Predictive State Restoration in Desktop Workstation Clusters

David Petrou
Douglas P. Ghormley   Thomas E. Anderson

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

Abstract

Though existing systems for sharing distributed resources in clusters of workstations are generally effective at recruiting idle resources, these systems often have a disruptive effect on desktop workstation users. Even when recruiting computing cycles from strictly idle workstations, a by-product of running foreign jobs is that the virtual memory pages of the original user's idle processes are flushed to disk and the workstation's file cache is disrupted. Consequently, users resuming work after an idle period experience delays while the system restores this state.

This paper presents novel methods for minimizing the disruptions to desktop workstation users in a cluster environment while still maintaining a high utilization of the idle resources of the cluster. Disruptions to the user are reduced by identifying the memory-resident state of the user's processes when the machine becomes idle and then actively restoring that state before the user returns, using measurements of past activity patterns to predict when that user is likely to return. Trace-driven simulations show that this method can predict a user's arrival up to 43% of the time while still recruiting 83% of a workstation's idle cycles.

1 Introduction

Various studies over the years have consistently shown that desktop workstations are idle for a significant amount of time [Theimer et al. 1985, Nichols 1987, Douglis & Ousterhout 1991, Arpaci et al. 1995]. In response to this, many workstation cluster systems have been built which allow users to run jobs on other idle workstations in the cluster. These systems include PVM [Sunderam 1990], Condor [Bricker et al. 1991], LSF [Zhou 1992], Locus [Walker et al. 1983], Butler [Nichols 1987], and Sprite [Douglis & Ousterhout 1991]. However, using an idle workstation to run foreign jobs can negatively impact the user of the workstation when that user resumes work on the workstation. For example, if foreign processes are still running on the workstation when the user returns, the user's processes will have to compete with the foreign processes for CPU time. Many

This work was supported in part by the Defense Advanced Research Projects Agency (N00600-93-C-2481, F30602-95-C-0014), the National Science Foundation (CDA 0401156), California MICRO, the AT&T Foundation, Digital Equipment Corporation, Exabyte, Hewlett Packard, Intel, IBM, Microsoft, Mitsubishi, Siemens Corporation, Sun Microsystems, and Xerox. The authors can be contacted at {dpetrou, ghorm, tea}@cs.berkeley.edu. An on-line version of this paper can be found on the WWW page: http://www.cs.berkeley.edu/~dpetrou/prepage/prepage.html.

systems recognize this and take steps to terminate, migrate, or reduce the priority of foreign jobs upon detecting that the user has returned to the workstation.

However, even if all foreign jobs are eliminated when the user returns to the workstation, there are residual effects from these jobs which can still disrupt the user. Running foreign jobs on the system can cause the operating system to page out the virtual memory pages used by the processes of the workstation's user. Modern workstations can take 30 seconds or more to load the virtual memory pages for a typical user's processes. Additionally, file accesses made by foreign jobs will alter the state of the file cache. Because of these factors, when resuming work after an idle period, users often experience a noticeable delay due to paging activity and file cache misses. The consequence of these disruptions is that users often resist having their workstations integrated into the resource sharing system. In particular, users who rarely run jobs which require remote resources will have little incentive to join the cluster.

It should be the goal of any cluster resource sharing system to minimize the disruptions to users of desktop workstations. Studies have shown that user productivity increases significantly when system response is consistent, even if on average the response is low [IBM 1982, Brady 1986]. Consequently, a single large disturbance followed by high responsiveness is less disruptive to a user's productivity than many small unpredictable disturbances. Therefore, when users return to their workstations after being idle, the system should immediately restore the entire virtual memory and file cache state of the user's processes rather than restoring it slowly over time like current systems do. An ideal system would in fact restore this state *before* the user returns to the workstation, completely hiding the fact that the workstation may have been recruited while the user was absent.

Our approach is to use a combination of active state restoration and prediction to minimize disruptions to the user. When a user stops working, the system identifies which virtual memory pages of the user's processes are resident in memory. When the system detects that the user has resumed work, the system actively brings the identified set of pages back into main memory. This is further augmented by a coarse-grained prediction algorithm which identifies general historical usage trends of a user in order to completely eliminate resource sharing costs in many instances.

We have implemented a prototype for identifying and restoring virtual memory state, currently running as a shared library on Solaris 2.5. We currently use trace-driven simulations to evaluate the prediction algorithm, but have not yet integrated prediction with the prototype. *(Note to the Usenix reviewers: we intend to have these pieces integrated, along with a file cache restoration prototype implemented and evaluated before publication.)*

The remainder of this paper is organized as follows. Section 2 describes the overall system design while Sections 3 and 4 present and evaluate the state restoration prototype and prediction algorithm, respectively. Section 5 highlights the similarities and differences between this system and previous work. We close in Sections 6 and 7 with future work and conclusions.

2 System Design

The system has two primary goals: first, to restore the virtual memory and file cache state of a workstation and second, to predict when the user will return. Figure 1 depicts the three components of our system design that is present on each participating workstation: the Workstation Activity Monitor, the Prediction Engine, and the State Restoration Library.

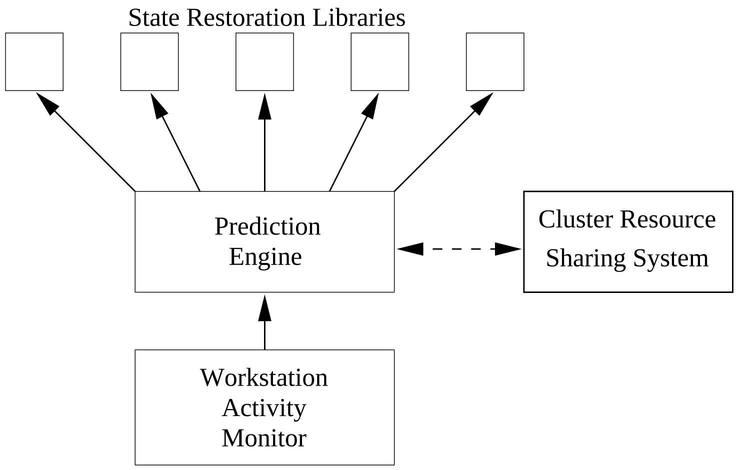

Figure 1: **System design.** The Workstation Activity Monitor detects idle and busy periods of the workstation. The Prediction Engine uses this information to estimate when the user is likely to return to the workstation. The State Restoration Library is responsible for identifying and restoring the virtual memory and file cache state for the user's processes.

2.1 Workstation Activity Monitor

The Workstation Activity Monitor inspects keyboard and mouse activity to determine when the user becomes idle and busy. Most current resource sharing systems have a local agent on each machine which is similar to the Workstation Activity Monitor, for example, Condor's *kbdd* and Utopia's *Load Information Manager*. The Workstation Activity Monitor notifies the Prediction Engine whenever the workstation becomes busy or idle.

2.2 Prediction Engine

The Prediction Engine is responsible for deciding when the machine should be made available to foreign jobs and when to snapshot and restore the state for a user's processes. The Prediction Engine uses the workstation state information from the Workstation Activity Monitor to identify daily and weekly patterns in the user's activity. The prediction algorithm and an evaluation of it are presented in Section 4.

When the user leaves the workstation, the Prediction Engine first signals the State Restoration Libraries to snapshot the virtual memory and file cache state and then notifies the cluster resource sharing system (e.g., Condor [Bricker et al. 1991], LFS [Zhou 1992]) that the machine is available to foreign jobs. When the Prediction Engine predicts that the user is likely to return soon, or if the user returns unexpectedly, the Prediction Engine notifies the cluster resource sharing system that the machine is unavailable to foreign jobs. If a process migration system is available, it can be invoked at this time to remove foreign processes from the workstation. The Prediction Engine also signals the State Restoration Libraries to restore the virtual memory and file cache state of the processes.

The Prediction Engine may also receive information from the cluster resource sharing system indicating that there are no more foreign jobs to run. In this case, the Prediction Engine notifies the State Restoration Libraries to restore the virtual memory and file cache state for the processes.

This eliminates any delays, should the user return to the workstation unexpectedly.

2.3 State Restoration Library

The State Restoration Library is responsible for both identifying and restoring the virtual memory and file cache state for a single process. The library is linked with each application transparently and, on initialization, establishes a communication channel with the Prediction Engine.

In order to identify the virtual memory state of a process, the library first finds the memory segments of the process and then queries the operating system to determine which pages of those segments are resident in memory. To restore the virtual memory state, the library merely reads a single byte from each page which should be resident.

Modern commercial operating systems do not currently provide a way for user level systems to query the state of the file cache. Therefore, in order to approximate this state, the State Restoration Library monitors the file system activity of the process, recording the most recent N file accesses. Then, as an approximation of restoring the file cache, the library replays the accesses to those files. Since most file accesses in a UNIX environment sequentially read or write the entire file [Baker et al. 1991], the system can be simplified by only storing recently accessed filenames rather than each individual access to the file. Restoring the file cache would then involve reading in the entire file.

3 State Restoration Prototype

We have implemented a prototype of the State Restoration Library on Solaris 2.5. The prototype uses the Solaris `/proc` file system to identify valid virtual memory regions for each process and the Solaris `mincore()` system call to identify which pages of those regions are resident in memory. Restoring the virtual memory state is accomplished by simply touching a byte in each page which should be resident, causing the operating system to page it back into main memory. Currently, the prototype does not save or restore file cache state. It also does not multithread the page restoration process, although we expect that multithreading will reduce the time required to restore the state.

We transparently introduce our modifications into existing applications via the dynamic C library. This method requires no modification or access to the source code for the C library. The C library is split into its component object files and then rebuilt with one extra object file containing the state restoration code. By including an `_init` function in the object file, the state restoration code is automatically initialized by the dynamic linker when the C library is loaded. This transparently installs the state restoration code on a user's processes without requiring manual per-process installation or relinking. In Solaris, since the vast majority of applications dynamically link with the C library[1], this will apply to most user applications. An alternative approach would be to insert the state restoration code into the kernel at the system call level using a system such as SLIC [Ghormley et al. 1996].

Upon initialization, the state restoration code registers a signal handler for communication with the Prediction Engine. When the signal is received from the Prediction Engine, the library checks for the existence of a temporary file which indicates which action the library should take: snapshot the process's virtual memory state or restore it.

[1] An analysis shows that all of the programs in a shared filesystem of over 1,400 binaries used at UC Berkeley use this library as well as the local directories `/bin` and `/usr/bin` of Solaris 2.4.

3.1 Limitations

The current prototype has a number of limitations. First, some programs reset the signal handler used by the state restoration code. This prevents the state restoration code from receiving the notifications to snapshot and restore state. Furthermore, this also may cause unexpected behavior in those applications when they receive the signal from the Prediction Engine. To help alleviate this problem, we use the `SIGPWR` signal which most user applications ignore. Currently this limitation has not proved to be a problem. Using an alternative communication mechanism such as Solaris Doors could eliminate this limitation altogether.

Second, between the time that the virtual memory snapshot was made and the state restoration is requested, the application may have unmapped regions of its address space. Consequently, before restoring each virtual memory page, the library must first verify its validity in order to avoid segmentation violations. However, there is a race condition. Another thread in the application may unmap the page after the library verifies it but before accessing it. In practice this has never happened during any of our test runs. This limitation could be removed by informing the operating system to ignore segmentation violations while restoring virtual memory pages.

Finally, the current prototype does not identify or restore file cache state. Since modern commercial operating systems do not provide a user level interface to the file cache, we plan to approximate saving and restoring its state by recording recent file accesses and replaying them. There are three main options for intercepting file system operations. The first option is to modify or binary patch the C library to reroute file operations to the state restoration code. The second option is to use the Solaris `/proc` file system to intercept the file system related system calls. The third option is to use a system such as SLIC [Ghormley et al. 1996] to insert the state restoration code into the kernel at the system call interface.

3.2 Evaluation

To quantify the delays that users experience due to virtual memory paging, we measured the performance of the standard Solaris 2.4 virtual memory system running on a 50 MHz Sparc 20 with 64 MBytes of memory. Figure 2 shows the time required to page in a certain amount of memory. To measure this, a test program allocated an array of the specified size and accessed each page of it to ensure that it was resident in memory. We then ran a cleaner which actively used 64 MBytes of memory, flushing the test array out of memory. The test program then measured the total time required to access each page of the original test array. The results demonstrate that even modestly-sized working sets of 20 MBytes can take over 20 seconds to reload after having been flushed to disk by foreign jobs. Virtual memory set sizes of 20 MBytes and more are typical on modern commercial operating systems.

We then evaluated the State Restoration Library's ability to restore virtual memory pages. The types of applications which will exhibit the worst disruption in a cluster environment are interactive applications which are idle while the user is away, but which consume substantial memory. We chose `emacs` as a representative of this class of application.

On the same platform used above, we ran two experiments. For the first experiment, we ran a copy of `emacs` and loaded a single 14 MByte file. The total amount of virtual memory used by the process was 22 MBytes. We then ran the same memory cleaner used above to flush the virtual memory state out to disk. We then monitored the system paging activity using `vmstat`, signaled the state restoration library to restore the virtual memory state, and began browsing through the 14 MByte file. For the second experiment, we ran exactly the same test but did not actively restore the virtual memory state.

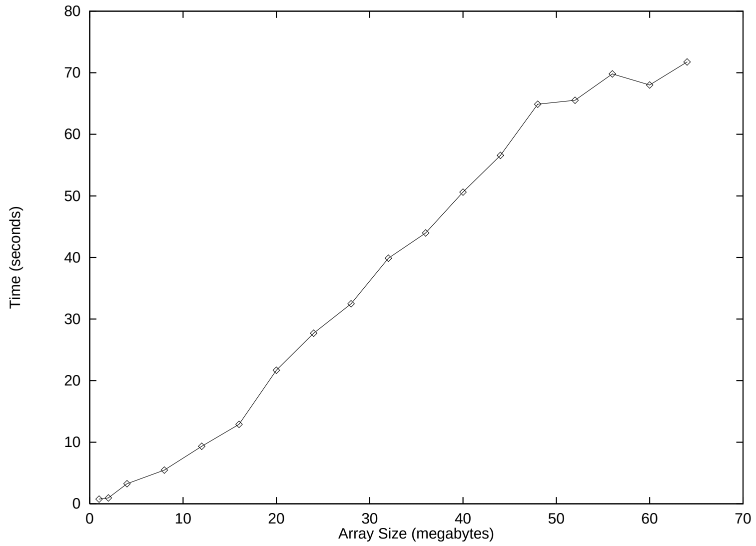

Figure 2: **Native paging performance.** This figure shows the time required to page in various amounts of memory on a 50MHz Sparc20 running Solaris 2.4. Values shown are averaged over 10 runs.

Figure 3 shows the paging activity over time for both experiments. In the first experiment, there was a single large delay at the beginning and no delays at all after that. However, in the second experiment, while browsing the file the user was noticeably interrupted periodically.

4 Prediction Engine

4.1 Recruitment Threshold

Current cluster resource sharing systems typically use a *recruitment threshold* to determine when to make a machine available for running foreign jobs. The recruitment threshold specifies how long the machine must be idle before it is made available for running foreign jobs. The purpose of imposing a recruitment threshold is to avoid using very short idle periods and frequently interrupting the user. Figure 4 shows the performance of different recruitment thresholds for a trace of one user's activity. This shows that recruitment thresholds can avoid up to 90% of potential user interruptions while making 90% of the workstation's idle time available to foreign jobs with a 10 minute recruitment threshold. Unfortunately, once a machine is used to run foreign jobs, the recruitment threshold policy is never able to predict when the user will return to the workstation.

4.2 Prediction Using History

In order to augment the recruitment threshold policy to include prediction, we generate a *probability set* for each workstation. The probability set encodes the percent of previous days that the workstation was in use at a particular time of day. For example, if a workstation was never in use at 7:00am, then the probability set will have a value of 0% for 7:00am. If the machine was in use at 7:00am during half of the previous days, then the probability set will report 50%.

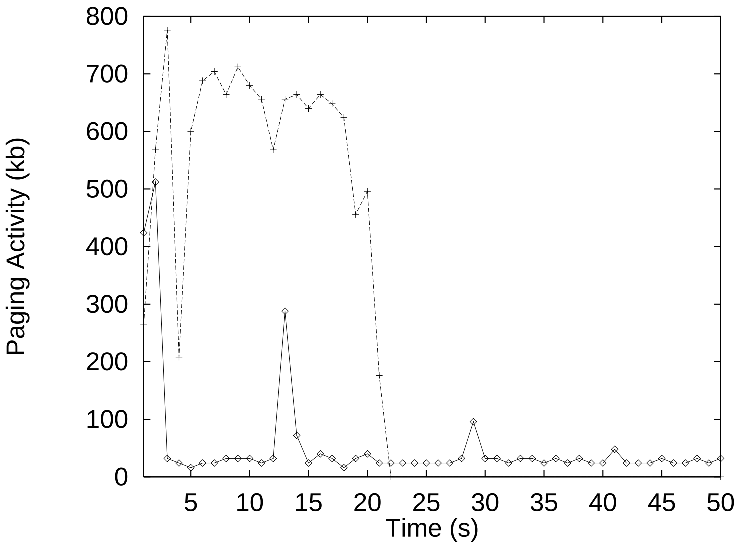

Figure 3: **Paging activity over time.** This figure shows the paging activity over time for an `emacs` process with 22 MBytes of virtual memory state running under Solaris 2.5 on a 167 MHz UltraSparc.

Figure 5 shows the probability set for a real user after 7 days. Notice that the system detects that the machine is always idle in the morning, during a lunch period, and in the evening. However, the machine is never consistently in use during the day since the probability never reaches 100%.

There are two parameters that are used in formulating a probability set: the granularity of time and the periodicity. The granularity indicates the smallest length of time which the prediction algorithm will use for running foreign jobs. The granularity is highly dependent on the cluster resource sharing system being used. For a system which supports process migration, the granularity must be large enough to allow foreign processes to be migrated off of the workstation. Systems which allow foreign processes to run to completion will need very large timeouts so that the foreign processes will terminate before the user returns.

The periodicity of the probability set indicates which past days are used to construct the set. The only two values that we consider here are daily and weekly periodicities. The daily periodicity is used to construct a one day probability set by looking at the same time from consecutive days, as described above. The weekly periodicity requires a week-long probability set and looks at the same day and time from previous weeks. The daily periodicity is conceptually simpler than the weekly periodicity, but treats weekends the same as week days and is therefore less accurate.

There are also two parameters which are used in conjunction with the probability set to determine when to make the workstation available for running foreign jobs. These parameters are the low and high water marks. When the probability of a workstation being busy rises above the high water mark, the algorithm makes the workstation unavailable. This causes the Prediction Engine to notify the cluster resource sharing system that the workstation is unavailable to foreign jobs and that currently running foreign jobs should be eliminated from the system. Then the Prediction Engine would notify the State Restoration Libraries to restore the state of the user's processes.

In order for the prediction algorithm to make the workstation available for running foreign jobs, it first waits for the machine to have no CPU activity for the duration of the recruitment threshold. After that point, it then waits for the probability of the machine being used to fall below the low water mark. Only then is the machine used to run foreign jobs.

4.3 Evaluation

We evaluated the accuracy of the prediction algorithm using trace-driven simulations, assuming an infinite load of foreign jobs. The traces used were taken directly from [Arpaci et al. 1995] and were generated over the course of a month on 53 assorted workstations in the Computer Science Division at U.C. Berkeley. The original traces were of varying quality and duration, so we selected 20 which had the longest duration and fewest gaps in the trace data.

The simulator recorded two primary statistics during the simulation of the traces: the percentage of time that the algorithm correctly predicted the user's return and the percent utilization of the idle time of the workstation. Note that the first statistic does not mean that the algorithm predicted *when* the user returned, only that it correctly made the workstation unavailable to foreign jobs before the user returned.

For purposes of simplifying the simulation space, we assumed zero cost state restoration and a fixed granularity of one second. We start using a periodicity of one day until a week of data is processed, and then switch to a periodicity of one week. We used the optimal recruitment threshold of 10 minutes from Figure 4.

We ran the simulator over a wide range of low and high water marks, using a single user's trace. Once we determined the optimal values for that user, we applied those same values across all users. The results are shows in Figure 6. The optimal values for the single user were found to be a lower water mark of 20% and a high water mark of 50%. That is, the machine is made available for running foreign jobs only when the probability set value for the next time granularity falls below 20%.

Figure 6 shows that the prediction algorithm succeeds 9 to 43 percent of the time, while still utilizing 75 to 93 percent of the idle time of the workstation.

5 Related Work

Significant work has been done that is specifically concerned with the issues of integrating desktop workstations into cluster environments. Previous systems such as Condor [Bricker et al. 1991], LSF [Zhou 1992], Sprite [Douglis & Ousterhout 1991], and Butler [Nichols 1987], attempt to reduce interruptions to the user by introducing a recruitment threshold, or timeout, before using the machine. This simple technique eliminates most false starts of the system and avoids interrupting the user after brief idle times. However, none of these systems make an effort to predict when the user will return once the machine has been used to run foreign jobs, nor do they attempt to actively restore the state of the user's processes.

[Pfister 1995] discusses the trade-offs between machine room versus desktop clusters, warning designers that desktop workstation users must not be inconvenienced by sharing their resources. Experience at DEC SRC and with Sprite and U.C. Berkeley showed what can happen when users feel inconvenienced by the system: users actually tried to deceive the system by tapping their keyboards periodically to avoid having their machine used for foreign jobs. Encouraging participation in a cluster requires reducing perceived disruptions to the user caused by foreign jobs.

[Arpaci et al. 1995] examines the interaction of parallel and sequential workloads with desktop workstation users in a cluster environment. In their traced environment they showed that, depending on the definition of idleness, 60 to 70% of the desktop workstations were available even in the middle of the day. Their system minimized parallel program slowdowns by implementing a 3 minute recruitment threshold. However, because their system did not attempt to predict workstation usage, they suggest a social contract mechanism to prevent any particular user from being overly interrupted. Unfortunately, their study shows that even limiting the number of user interuptions to 8 per day can cause a factor of 2 slowdown for parallel applications. In addition, there is a large penalty for not scheduling parallel programs on workstations that have a high likelihood to remain idle. A large class of parallel applications require fine-grained synchronization and any slow node can ruin performance. Hence, the ability to predict when a workstation will become non-idle is also beneficial for the performance of parallel jobs.

6 Future Work

6.1 Better Prediction

Predicting the time at which a user will resume work requires the identification of patterns in past behavior. This is precisely what compression algorithms are designed to do, though in a different context. The theoretical basis for using compression algorithms for prediction is presented in [Feder et al. 1992]. Adapting a compression algorithm such as Ziv-Lempel [Ziv & Lempel 1978] may perform a better job at detecting workstation usage patterns than the technique described earlier. With better prediction, we can lower user interruptions and raise the availability of workstations as shared resources.

6.2 Integration with Resource Sharing Package

We will integrate our system with GLUnix, the global operating system support layer of the Berkeley NOW project [Anderson et al. 1995]. Being the primary developers for GLUnix, we have had much pressure to integrate this system from desktop workstation users experiencing state restoration delays.

6.3 Eliminating State Restoration Restrictions

During the discussion of the prepaging prototype, we presented some limitations with our approach. SLIC [Ghormley et al. 1996], a package for adding extensions to modern operating systems may be used to provide more robust prepaging. A prepaging extension to the operating system may catch page faults to monitor one's working set, and later cause pages to be restored without modifying the dynamic C library, and without the limitations associated with this.

Aside from memory pages, file caches are cleaned when a workstation has other loads on it. Work will be done to measure the detrimental effect of losing one's file cache. SLIC may also provide us with a way to correctly monitor and restore a user's file cache by providing access to file caching primitives that reside inside the kernel.

6.4 Detecting Users for Prepaging

An interesting possibility is for users to wear special unique badges so that when they enter the workplace, the cluster is aware that they may start using their workstations soon. The Prediction

Engine and State Restoration Libraries can use this information to perform their operations. Using badges in this manner is similar to the system at Xerox PARC in which telephone calls are routed to the telephone nearest the callee.

7 Conclusion

This paper presented two methods for minimizing disruptions to the desktop workstation user in a shared resource environment: actively restoring the virtual memory and file cache state for a user's processes, and predicting the future behavior of a user by monitoring past activity. We have demonstrated through direct measurements of a prototype that actively restoring state can decrease the long-term disruptions experienced by users after their workstation have been used to run foreign jobs. Using trace-driven simulation we have shown that it is possible to completely eliminate 9-43% of the potential disruptions to the user in a shared resource system and that this can be done while utilizing 75-93% of the idle time of the workstation. The combination of these two factors indicate that current cluster resource sharing systems can benefit from these methods, significantly reducing the participation costs for users.

8 Acknowledgments

We express our thanks to Remzi Arpaci and Amin Vahdat for providing the traces used in the analysis of this paper's algorithms.

References

[Anderson et al. 1995] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team. "A Case for NOW (Networks of Workstations)". *IEEE Micro*, February 1995.

[Arpaci et al. 1995] R. Arpaci, A. Dusseau, A. Vahdat, L. Liu, T. Anderson, and D. Patterson. "The Interaction of Parallel and Sequential Workload on a Network of Workstations". In *Proceedings of Performance/Sigmetrics*, May 1995.

[Baker et al. 1991] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. "Measurements of a Distributed File System". In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 198–212, October 1991.

[Brady 1986] J. Brady. "A Theory of Productivity in the Creative Process". In *IEEE CG&A*, May 1986.

[Bricker et al. 1991] A. Bricker, M. Litzkow, and M. Livny. "Condor Technical Summary". Technical Report 1069, University of Wisconsin—Madison, Computer Science Department, October 1991.

[Douglis & Ousterhout 1991] F. Douglis and J. Ousterhout. "Transparent Process Migration: Design Alternatives and the Sprite Implementation". *Software - Practice and Experience*, 21(8):757–85, August 1991.

[Feder et al. 1992] M. Feder, N. Merhav, and M. Gutman. "Universal Prediction of Individual Sequences". *IEEE Transactions on Information Theory*, 38(4):1258–1270, July 1992.

[Ghormley et al. 1996] D. P. Ghormley, D. Petrou, and T. E. Anderson. "SLIC: Secure Loadable Interposition Code". May 1996. Submitted for publication. A draft can be downloaded from `http://now.cs.berkeley.edu/Slic/`.

[IBM 1982] IBM. "The Economic Value of Rapid Response Time". In *GE20-0752-0*, November 1982.

[Nichols 1987] D. Nichols. "Using Idle Workstations in a Shared Computing Environment". In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pp. 5–12, November 1987.

[Pfister 1995] G. F. Pfister. *In Search of Clusters.* Prentice Hall, Inc., 1995.

[Sunderam 1990] V. Sunderam. "PVM: A Framework for Parallel Distributed Computing". *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[Theimer et al. 1985] M. Theimer, K. Landtz, and D. Cheriton. "Preemptable Remote Execution Facilities for the V System". In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pp. 2–12, December 1985.

[Walker et al. 1983] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. "The LOCUS Distributed Operating System". In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pp. 49–70, October 1983.

[Zhou 1992] S. Zhou. "LSF: load sharing in large-scale heterogeneous distributed systems". In *Proceedings of the Workshop on Cluster Computing*, December 1992.

[Ziv & Lempel 1978] J. Ziv and A. Lempel. "Compression of Individual Sequences Via Variable Rate Coding". In *IEEE Transactions on Information Theory*, September 1978.

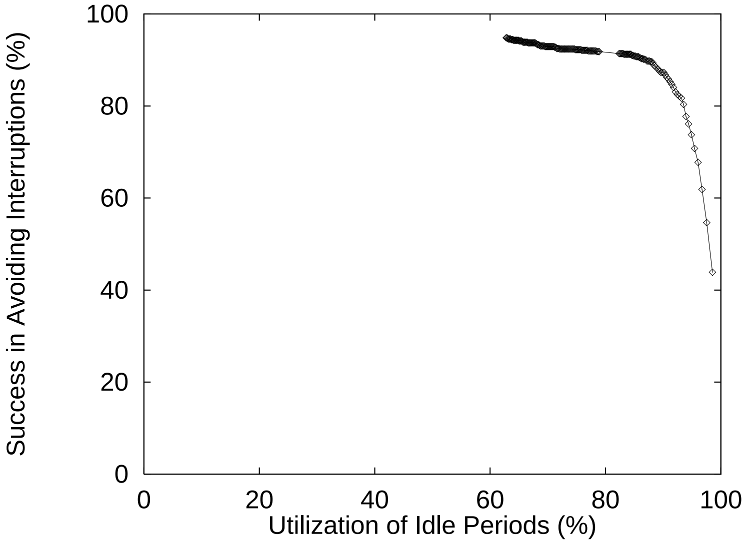

Figure 4: **Recruitment Threshold Performance**. This graph shows the performance of an algorithm utilizing a recruitment threshold with no prediction facility.

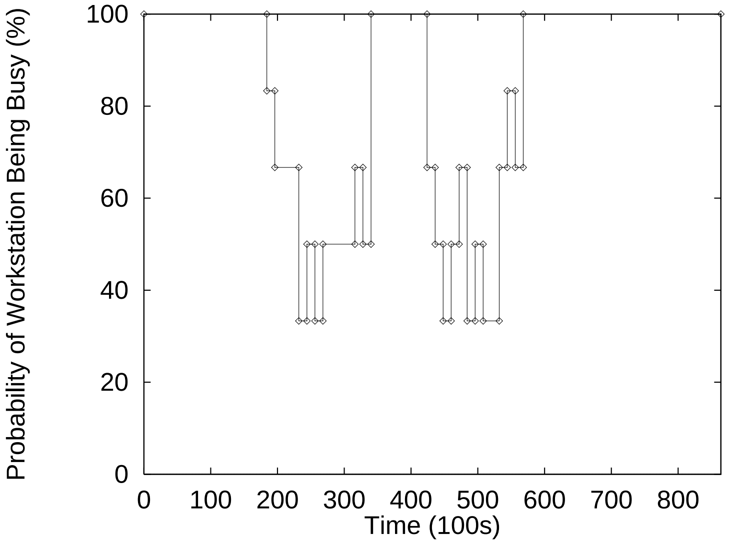

Figure 5: **Sample probability set**. This graph shows a sample probability set for a real user after 7 days of history.

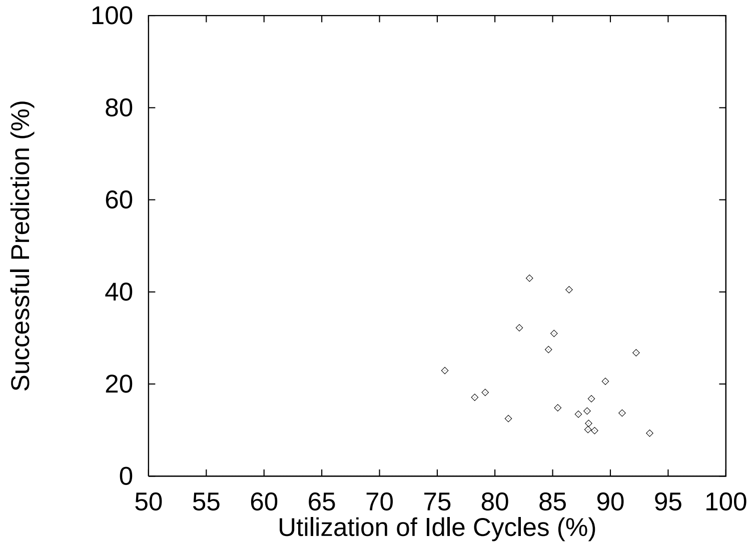

Figure 6: **Prediction accuracy vs. workstation utilization**. This graph shows the prediction algorithm performance for the top 20 machines using a low water mark of 20% and a high water mark of 50%.