This observation can have an important effect on the archival and real-time analysis of Windows performance data.

For future work, we foresee many avenues. During correlation elimination, we would like to take into account negative correlations during KNN clustering. Currently our algorithm groups counters only by positive correlations. We would also like to employ principal component analysis (PCA) as another statistical technique for data reduction. We have done PCA on performance objects from individual machines but have not yet come up with combined results across a number of machines. We believe PCA may provide better compression, although its results are conceptually harder to grasp and possibly harder to use. Finally, this data reduction work is a first step toward our ultimate goal of state detection.

7. ACKNOWLEDGMENTS

We thank Praveen K. Paritosh for his initial work on the correlation elimination scheme. We would also like to thank Jason A. Skicewicz for his help with the late-night data analysis. The submitted manuscript has been created in part by the University of Chicago as Operator of Argonne National Laboratory ("Argonne") under Contract No. W-31-109-ENG-38 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. Additional support for this work came from the Microsoft Corporation under the Work-for-Others Agreement ACK #853L3, as well as the National Science Foundation through grants ANI-0093221, ACI-0112891, and EIA-0130869.

8. REFERENCES

[1] S. Browne, J. Dongarra, N. Garner, G. Ho, P. Mucci, *A Portable Programming Interface for Performance Evaluation on Modern Processors*, The International Journal of High Performance Computing Applications, Vol. 14, No. 3, pp. 189-204, Fall 2000.

[2] P. Dinda, *Resource Signal Prediction and Its Application to Real-time Scheduling Advisors* (Ph.D. Dissertation), Technical Report CMU-CS-00-131, School of Computer Science, Carnegie Mellon University, February 2000.

[3] P. Dinda, B. Lowekamp, L. Kallivokas, D. O'Hallaron, *The Case for Prediction-Based Best-Effort Real-Time Systems*, Proceedings of the 7th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS99), pp. 309-318, April 1999.

[4] P. Dinda, D. O'Hallaron, *An Extensible Toolkit for Resource Prediction in Distributed Systems*, Technical Report CMU-CS-99-138, Carnegie Mellon University, 1999.

[5] P. Dinda, D. O'Hallaron, *Host Load Prediction Using Linear Models*, Cluster Computing, Vol. 3, No. 4, 2000.

[6] E. Galstad, *NetSaint Network Monitor*, http://www.netsaint.org, 2002.

[7] IBM, *Autonomic Computing*, http://www.research.ibm.com/autonomic, 2002.

[8] K. Ilgun, A. Kemmerer, P. Porras, *State Transition Analysis: A Rule-Based Intrusion Detection Approach*, IEEE Transactions on Software Engineering, Vol. 21, No. 3, pp. 181-199, March 1995.

[9] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, J. Subhlok, *A Resource Query Interface for Network-Aware Applications*, 7th IEEE Symposium on High-Performance Distributed Computing (HPDC7), 1998.

[10] Microsoft Corporation, *Microsoft Platform SDK*, http://www.microsoft.com/msdownload/platformsdk/sdkupdate/, 2002.

[11] Microsoft Corporation, *Perfmon: Performance Monitor*, http://msdn.microsoft.com/library/en-us/vcsample98/html/vcsmpPerfmon.asp, 2002.

[12] R. Rajkumar, C. Lee, J. Lehozcky, D. Siewiorek, *A Resource Allocation Model for QoS Management*, Proceedings of the 18th IEEE Real-Time Systems Symposium, December 1997.

[13] A. Rencher, *Methods of Multivariate Analysis*, Wiley, New York, 1995.

[14] R. Ribler, J. Vetter, H. Simitci, D. Reed, *Autopilot: Adaptive Control of Distributed Applications*, 7th IEEE Symposium on High-Performance Distributed Computing (HPDC7), 1998.

[15] Y. Rubin, *NSClient Official Site*, http://nsclient.ready2run.nl/, 2002.

[16] R. Saavedra-Barrera, A. Smith, E. Miya, *Machine Characterization Based on an Abstract High-Level Language Machine*, IEEE Transactions on Computers, Vol. 38, No. 12, pp. 1659-1679, December 1989.

[17] G. Sampemane, S. Pakin, A. Chien, *Performance Monitoring on an HPVM Cluster*, Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA00), June 2000.

[18] J. Schopf, F. Berman, *Stochastic Scheduling*, Super Computing 1999 (SC99), 1999.

[19] N. Thompson, *Creating a Simple Win32 Service in C++*, http://msdn.microsoft.com/library/en-us/dndllpro/html/msdn_ntservic.asp, November 1995.

[20] P. Tomlinson, *Windows NT Programming in Practice: Practical Techniques from Master Programmers*, R&D Books, 1997.

[21] R. Wolski, *Dynamically Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service*, Proceedings of the 6th High-Performance Distributed Computing Conference (HPDC6), August 1997.