GLUnix: a Global Layer Unix for a Network of Workstations

Douglas P. Ghormley, David Petrou, Steven H. Rodrigues,

Amin M. Vahdat, and Thomas E. Anderson

{ghorm,dpetrou,steverod,vahdat,tea}@cs.berkeley.edu

Computer Science Division

University of California at Berkeley

Berkeley, CA 94720

August 14, 1997

Abstract

Recent improvements in network and workstation performance have made clusters an attractive architecture for diverse workloads, including sequential and parallel interactive applications. However, although viable hardware solutions are available today, the largest challenge in making such a cluster usable lies in the system software. This paper describes the design and implementation of GLUnix, an operating system layer for a cluster of workstations. GLUnix is designed to provide transparent remote execution, support for interactive parallel and sequential jobs, load balancing, and backward compatibility for existing application binaries. GLUnix is a multi-user, user-level system which was constructed to be easily portable to a number of platforms.

GLUnix has been in daily use for over two years and is currently running on a 100-node cluster of Sun UltraSparcs. Performance measurements indicate a 100-node parallel program can be run in 1.3 seconds and that the centralized GLUnix master is not the performance bottleneck of the system.

This paper relates our experiences with designing, building, and running GLUnix. We evaluate the original goals of the project in contrast with the final features of the system. The GLUnix architecture and implementation are presented, along with performance and scalability measurements. The discussion focuses on the lessons we have learned from the system, including a characterization of the limitations of a user-level implementation and the social considerations encountered when supporting a large user community.

1 Introduction

Historically, computing models have been separated into three distinct traditional paradigms, each optimized for a particular class of applications: desktop workstations were primarily used for low-demand, interactive jobs; centralized compute servers such as supercomputers and mainframes were used for high-demand, batch jobs; and massively parallel processors (MPPs) were used for parallel jobs. However, technology trends of the past two decades have steadily eroded the boundaries of these paradigms. The narrowing of the performance gap between commodity workstations and supercomputers as well as the availability of high-speed, commodity local area networks have led to the ability to run both compute-bound and massively parallel programs on networks of workstations (NOWs) [2]. Consequently, NOWs are capable of servicing all three of the historical workloads.

Given the availability of commodity high-performance workstations and networks, the primary challenge in building a NOW lies in the system software needed to manage the cluster. Ideally, this software should provide support for interactive parallel and sequential jobs, transparent remote execution of existing applications using traditional UNIX I/O and job control semantics, dynamic load balancing, and rapid portability to the latest architectures.

When we set out in 1993 to select the system layer for our NOW, some of the features listed above were present in a few existing systems, but no system provided a complete solution. LSF [37] and Utopia [36] provided support for remote execution of interactive sequential jobs, but did not support parallel jobs or dynamic load balancing. PVM [29] did provide support for parallel jobs but did not provide gang scheduling for closely synchronized parallel jobs. PVM also did not integrate the parallel job environment with interactive sequential jobs or provide dynamic load balancing. Condor [8] was able to dynamically balance cluster load by migrating jobs, but only if those jobs were restricted to a certain set of features and were re-linked with a special library. Further, Condor did not transparently support existing binaries and operated only in batch mode. Sprite [25] provided dynamic load balancing through process migration [15] and maintained full UNIX I/O and job control semantics for remote jobs. However, it did not support parallel jobs and was a complete custom operating system, making hardware updates difficult to track.

GLUnix (Global Layer Unix) was desiged to incorporate the features listed above into a single, portable, coherent, clustering system. Its design supports interactive and batch-style remote execution of both parallel and sequential jobs, maintains traditional UNIX semantics for I/O and job control, provides load balancing through intelligent job placement and dynamic process migration, and tolerates single-node faults. For reasons discussed in this paper, GLUnix was unable to completely fulfill this vision while remaining a portable, user-level solution. GLUnix, as it currently stands, maintains most, but not all, UNIX I/O semantics, provides load balancing at job startup, and can tolerate most single-node faults. Despite these limitations, GLUnix has been in daily use for nearly two years and has had 130 users and run over 1 million jobs in that time period. User experience with the GLUnix has been largely positive, and the system has enabled or assisted systems research and program development in a number of areas.

While our discussion centers around GLUnix, the lessons learned are generally applicable to any system layer attempting to abstract network-wide resources. We begin by describing in more detail our goals and the abstractions exported by GLUnix. This leads to an architectural overview of GLUnix and an evaluation of its scalability and performance. We then focus on our experiences with constructing a portable, user-level cluster system, and the difficulties we faced in building transparent remote execution image using only user-level primitives. Next, we describe our experiences with supporting a substantial user community, including additions that were made to the system to satisfy user demands. The paper concludes with a discussion of related work, future directions, and our conclusions.

2 Goals and Features

2.1 History

GLUnix was originally conceived as the global operating system layer for the Berkeley Network of Workstations (NOW) project. A primary goal of the Berkeley NOW project was to construct a platform that would support a mixture of both parallel and sequential interactive workloads on commodity hardware and to put this platform into daily use as a means of evaluation. The feasibility of supporting integrated parallel and sequential workloads was established by an early simulation study [5] of workloads measured from a 32-node CM-5 at Los Alamos National Lab and a 70-node workstation cluster at U.C. Berkeley. The study concluded that harvesting idle cycles on a 60-node workstation cluster could support both the parallel and sequential workloads with minimal slowdown.

Given the goals for the Berkeley NOW project, we identified the following set of requirements for our cluster software system:

- *Transparent Remote Execution:* Applications should be unaware that they are being executed on remote nodes. They should retain access to home node resources including shared memory segments, processes, devices, files, etc.
- *Load Balancing:* The system should provide intelligent initial job placement policies to determine the best location to execute each job. The system should also dynamically migrate jobs to new nodes either as load becomes unbalanced or as nodes are dynamically added or removed from the system.
- *Coscheduling of Parallel Jobs:* Efficient execution of communicating parallel jobs requires that the individual processes be scheduled at roughly the same time [26, 6, 16].
- *Binary Compatibility:* The system should be able to provide remote execution, load balancing, and coscheduling without requiring application modification or relinking. Existing applications should be able to transparently benefit from new features.
- *High Availability:* When a node crashes, the system should continue to operate. It is acceptable for jobs which have a footprint on a failed node to be terminated.
- *Portable Solution:* The system should be easily portable to multiple architectures and operating systems.

2.2 Architecture Alternatives

We considered a number of alternative strategies in designing the initial GLUnix architecture. The first, and perhaps most important, was to decide what to leverage from the underlying system. Our options included (i) building a new operating system on top of the raw hardware, (ii) modifying an existing operating system, and (iii) attempting to "build on top" of existing systems, possibly with minor operating system modifications. We eliminated the option of building an operating system from scratch since the time involved in such an effort was incompatible with the Berkeley NOW project's goal of supporting end users within the project's lifetime. Previous efforts in building operating systems from scratch have demonstrated that several years are required to produce a full-featured system capable of booting, remaining stable for extended periods, and running existing binaries. Further, such systems are difficult to port different platforms, meaning that they often fail to keep up with rapidly advancing hardware platforms.

The two primary options we considered were making modifications to existing operating systems or building a global runtime environment at the user level. Modifying an existing operating system kernel has a number of advantages. First, our implementation need not be constrained to services available at the user level. Second, if the performance of existing system abstractions proved unacceptable, optimizing or reimplementing those abstractions would be a valid option. Finally, existing applications could run unmodified, transparently taking advantage of improved semantics of the underlying system (e.g., load balancing and process migration). Unfortunately, Modifying an existing kernel considerably impacts system portability: not only are kernel modifications not portable across platforms, but kernel modifications are often not even valid across different releases of the same operating system [35]. In 1993, the primary operating systems available for state-of-the-art architectures were commercial products. Modifying a commercial operating system limits the ability to distribute the resulting system. Further, the time required to obtain source, learn, modify, and debug a commercial kernel is significant.

The alternative to modifying the operating system kernel was to build a global runtime environment at the user level, leveraging available operating systems primitives as building blocks to implement our requisite feature list. This approach essentially reverses the list of advantages and disadvantages outlined above for modifying an existing operating system. A user-level system is simpler to implement, easier to debug, and more portable. The primary disadvantage is that a user-level implementation is limited to the functionality exported by the underlying operating system; this exported fnctionality may prove insufficient. Even in the case where exported system primitives are sufficient, the performance of those primitives (system calls, page mappings, signal delivery) may be unacceptable.

In particular, we were concerned that the limitations of previous user level systems [8] may have been an indication that the functionality needed to implement remote execution transparently was not available at the user level. As a simple example, maintaining the semantics of UNIX signals for remote jobs requires intercepting either the signal request or the signal delivery and sending the request to the node executing the job in question. While intercepting such events is relatively straightforward when modifying the underlying operating system, it is often more difficult to implement portably at the user-level. Section 7 details this and other problems associated with user-level solutions.

Despite the above limitations, we believed that binary rewriting techniques [33, 21] or new kernel extension technologies could be employed to catch and redirect the appropriate system events to support transparent remote execution. In the end, the goals of rapid prototyping, porting to multiple platforms, and distributing the software to multiple sites outweighed the possible performance and transparency concerns of the user level approach; we chose to implement the system without making modifications to the underlying operating system.

Given our decision to implement GLUnix at the user level, we then faced the question of whether all programs would run under GLUnix or whether we would begin with only GLUnix-aware programs. Since we were initially unsure of GLUnix's ability to provide fully transparent remote execution, we chose a multiphased approach. Initially, we would implement two different process domains: the standard UNIX domain and a new GLUnix domain. A specific request would have to be made to run a program under the GLUnix domain. As we identified and resolved the transparency issues, we would gradually incorporate an increasing number of UNIX programs into the GLUnix domain until all processes were subsumed into GLUnix. This path would enable us to maintain the stability of the nodes during initial development and debugging (for example, we did not want operating system daemons running under GLUnix until it was stable).

We next had to choose the high level system architecture for GLUnix. We considered both a centralized master and a peer-to-peer architecture. Peer-to-peer architectures have the advantage of improved availability (no single point of failure) as well as offering the potential for improved performance (no centralized bottleneck). Centralized designs, on the other hand, are easier to design, implement, and debug since they simplify state management and synchronization. Testaments to the surprising scalability of the centralized architecture of V [31, 11] suggested that a centralized design might have acceptable performance for the system scale we were targeting (100 nodes). We decided, therefore, to begin with a centralized prototype and to later distribute and replicate the components of the system which proved to be central bottlenecks or single points of failure.

The third and final design decision involved the internal architecture of individual GLUnix components. We considered both multi-threaded and event-driven architectures. Multi-threaded architectures have two primary advantages over event-drive architectures: increased concurrency with the potential for improved performance, and the relative simplicity of dealing with blocking operations. In a multi-threaded architecture, a server can allow one thread to block while waiting on a reply from a client while other threads will continue to make progress on other communication channels; if necessary, a master thread can be responsible for timing out threads which have blocked while communicating with unresponsive clients. In contrast, an event-driven architecture requires more programming effort to deal with blocking operations. An event-driven architecture cannot generally block awaiting a reply from a client, but would instead have to package up the current state information and return to the event loop, processing the client's reply as a separate event. Event driven models are easier to reason about and simpler to develop since a whole class of race conditions can be avoided. In the end, the lack of a portable kernel thread package and thread-aware debuggers in 1993[1] convinced us to adopt an event-driven architecture for GLUnix.

2.3 Retrospective

Since the initial GLUnix design in 1993, a number of technological advances have occurred which would impact the goals and design choices of GLUnix. While GLUnix was designed to provide the features available in desktop workstations, supercomputers, and massively parallel processors (MPPs), symmetric multi-processors (SMPs) have, in recent years, become increasingly cost effecitve and have become more widely used. Today, the supercomputers and MPPs that dominated the high-end and scientific computing niche have largely been replaced by SMPs. The tight processor integration offered by SMP operating systems presents a number of advantages over traditional MPP operating environments, including low-cost dynamic load balancing, low-cost shared memory, and support for interactive jobs. SMPs also trivially provide a single system image; that is, all resources of the SMP appear to the user and to programs as part of a single machine. To compete effectively with SMP operating systems, a cluster operating system would need to provide a single system image comperable to that of SMPs. Our experience has demonstrated, however, that it is very difficult for a user-level cluster system to provide the illusion of a single system image (see Section 7). In response to the obstacles we have encountered, we are developing SLIC (Secure Loadable

[1] For example, one of our target architectures, HP-UX, did not support kernel threads at all until 1996.

Interposition Code), a system to enable the transparent interposition of trusted extension code into existing commodity operating systems. SLIC is intended to overcome the limitations of developing at the user level and will enable GLUnix to provide functionality comperable to the single system image of SMPs.

A second technological advancement that has occured in the past four years is that kernel modification has become a more attractive option. Commercial operating systems such as Solaris provide well-designed interfaces for new kernel modules, such as file systems, scheduling policies, and network drivers. Distributing such modules does not require distribution of source code; these modules are dynamically linked into the kernel without requiring a recompilation or even system reboot. Another recent development is the wider availability of stable, well-documented, free operating systems such as Linux and NetBSD, both of which are capable of running on multiple hardware platforms. These operating systems also include significant advances in kernel debugging support which were unavailable in 1993.

With respect to our other design decisions, we have found that the performance of a centralized master does scale to 100 nodes. Additional performance advances can be easily achieved by porting GLUnix to use the 1.28 Gb/s Myrinet instead of the switched 10 Mb/s ethernet it is using today. However, the performance of parallel program co-scheduling suffers as a result of the user-level mechanisms employed in the implementation (see Section 6.1). Finally, given the system failures we have encountered as a result of our single-threaded design (as described in Section 7), and the more wide-spread availability of POSIX-compliant threading packages and thread-aware debuggers, today we would implement GLUnix as a multi-threaded program.

2.4 Final Features

GLUnix, as actually built, performs remote execution, but does not do so with complete transparency. GLUnix provides load balancing through intelligent job placement, but does not provide migration. Section 7 discusses some of the reasons for these limitations. GLUnix does provide coscheduling of parallel jobs and runs existing application binaries. As described in section 4, GLUnix is able to tolerate any number of daemon node failures, although it cannot tolerate a failure of the node running the GLUnix master process.

3 GLUnix Abstractions

To achieve the goal of providing remote execution of parallel and sequential applications, GLUnix provides several new abstractions, borrowing heavily from MPP environments such as that on the CM-5, and extends some existing UNIX abstractions. The abstractions we decided upon include globally unique network process identifiers (NPIDs) for GLUnix jobs, Virtual Node Numbers (VNNs) to name constituent processes of parallel programs, signal delivery to remote applications, I/O redirection. These abstractions, and a programming API to access them, are described in more detail in the following subsections.

3.1 Network Processes

To provide a cluster-wide process name space, GLUnix introduces Network PIDs (NPIDs) and Virtual Node Numbers (VNNs). NPIDs are globally unique, GLUnix-assigned 32-bit process identifiers used to identify both sequential and parallel programs throughout the system. We use the term *parallel program* to refer to multiple instances of the same binary, all working together as a unit toward a single objective (i.e., the Single-Program-Multiple-Data (SPMD) model). The *parallel degree* of a program refers to the number of processes comprising the program. In this model, sequential and parallel programs are treated largely the same; sequential programs merely have a parallel degree of 1. GLUnix uses a single NPID to identify all *N* processes in an *N*-way parallel program, providing a convenient method for signaling all processes in a parallel program. A single NPID space is used for both parallel and sequential programs. This enables the standard job control signals such as `Ctrl-C` and `Ctrl-Z` to work as expected for both parallel and sequential programs.

To facilitate communication among the processes of a parallel program in the face of migration, GLUnix uses VNNs to distinguish individual processes of a parallel program. At execution time, GLUnix assigns a separate VNN to each process of a *N*-way parallel program, numbered 0 through *N*-1. The VNN name space is local to each NPID, providing a location independent name for each process: <NPID, VNN>.

Together, NPIDs and VNNs allow for the global naming of entire programs as well as individual components of parallel programs. As discussed in Section 3.3, GLUnix operations such as signaling can be directed to an entire NPID, or just a particular VNN of a NPID. These abstractions are needed to enable transparent remote execution and dynamic process migration. Communicating processes cannot rely on the location-dependent naming schemes, such as IP-address, but must rely on a level of virtualization provided by the system, in our case, NPIDs and VNNs.

3.2 GLUnix Tools

glurun	run sequential or parallel applications on the cluster
glukill	send a signal to a GLUnix job
glumake	parallel version of GNU make facility
glush	modified `tcsh` to run user commands remotely without requiring an explicit `glurun`
glups	query currently running jobs
glustat	query current status of all NOW machines

Table 1: The GLUnix tools provided to users. Each of these utilities is linked with the GLUnix library to provide a convenient interface to some GLUnix service.

To interact with GLUnix and to manipulate NPIDs and VNNs, we created a set of command-line tools analogous to traditional UNIX counterparts. The tools and their functionality are described in Table 1. The `glurun` utility allows unmodified applications to take advantage of GLUnix functionality; `glurun` distributes jobs across one or more nodes using GLUnix to select the least loaded nodes in the cluster. The `glumake` utility enables users to distribute compilations across the entire cluster, often speeding up compilation by an order of magnitude or more; `glumake` quickly became the most popular GLUnix tool. Tools such as `glups` and `glustat` monitor cluster status.

3.3 Programming Interface

```
int Glib_Initialize();
int Glib_Spawn(int numNodes, char *progPath, char **argv);
int Glib_CoSpawn(Npid npid, char *progPath, char **argv);

int Glib_GetMyNpid();
int Glib_GetMyVNN();

int Glib_Signal(Npid npid, VNN vnn, int sig);
int Glib_Barrier();
```

Table 2: Sample functions of the GLUnix API implemented by the GLUnix library.

Table 2 shows some common GLUnix library functions. `Glib_Spawn()` is the basic function to run a program under GLUnix. `Glib_CoSpawn()` is used to place an application on the same nodes as a currently running application; this function is currently used by the Mantis [22] parallel debugger. `Glib_GetMyNpid()` and `Glib_GetMyVNN()` return the Npid and VNN, respectively, of the requesting application. `Glib_Signal()` is used to send a signal to one or more of the VNNs of a program running under GLUnix. The `Glib_Barrier()` function is used to synchronize parallel applications.

3.4 Remote Execution

3.4.1 Backward Compatibility

Using the `glurun` shell command or the modified GLUnix shell, `glush`, programs executed from the command line are distributed to the least loaded node in the cluster. To both the end user and application program, the system attempts to provide the illusion that jobs are running on a single large computer. User keystrokes are sent to the node where the job is physically running and output is piped back to the user's shell on the node where the process was started. Similar redirection support is provided for terminal I/O and UNIX signals. This redirection functionality is provided through a GLUnix startup process that acts as a lightweight proxy for the remote job. When a user runs an application under GLUnix, the process which is actually started by the shell is the GLUnix startup process. Implementation details are described in Section 4.3.

3.4.2 Parallel Program Support

GLUnix supports the Single Program Multiple Data (SPMD) model of parallel programming in which multiple copies of the same program binary run on multiple nodes of the system. Just like MPP operating systems, GLUnix provides specialized scheduling support for parallel programs in the form of barriers and coscheduling [26].

Parallel programs under GLUnix invoke a barrier synchronization by calling the `Glunix_Barrier()` library call. When an individual process of a parallel program enters the barrier, it is blocked until all other processes of the program have also entered the barrier. Once this occurs, all processes in the program are released from the barrier to resume execution. Since individual parallel program instructions are not executed in lock-step in the SPMD model, barriers are necessary to synchronize cooperating processes, enabling programmers to make assumptions about the state of computation and data in other processes.

Coscheduling, or gang scheduling, ensures that all processes of a single parallel job are scheduled simultaneously and provides coordinated time-slicing between different parallel jobs. In the absence of coscheduling, the individual operating systems in the NOW would independently schedule the processes of a parallel application. User-level cluster systems such as PVM [29] typically take this approach. However, a number of studies [12, 17, 7, 16] demonstrate that this technique leads to unacceptable execution times for frequently-communicating processes. This slowdown occurs when a process stalls while attempting to communicate or synchronize with a currently unscheduled process. The implementation of both barriers and coscheduling is described in Section 4.4.

4 GLUnix Implementation

This section provides an overview of the GLUnix architecture. In particular, this section lays out the assumptions that the system makes, how the system implements the features described above, and the lessons we learned concerning the architecture of a cluster system.

4.1 Assumptions

In constructing GLUnix, we made two primary assumptions concerning the target environment: shared filesystems and homogeneous operating systems. Since the Berkeley NOW proposal included a high performance global file system [3], GLUnix assumes that each node of the cluster shares the same file systems, making pathnames valid on all nodes. Further, GLUnix uses a shared directory to store the network address of the GLUnix master for bootstrapping. Our cluster uses NFS-mounted file systems for this purpose.

The second assumption was homogeneous operating systems and binary formats across all nodes in the cluster. That is, a single GLUnix cluster can include SparcStation 10's, 20's, and UltraSparcs running Solaris 2.3—2.6, but it cannot include both Sparc and 80x86 machines. This assumption simplifies the implementation of GLUnix by eliminating difficulties with byte ordering, data type sizes, and data structure layout for GLUnix internal communication services and ensures that application binaries can run on any machine in the cluster. Enabling multi-platform support for GLUnix would require porting to a network data transport mechanism such as XDR [28] and developing a mechanism for supporting multi-platform binaries.

4.2 System Architecture Overview

This section presents a brief overview of the individual GLUnix components making up the internal architecture.

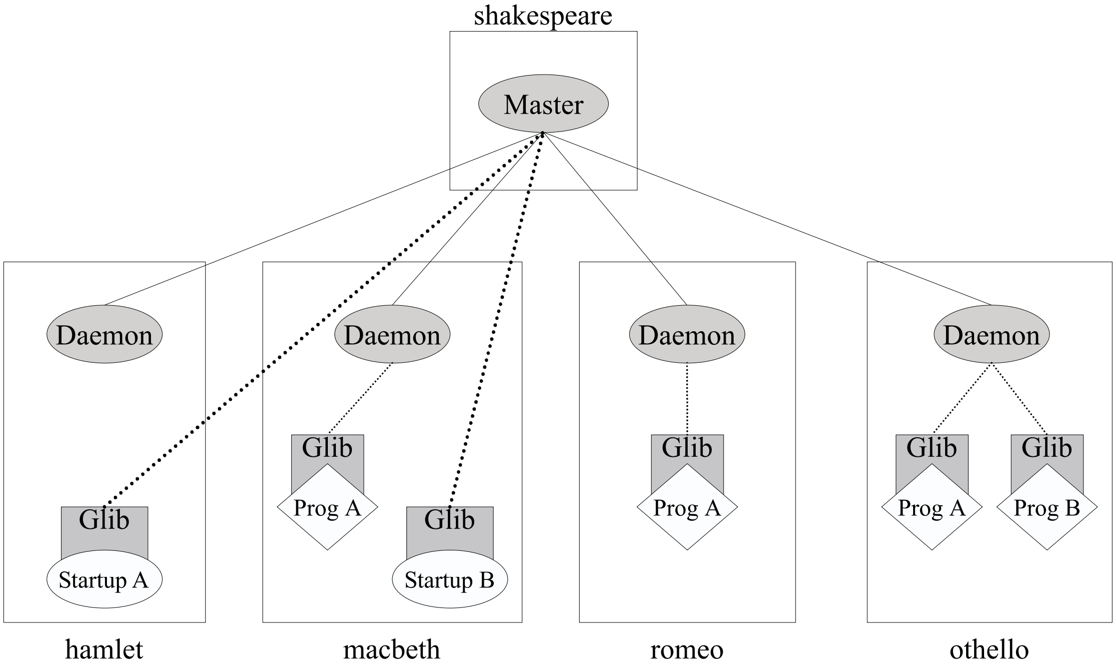

Figure 1: The basic structure of a GLUnix cluster. In this example, the master is located on the machine names `shakespeare` and daemons are located on `hamlet`, `macbeth`, `romeo`, and `othello`. A user on `hamlet` has executed job A, a parallel job consisting of three processes located on `macbeth`, `romeo`, and `othello`. A second user has executed a sequential job B which GLUnix has placed on `othello`.

4.2.1 Components

GLUnix consists of three components: a per-cluster master, a per-node daemon, and a per-application library (see Figure 1). The GLUnix master is primarily responsible for storing and managing cluster state (e.g., load information for each node and process information) and for making centralized resource allocation decisions (e.g., execution-time job placement and coscheduling). The GLUnix daemon running on each node performs information collection duties for the master, checking local load information every few seconds and notifying the master of changes. The daemon also serves as a local proxy for the master in starting and signaling jobs and in detecting job termination. The GLUnix library implements the GLUnix API, providing a way for applications to explicitly request GLUnix services. We support execution of unmodified applications by allowing a separate stub process to request execution of an unmodified binary image on a cluster node.

Sequential and parallel jobs are submitted to GLUnix by a startup process which is linked to the GLUnix library. The startup process issues the execution request to the GLUnix master and then acts as a local proxy for the user's job, intercepting job control signals and forwarding them to the master and receiving I/O from the remote program and displaying it for the user.

4.2.2 Internal structure

Figure 2 depicts the logical internal structure of the master, the daemons, and the GLUnix library. Table 3 summarizes the primary function of each module. The `Comm` module is currently implemented using Berkeley Stream Sockets over TCP/IP, though it is designed to be easily replaced by a module using faster communication primitives, such as Active Messages [32].

GLUnix events consist of both messages and signals, with the main event loop located in the `Comm` module. When a message arrives, the `Msg` module unpacks the message into a local data structure and then invokes the message handler. Outgoing messages are also packaged up by the `Msg` module and then transmitted by the `Comm` module. When a signal event arrives, the event loop invokes the appropriate handler for the event. Each module registers handlers for the events they define.

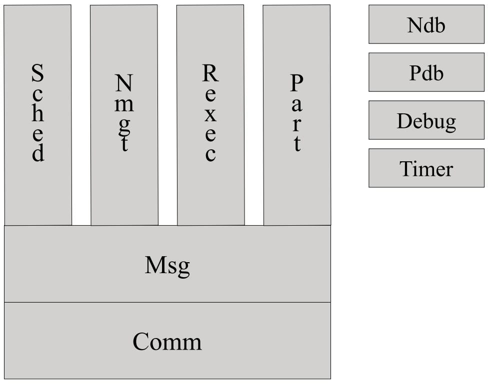

Figure 2: The logical internal structure of the GLUnix master, daemon, and library. Although the module structure is the same in all three components, most modules have different code for each.

Comm	Abstracts away network communication; contains the main event loop of GLUnix
Msg	Performs message packing and unpacking, invokes handlers for incoming message
Ndb	node database which stores and manages state relating to the individual nodes in the system
Pdb	process database which keeps track of the currently executing GLUnix processes
Sched	implements coscheduling policy algorithm
Nmgt	performs node management functions: collecting load data, detecting failed nodes
Rexec	performs remote execution
Part	manages GLUnix partitions and reservations
Debug	defines multiple debug levels, supports dynamic selection of debug levels
Timer	schedules periodic events

Table 3: Descriptions of the function of the GLUnix modules.

4.2.3 Faults

GLUnix is currently able to tolerate failures of the GLUnix daemon and startup processes. These failures are detected by the GLUnix master when its network connections fail. During normal operation, the GLUnix master maintains an open connection to each GLUnix daemon and startup process. When the master identifies a broken connection to a daemon, it performs a lookup in its process database for any sequential GLUnix processes which were executing on that node, marks them as killed, and notifies the appropriate startup processes. If any parallel GLUnix programs had one of their processes on that node, GLUnix additionally sends a `SIGKILL` to the other processes of those programs. The master then removes the failed node from its node database and continues normal operation.

Similarly, when the master detects that a startup process has died in response to a node crash or a `SIGKILL`, the master sends a `SIGKILL` to the remote program associated with that startup and removes the process from its process database. Alternatively, the system need not kill the entire parallel program whenever a single process in it dies. The master could notify the remaining processes of the failure, using a mechanism such as Scheduler Activations [1], and continue normal operation.

Currently the system does not provide any support for checkpointing or restarting applications, though applications are free to use checkpointing packages such as Libckpt [27] if necessary.

4.3 Transparent Remote Execution

When users run GLUnix programs from the command-line, the executed process becomes the GLUnix startup process. The startup process must be linked to the GLUnix library and invokes a GLUnix library routine to send a remote execution request to the master. The GLUnix library sends to the master a request containing the command-line arguments, the parallel degree of the program, and the user's current environment including current group, current working directory, umask, and environment variables. When the master receives this request, it uses load information periodically sent from the daemons to select the N least loaded nodes in the system. Once the nodes have been selected, the `Rexec` module of the master registers the new process with the `Pdb` module and then sends the execution request to each of the `Rexec` modules of the selected daemons. Upon receipt of the execution request, the GLUnix daemon's `Rexec` module performs a `fork()`, recreates the environment packaged up by the startup, and then does an `exec()` of the program. The daemon will then report success or failure to the master, providing an error code in the latter case which is passed back to the startup process.

To maintain signaling semantics for remote jobs, the startup process, on initialization, registers handlers for all catchable signals. When the startup catches a signal, sent to it either by the `tty` in response to a `Ctrl-C` or `Ctrl-Z` or via the `kill()` system call, the startup process sends a signal request to the master. The master forwards the signal request to the daemons which are managing the processes of this parallel program and those daemons then signal the appropriate local processes using the `kill()` system call.

When running GLUnix jobs, keystrokes are sent to the remote job and output produced by local processes is multiplexed onto the user's console. A runtime option allows prepending of the VNN or hostname of the process generating a given line of output. GLUnix commands can be piped together with `stdout` and `stderr` semantics preserved. In the case of parallel programs, `stdin` is replicated to all processes, while `stdout` and `stderr` from all processes are properly displayed on the user's console.

GLUnix is able to maintain most, but not all, of the standard UNIX I/O semantics. UNIX provides two distinct output streams, `stdout` and `stderr`. While `stderr` data is usually directed to a user's console, `stdout` can be easily redirected to a file or to another program using standard shell redirection facilities. Many remote execution facilities such as `rsh` and PVM [29] do not keep these two I/O streams separate and thus do not maintain proper redirection and piping semantics. GLUnix maintains two separate output streams.

The I/O semantics that GLUnix does not properly handle concern `tty` behavior. When executing a process remotely, GLUnix installs a pseudo-terminal and places the user's local terminal in `RAW` mode. This is the standard approach used by `rsh`, `rlogin`, and other remote execution and login facilities. However, in order to keep `stdout` and `stderr` separate, GLUnix must set up a separate pseudo-terminal for each. This violates standard terminal semantics. If both `stdout` and `stderr` are tied to terminals, they are assumed to be the same terminal. Programs such as `more` and `less` often rely on this fact. Additionally, this scheme uses 2 pseudo-terminals for each GLUnix program; since pseudo-terminals are a limited resource in modern UNIX operating systems, this restricts the number of GLUnix jobs that can be run on each machine.

4.4 Barriers and Coscheduling

The GLUnix barrier is implemented using a standard binary-tree algorithm. The `GLUnix_Barrier()` library call sends a barrier request message to the local daemon. That daemon identifies the other daemons which re managing processes for this parallel program. The daemons then use a standard binary tree reduction algorithm to identify when all processes of the parallel program have entered the barrier. When the daemon at the root of the barrier tree detects this, the release message is broadcast back down the daemon tree, with each daemon sending a reply message to its local process, indicating that the barrier has completed. At this point the `GLUnix_Barrier()` library call will return and the process will continue executing.

GLUnix implements an approximation of coscheduling through a simple user-level strategy. The GLUnix master uses a matrix algorithm [26] to determine a time-slice order for all runnable parallel programs. Periodically, the master sends a scheduling message to all daemons instructing them to run a new program, identified by NPID. Each daemon maintains a NPID to local `pid` mapping in its process database. The daemon uses this mapping to send a UNIX `SIGSTOP` signal[2] to the currently running parallel process, and a `SIGCONT` signal to the newly runnable process. We chose this technique to avoid kernel modifications, maintaining an entirely user-level solution.

This technique has two limitations: it is only an approximation and it is not transparent. By sending a `SIGSTOP` to those GLUnix processes which should not currently be running, GLUnix reduces the set of runnable processes which the local operating system will choose from. The current implementation of GLUnix, however, does not stop all other processes in the system, including system daemons or other non-GLUnix jobs. Thus, there may be other non-GLUnix jobs which will run during a parallel program's time slice. The reason for this decision is that if GLUnix were to crash, stopped jobs would never be restarted.

Second, this technique is not transparent to some programs. The delivery of the `SIGCONT` signal will interrupt any system call which the process may have been blocked in, returning a error code `EINTR` to that process. While this is the same behav-

[2] UNIX processes cannot catch `SIGSTOP`. Thus, there are no worries about malicious programmers ignoring scheduling directives.

ior that a `Ctrl-Z/fg` combination would have from a shell, we have encountered a small number of programs which do not properly check for this condition and thus do not execute correctly when scheduled using this method. In particular, system daemons, which do not typically have to deal with job control, may not operate correctly.

4.5 Software Engineering Experience

In order to make the code as robust as possible, the internal GLUnix modules were initially written to mask internal faults and to continue in the face of bugs whenever possible. This technique actually had the opposite effect. Rather than making the system more stable, masking faults in this way separated the point of fault origin from the point of failure, making the bugs more difficult to locate and fix. The book "Writing Solid Code" [23] led us to a adopt a less tolerant internal structure. System debugging became significantly easier and proceeded more quickly. Making the inter-component interactions less tolerant of each other's failures would also probably make the system easier to debug, but would also make it less stable in the short term.

We found that many of the problems encountered during development only manifest themselves when run on large clusters. For example, problems with our I/O system did not exhibit themselves until we tested GLUnix on a cluster of 85 nodes, at which point the default file descriptor limit of 256 was exhausted (recall that each process opens 3 file descriptors to the startup process for piping of `stdin`, `stdout`, and `stderr`). Initially GLUnix development took place on a dedicated cluster of 8 nodes and ran on a production cluster of 16 nodes. Running GLUnix on a larger cluster of 100 nodes uncovered a number of interesting performance limitations and implementation bugs. Some of these issues are described in more detail in Section 6.2.

4.6 Implementation Status

GLUnix has been in daily use for nearly two years and has had 130 users run over 1 million jobs in that time period. User experience with the GLUnix has been largely positive, and the system has enabled or assisted systems research and program development in a number of areas. Implementations of the parallel programming language Split-C [13] and the Message Passing Interface (MPI) [30] use the GLUnix library routines to run on the Berkeley NOW. A number of interesting parallel programs have been implemented using these facilities, including: NOW-Sort [4], which holds the record for the world's fastest sort time: 8.4 GB in under one minute, p-murphi [14], a parallel version of the popular protocol verification tool, and a 32K by 32K LINPACK implementation that achieved 10.1 GFlops on 100 UltraSPARC-I processors, placing the Berkeley NOW 345th on the list of the world's 500 fastest supercomputers[3].

5 GLUnix Usage

GLUnix has been running and in daily use by the Berkeley CS Department since approximately September 1995. While initially running on 35 SparcStation 10's and 20's, it has been running daily on 100 UltraSparcs since roughly June of 1996. As of June, 1997, GLUnix has had 130 users over the course of its lifetime and has run over one million jobs. GLUnix has been in daily use by both production users as well as system developers. It has been used by the graduate Parallel Architecture class

[3] See `http://netlib.cs.utk.edu/benchmark/top500.html`.

here at Berkeley, by many other research departments here at Berkeley, and by others not affiliated with the Berkeley NOW project. GLUnix is an entirely user-level solution, and has run on SparcStation 10's, 20's and UltraSparcs running Solaris versions 2.3–2.6. A basic port to Linux (kernel version 1.2) on x86's has also been completed.

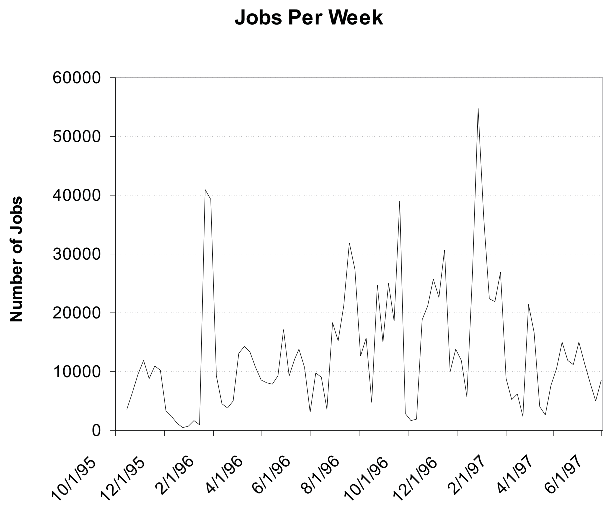

Figure 3: Jobs run per week from October, 1995 through June, 1997.

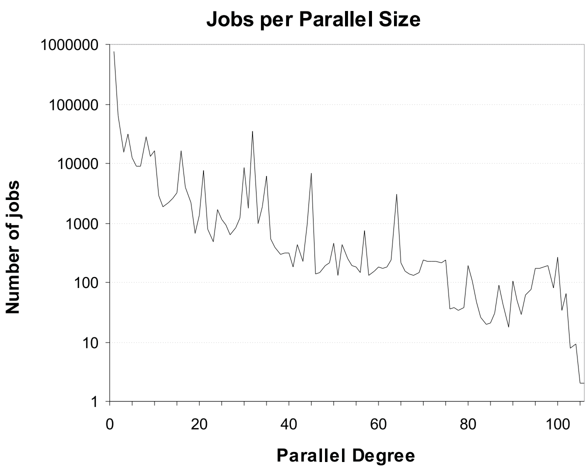

Figure 4: Histogram of the parallel degree of the jobs run under GLUnix.

The largest use of GLUnix and the Berkeley NOW cluster has been as a compute server for distilling thumbnail images off of Usenet[4]. The second most common use of the cluster has been as a parallel compilation server via the `glumake` utility. The cluster has been used to achieve a world record in disk-to-disk sorting [4], for simulations of advanced log structured file sys-

[4]See `http://http.cs.berkeley.edu/~eanders/pictures/index.html`

tem research [24], two-dimensional particle-in-cell plasma, three-dimensional fast Fourier transforms, and genetic inference algorithms.

GLUnix was also found to be extremeley useful for testing and system administration. Using GLUnix for interactive parallel stress testing of the Myrinet network revealed bugs which could not be recreated using traditional UNIX functionality such as `rsh`. With respect to system administration, a large fraction of the parallel job requests under GLUnix have been for programs such as `ls`, `rm`, and `who`, which are not traditional parallel programs. System administrators and developers use GLUnix to run these programs simultaneously on all nodes in the cluster to carry out various administrative tasks.

The GLUnix batch facility (see Section 8.2) has been used in situations where very large numbers of short and medium simulations were run, for example, the simulations for Implicit Coscheduling research [16].

6 Performance and Scalability

This section evaluates the scalability and performance of the system, relating some of our experiences in performance tuning. All of the data presented in this section is measured with GLUnix running on 167MHz Sun UltraSparcs running Solaris 2.5. Although the cluster has a 160 Mb/s Myrinet network which is used by parallel programs running under GLUnix, our networking software has only recently supported a client/server model and GLUnix has not yet been ported to use the fast network.

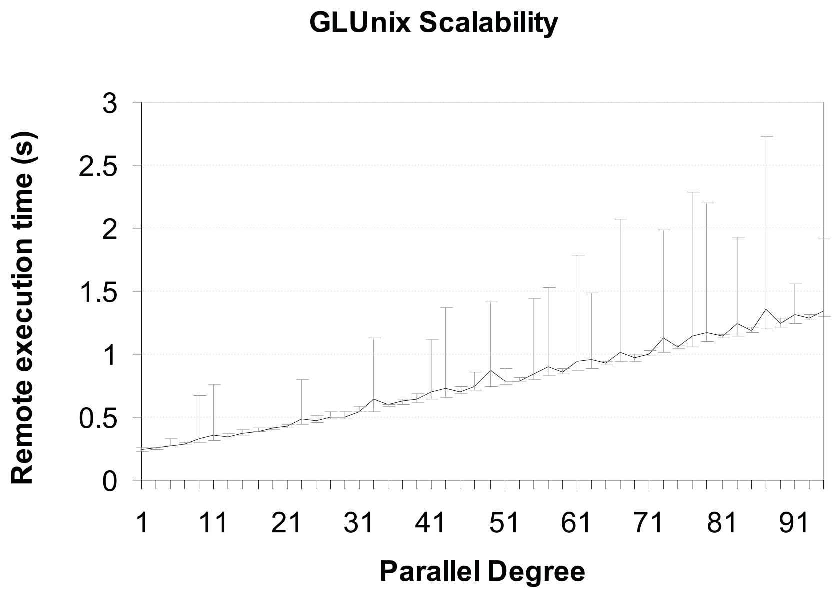

Figure 5: The time taken to run parallel programs of various sizes. Each data point is averaged over 20 runs. The error bars indicate the minimum and maximum time of those 20 runs.

6.1 Scalability

One of the main concerns of a centralized master architecture is that the master will become a bottleneck. Figure 5 demonstrates the scalability of the centralized master in running parallel jobs. This figure shows the remote execution time for jobs of varying degrees of parallelism. The time for running a sequential job under GLUnix is equivalent to a parallel program of degree 1. The data in Figure 5 were generated by using `tcsh`'s built-in `time` command for 20 consecutive runs of a null program for each parallel degree. Each run of the same parallel degree was sent to the same set of nodes to eliminate file cache effects. GLUnix can run a 95-node parallel program in 1.3 seconds. For comparison, an `rsh` of a `null` application to a single node under similar conditions takes roughly 4 seconds.

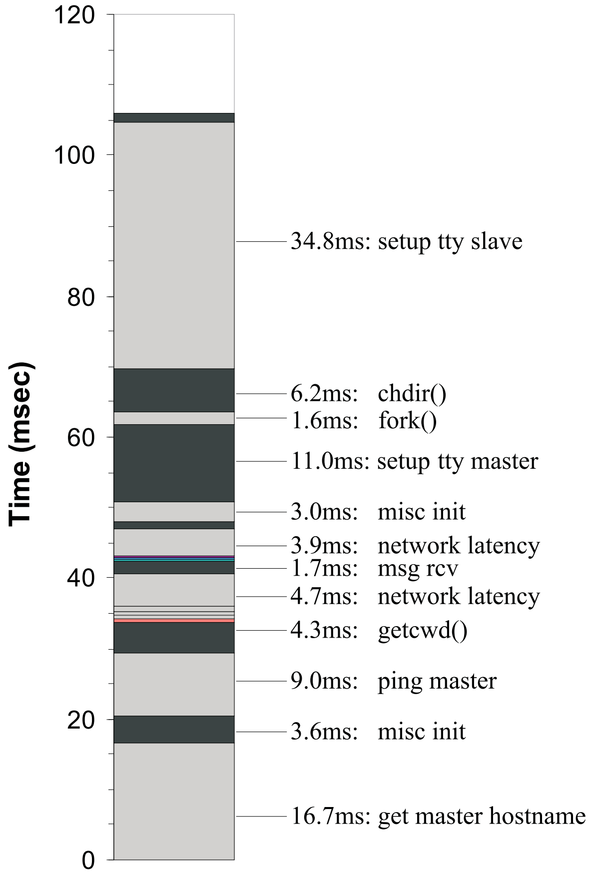

Figure 6: An analysis of the remote execution time of GLUnix. This breakdown is for a sequential program. Numbers are averaged over 50 runs. The total time depicted is 106.0 msec. Only the components which take 1.0 msec or more are labeled. The unlabeled segments amount to 3.8 msec, or 3.6%, of the time. The time spent in the master on non-network related activities is 0.4 msec. The final `exec()` call performed by the daemon is not depicted in this figure.

Figure 6 breaks down the remote execution time for a sequential job into its components. This breakdown was generated by inserting `gettimeofday()` calls at strategic points in the code. The overhead of calling `gettimeofday()` is approximately 1 μsec, causing negligible perturbation of the system's performance. The "get master hostname" segment represents reading the master's hostname from a NFS-mounted filesystem. The "ping master" segment wasn't part of the original GLUnix design, but was added later in response to user feedback; it works as follows: when a program first initializes the GLUnix library, the library sends a ping message to the master to see if GLUnix is running. Without this feature a program using the GLUnix library would not be informed that GLUnix was down until some subsequent library call. The ping's cost includes the setup of a TCP connection between the startup process and the master as well as the exchange of GLUnix-internal connection information. The tty-related costs are only incurred for sequential jobs—GLUnix does not provide tty emulation for parallel jobs. Snooping the ethernet revealed that the `getcwd()` library call from our NFS-mounted test directory was generating two NFS RPCs to the file server. Figure 6 shows that very little of a program's latency is due to the master. The dominant factor in remote execution latency is the tty support which accounts for 45.8 msec, or 43.1%, of the total time.

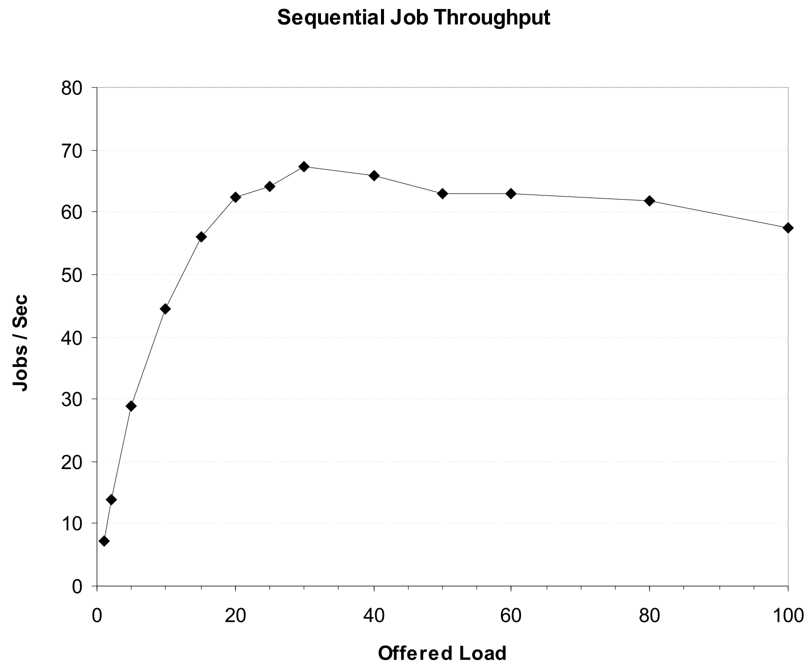

Figure 7: This graph measures the throughput of GLUnix for sequential jobs as the offered load changes. The value along the x-axis represents the number of nodes making requests. Each node ran a single remote execution request in an endless loop. The CPU utilization of the master never exceeded 53.1%.

Figure 7 further evaluates the scalability of the centralized master by graphing the throughput of the system. Although GLUnix throughput peaks at 67.3 jobs/second, at this load the CPU utilization of the master was ony 50.8%. This indicates that the throughput bottleneck is not the computation performed by the master.

Figure 8 quantifies the coscheduling skew for parallel programs. The coscheduling skew is the difference in time between when the first and last processes of a parallel jobs are sent a scheduling signal. The skew was measured by taking a timestamp in each daemon just before sending the `SIGSTOP` or `SIGCONT` signals to a parallel application. These measurements presume

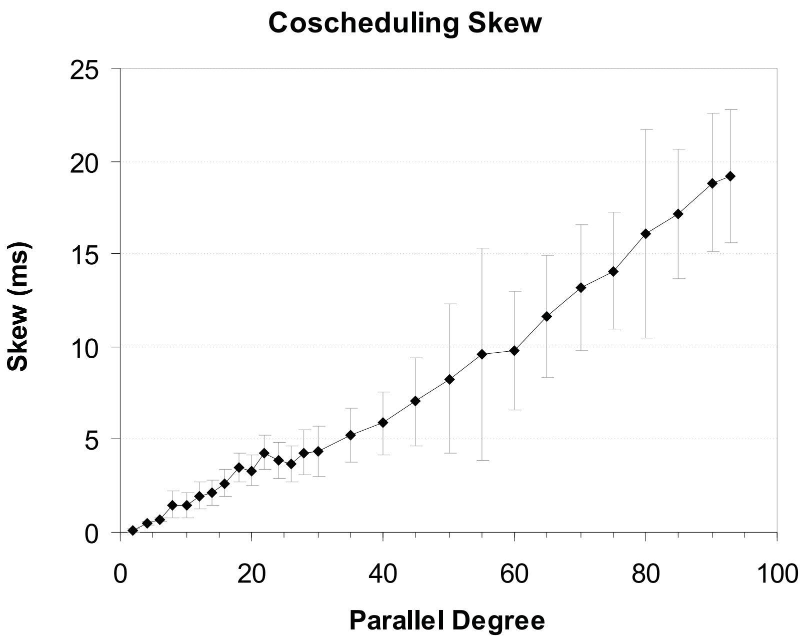

Figure 8: This graph measures the skew arising from coscheduling. The skew is defined as the difference between the earliest and latest times that two processes in a parallel program are started or stopped. Each data point represents the skew for a particular parallel degree averaged over 200 coscheduling events. The error bars indicate one standard deviation above and below the average.

that the signal delivery and rescheduling times through the local operating system are statistically equivalent for different parallel degrees, a reasonable assumption given that each operating system performs the actions entirely independently. The coscheduling time slice used for these tests was one second. We believe the wide variance seen is due to network contention of the ethernet.

The data presented in Figure 8 reveals two things. First, network and file system costs account for a significant portion of the remote execution time. Porting GLUnix to use Active Messages (which provides a very low-latency RPC) and a cluster file system like xFS [3] (which provides improved caching over NFS) would improve remote execution time significantly. Second, the master is not a central bottleneck. Only 0.7 msec of the remote execution latency (less than 1%) is spent in the master. This point is further demonstrated by the throughput tests described above which demonstrate that the processing required by a centralized master will not saturate modern CPUs for our target 100 node cluster. This result implies that a fully peer-to-peer system architecture is not necessarily warranted for performance reasons.

6.2 Performance Tuning

The system performance described above reflects the current state of our implementation. However, the system required significant evolution, redesign and performance to achieve the current performance levels. In this section, we will describe some of our experience with improving the performance of remote execution. These changes were driven by the needs of the parallel NOW-Sort application [4], which currently holds the record for the fastest disk-to-disk sort. To set the record, NOW-Sort had to sort as much data as possible in 60 seconds. The sort was structured as a parallel program, with each node sorting a portion of a large datafile. NOW-Sort was the first program that required fast startup times on more than 32 nodes. Unfortunately, initial runs revealed that it took more than 60 seconds simply to start the program on 100 nodes.

Some system experimentation revealed that GLUnix was flooding the local NIS server for large program startup. When executing a program on a remote node, the GLUnix daemon must initialize the UNIX group membership for the remote program. Initially this was done using the `initgroups()` command. However, this command accesses the cluster's NIS server to retrieve the requested group information. Running a parallel program on 100 nodes thus floods the NIS server with 100 network connection requests at roughly the same time. It was found that the first few requests were processed quickly, but that others were quite slow, taking nearly one second each to complete. Further investigation revealed that the `getgroups()/setgroups()` system calls provide similar functionality without the NIS lookup.

After making the above change, we were quite surprised to discover that remote execution performance was actually worse. Additional measurements indicated that establishment of the I/O connections to the GLUnix startup process had become slower. Each process in the parallel program was making three I/O connections to the startup process. Unfortunately, the default connection queue length in Solaris 2.5 defaults to 32. Thus, the first connections completed quickly, but when the queue was overrun, the connection requests had to timeout and retry. Optimized for the wide area, the TCP connection retries proceeded very slowly once the queue was overrun. To resolve this, we used a Solaris administration command to set the TCP driver's accept queue length to its maximum value of 1024. This improved performance dramatically. We further concluded that the NIS requests staggered the execution of the GLUnix daemons, effectively serializing the I/O connections to the startup and eliminating the overrun of the accept queue.

6.3 Limitations on Scalability

In order to pipe the input and output between the remote program and the user's shell, GLUnix sets up TCP stream connections between each remote process of the parallel and the startup process. In order to properly keep `stdout` and `stderr` separated so that shell piping will work as expected, GLUnix requires two connections. While it is possible to multiplex `stdin` onto one of the two output connections, our initial implementation uses a third connection for simplicity. Consequently, the startup process will have three sockets open to each of the processes in the parallel program. These I/O connections are currently a limiting factor for parallel programs in terms of both the startup time and the maximum program size.

Because GLUnix uses three socket connections to the startup for each process in a parallel program, running a 100-way parallel program would require 300 I/O connections to the GLUnix startup process. Microbenchmarks reveal that our UltraSparc machines are capable of accepting roughly 500 connection establishments per second, making I/O connection establishment a dominant cost for large parallel program startup. Furthermore, In Solaris 2.5, the number of file descriptors for a single process is limited to 1024. Consequently, the current implementation of GLUnix can only support parallel programs which have 341 processes or fewer[5].

[5] $\lfloor 1024/3 \rfloor = 341$

7 User-Level Challenges and Tradeoffs

The decision to implement GLUnix at the user-level had a major impact on the ability of GLUnix to transparently execute programs remotely. This section discusses difficulties in properly handling terminal I/O, signals, and location independence. All of these user level challenges relate to the fact that there are some program/kernel interactions which cannot be efficiently intercepted from the user level in a portable manner. Although pseudo-devices such as the `/proc` filesystem of Solaris can transparently intercept system calls and redirect them to a user-level process, it has two disadvantages. First, it is not generally available in other operating systems. Second, it limits performance since it requires a context switch for every operation. For example, to intercept terminal I/O using `/proc`, the GLUnix daemon would have to intercept *all* `read()` and `write()` system calls to any file descriptor, impacting the performance of all application I/O, not just terminal I/O.

7.1 Lost Input

Standard UNIX semantics for I/O dictate that, when a job reads from UNIX `stdin`, it consumes only as many input characters as requested; excess characters should remain in system buffers to be read by the next operation. This standard UNIX feature manifests itself, for example, by allowing users to begin typing subsequent shell commands before the first has completed. Transparent remote execution requires that this behavior be the same for remote jobs as well. However, this behavior cannot be implemented for existing application binaries using only user-level services. Implementing this behavior would require the GLUnix startup to know how many characters of input were being requested by the program. While this can be easily accomplished if the remote program has been linked with a GLUnix library, this solution is not compatible with existing application binaries. Consequently, the GLUnix startup process forwards all available input characters to remote processes as they become available. If those characters are not all consumed by the remote program, they are lost. This policy means that users cannot type ahead while a GLUnix command is executing, compromising the transparency of remote execution.

Solaris's `/proc` filesystem can provide the necessary functionality, but at a high cost. Intercepting each `read()` system call to ascertain the file descriptor and length would not only decrease the performance of `stdin` but would also slow down other I/O operations, including file accesses, pipes, etc. Even if the performance of `/proc` were improved, intercepting the `read()` system call and sending an input request back to the startup process would increase the latency of terminal `read()` operations significantly.

The ideal solution to this problem would be to optimistically send the input to remote nodes and then to be able to recover the unused portions from the network and operating system buffers so that the input could be returned to the original machine. Any unused input could then be reinserted into the appropriate streams to be consumed by future `read()` system calls. Today's UNIX operating systems do not provide a way to do this, nor would this technique necessarily be very efficient.

7.2 Signals

GLUnix currently uses the startup process as a proxy to catch signals and forward them to remote children. However, in UNIX, a process cannot determine if a caught signal was destined only for itself or for its process group. Consequently, GLUnix cannot maintain process group signaling semantics for remote processes. Solaris's `/proc` could be used to ascertain this information in certain cases by catching `kill()` system calls from all processes and checking for a process group identifier. However, this method cannot be used on the kernel or device drivers (in particular the terminal device driver which is used for job control signaling from a shell).

7.3 Location Independence

GLUnix cannot fully present a location independent execution environment using standard UNIX operating system facilities. When an application issues system calls to determine the machine's hostname or establish network connections, those system calls are carried out on the remote node. This compromises the transparency of remote execution because remote applications are executing in a different environment than the user's home node and may behave differently. Again, Solaris's `/proc` filesystem may be used to forward system calls back to the home node, this solution does not work in general. Specifically, it cannot handle the case of `ioctl()` system calls. The `ioctl()` system call in UNIX does not have a fixed invocation specification – it can take a variable number of arguments, the format of which are specified by the device being accessed. Consequently, in general a system using `/proc` would not know what data to transfer back to the home node when issuing the `ioctl()` system call. Although a system may attempt to handle cases for well-known devices such as terminals, this approach would require modifying the system each time a new device is added.

One possible solution to this problem is to send the `ioctl()` system call to the process's home node (where any special devices are located). The system can then issue the system call and ascertain the portions of a process's address space needed to execute the ioctl. Requests to read the necessary portions of the address space could then be sent back to the executing node, when the necessary data is available, the `ioctl` can once again be issued with the home kernel supplying any necessary process address space values. Clearly, such functionality is not supported by either standard user-level services or Solaris's `/proc` API.

7.4 NFS

The weak cache consistency policy of NFS [34] for file attributes limits the utility of the `glumake` utility. To improve performance, NFS clients cache recently accessed file attributes. Elements in this cache are flushed after a timeout period, eliminating most stale accesses in traditional computing environments. However, the `glumake` tool encounters problems when compiling files on remote nodes. In this case, the file attributes are properly updated on the node where compilation takes place, but the old attributes remain cached on other nodes. When `glumake` checks file modification times to decide which objects need to be rebuilt, it often incorrectly concludes that all objects are up to date because of these stale cached values. This problem often manifests itself when glumake compiles source files into object files, but fails to relink the executable from the newly created object files. Table 4 shows a sample Makefile that exhibits this problem. The main manifestation of this problem in our environment is when a developer modifies a single source file, recompiles, and runs the old executable, wondering where the changes went. Modifying the `Makefile` to add dependencies between the source files and the executable can resolve this problem for a given Makefile. However, requiring users to modify Makefiles to avoid consistency problems in the filesystem is neither transparent nor reasonable. This argues for a strongly-consistent cluster file system [19, 3].

```
file:  file.o
       cc -o file file.o

.o.c:
       cc -c $^
```

Table 4: Sample Makefile which exposes inconsistency of NFS when run under `glumake`.

8 Social Considerations

Over the past few years we have learned a number of things from supporting an active user community. This section summarizes a few of the GLUnix features that users have found most useful and then relates our experiences regarding user-initiated system restart and the need for batch and reservation facilities.

One of the elements of feedback that we consistently hear from users is that the ability to use standard UNIX job control from a shell (e.g., `Ctrl-C` and `Ctrl-Z`) to control parallel programs is extremely useful. Another useful feature is the ability to run standard UNIX tools on all nodes in the system, facilitating a number of system administration tasks, such as installing a new device driver on all nodes or verifying configuration files. Since input is replicated to all processes in a parallel program, commands need to be typed only once. Although there are other tools such as `rdist` which can perform these operations, being able to use standard UNIX commands on remote nodes is both natural and convenient.

8.1 User-Initiated System Restart

As the system user community increased, it became increasingly important to have GLUnix continuously available. Periodically, the system would fail, either as a result of system bugs or machine crashes. Initially, only system developers had the authority to restart the system (perhaps after inspecting the status of the system for any bugs). However, since GLUnix developers could not be continuously available, we decided to allow ordinary users should be permitted to restart the system.

The decision to allow restarts by the general population lead to an interesting, yet subtle consequence. Users can mistakenly restart the system when nothing is wrong with it. With the introduction of the reservation system, GLUnix would prevent users from running jobs on machines which were reserved by other users. Many users mistakenly interpreted this to mean that those machines had crashed, and so they restarted the system. Additionally, as the user community continues to grow, many new users are added which are less and less familiar with our infrastructure. These users often encounter errors in other experimental aspects of the system (such as with the AM-II network) and, being unsure of the source of the error, restart GLUnix as a reflex. These two factors have led to an explosion of entirely unnecessary GLUnix restarts.

8.2 Batch Jobs

Although the original goal of GLUnix focussed on providing support for interactive jobs, the lack of a batch facility for the system proved to be a problem. A number of GLUnix users were involved in research which required a very large number of simulations to be run with different parameters. With the original GLUnix tools, the only supported options were to submit all the jobs at once or to submit the jobs one at a time, waiting for the previous job to complete before submitting the next. The first option would flood the system, making it unusable to others, while the second made it difficult to take advantage of the parallelism in the system.

To address this need, we implemented a standard queuing batch system. Users could submit an arbitrary number of jobs to the batch system. The batch system monitors the load of the GLUnix cluster to determine how many machines are idle. The batch system then regulates the submission of the batch jobs to GLUnix, being careful not to overrun the cluster. The batch system also has an option which will cause it to terminate submitted batch jobs if the load of the cluster exceeds a certain threshold; this allows low priority "background" jobs to use all available idle resources without significantly impacting the availability of the system for other users.

8.3 GLUnix Reservations

When GLUnix was initially deployed, it had no support for partitions or reservations. However, as the system became more heavily used, the need for a reservation system became more apparent. System developers as well as researchers often need exclusive access to a particular set of machines to take performance measurements. Initially, these reservations were simply made by sending e-mail to a user distribution list. However, this approach rapidly became untenable. As the number of people making reservations increased, it became increasingly difficult to identify the set of machines available for general purpose use. To illustrate the magnitude of the problem, at one point it required an exchange of roughly 40 e-mail messages over the course of two days to establish a reservation for 32 nodes.

In response to this, we developed a partitioning and reservation system for GLUnix. The partitioning and reservation system works by associating three lists with each machine: aliases, users, and owners. The alias list simply designates the list of names identifying each machine; the user list identifies the users who currently have permission to run applications on that machine; and the owner list specifies those individuals who have permission to reserve that machine. When running an application, a user can provide an alias[6] to indicate to GLUnix which nodes should be used to run the program. GLUnix then filters that set of machines by the user list associated with each machine, ensuring that the application will run only on those node for which the user has permission. To make a reservation, a user supplies a description of the nodes to reserve, the time of the reservation, and a name for the reservation. When the time for the reservation arrives, GLUnix filters the set of requested nodes by each machine's owner list, preventing users from reserving machines for which they do not have permission. GLUnix then assigns the reservation name as an alias to each machine selected. This allows the user to run applications on just the reserved nodes, or other unreserved nodes as well.

In addition to dramatically reducing the problems described above, the ability to associate a set of aliases with a group of machines and then to use that name to identify them has proved to be extremely useful. At many points in the project we had rapidly changing cluster configurations as different versions of networking software were installed and disks and memory rearranged. In addition, we have different architectures: SparcStation 10's, 20's and UltraSparcs, in both single processor and multiprocessor configurations. By assigning aliases to machines, users can refer to a set of machines by some named

[6]The node specification is actually an equation consisting of aliases and set operations on those aliases.

characteristic rather than having to keep up with the current configuration. Whenever the cluster configuration is modified, the aliases are updated, allowing subsequent uses of the alias to properly use the new configuration.

8.4 Desktop GLUnix systems

A number of the UltraSparcs in our cluster are located on desktops and are used as personal workstations. It was found early on that the users of these machines were extremely unhappy about demanding jobs being placed on those machine by GLUnix while they were working. Even if GLUnix were to prevent jobs from being run on desktop machines while users were present, many GLUnix jobs are long-running simulations which may still be running on the machine when the workstation owner returns. The lack of migration in GLUnix led us to a situation where desktop machines protected via the reservation system from running any GLUnix jobs except the user's own. One of the original motivating studies for GLUnix demonstrated that a desktop workstations were idle during a significant fraction of the day, but without migration, GLUnix is unable to effectively harvest that idle time without impacting workstation owners.

9 Related Work

When we began design of GLUnix in 1993, a number of systems provided partial implementations for our desired functionality. While the implementation of such systems has naturally matured over the years, to our knowledge, no single system supports all of our desired functionality for a NOW operating system. In this section we briefly list some of the projects with goals similar to our own. LSF [37] provides command line integration with a load balancing and queuing facility. Both V [11] and Sprite [15] support process migration in the kernel, while Condor [8] provides a user-level, though not fully transparent, solution. PVM [29] and Linda [9] provide support for parallel programs on clusters of distributed workstations, and PVM has only recently begun to support coscheduling [26]. More recently, the Hive [10] effort sought to provide fault containment clusters of SMPs, and the Solaris MC [20] group modified the Solaris kernel to provide a single system image to sequential and multithreaded applications.

10 Future Work

There are two further research efforts that have arisen out of GLUnix. The first is an effort called Implicit Coscheduling [16] which is examining the effect of enabling communicating processes to coschedule themselves using various heuristics. The algorithm uses adaptive two-phase blocking. When a process would normally block, waiting for a communication event, it instead spins for an adaptive period of time. If the message arrives during the spin time, the process continues to run, otherwise a new process is scheduled to run. In this way, the constituent processes of a parallel program dynamically adjust themselves to run at approximately the same time despite independent local operating system schedulers. Experimentation reveals that implicit coscheduling performs within +/-35% of coscheduling without the need for global coordination.

The second research effort is SLIC [18] which is a method for alleviating the restrictions of implementing system facilities at the user-level. By loading trusted extensions into a commodity kernel, system software developers can eliminating many of the limitations discussed in section 7 while retaining the majority of advantages that user-level software development affords.

11 Conclusions

The GLUnix operating system was designed and built to evaluate techniques for building portable, user-level system software for networks of workstations. The system was designed to provide transparent remote execution, dynamic load balancing through migration, and support for interactive sequential and parallel programs. Though the system has supported both the production and research work of over 130 users over 2 years, it did not succeed in meeting all of its goals. A number of limitations prevented us from providing transparent access to cluster resources in a completely user-level implementation. To address the limitations identified by our user-level approach, we are currently building SLIC, a small kernel toolkit which will securely export the necessary system abstractions to the user level. Although initially designed to support a new version of GLUnix, SLIC will enable general purpose extension of commercial operating systems.

12 Acknowledgments

There are a great many people would deserve credit for the production and maintenance of GLUnix. In addition to the authors of this paper, Keith Vetter and Remzi Arpaci-Dusseau have contributed portions of code to the GLUnix system. Special thanks go to Remzi Arpaci-Dusseau for his help in tuning GLUnix performance and to the entire NOW group at Berkeley for their rapid identification and, often times, diagnosis of system instabilities. Thanks also go to Joshua Coates for assisting in performance evaluation.

References

[1] Thomas Anderson, Brian Bershad, Edward Lazowska, and Henry Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *ACM Transactions on Computer Systems*, pages 53–79, February 1992.

[2] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW Team. A Case for NOW (Networks of Workstations). *IEEE Micro*, February 1995.

[3] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 109–126, December 1995.

[4] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau and David E. Culler and Joseph M. Hellerstein and David A. Patterson. High-Performance Sorting on Networks of Workstations. In *SIGMOD '97*, May 1997.

[5] Remzi Arpaci, Andrea Dusseau, Amin Vahdat, Lok Liu, Thomas Anderson, and David Patterson. The Interaction of Parallel and Sequential Workload on a Network of Workstations. In *Proceedings of Performance/Sigmetrics*, May 1995.

[6] Remzi Arpaci, Andrea Dusseau, Amin Vahdat, Lok Liu, Thomas Anderson, and David Patterson. The Interaction of Parallel and Sequential Workload on a Network of Workstations. Technical Report CSD-94-838, U.C. Berkeley, October 1994.

[7] Remzi H. Arpaci, David E. Culler, Arvind Krishnamurthy, Steve Steinberg, and Kathy Yelick. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.

[8] Allan Bricker, Michael Litzkow, and Miron Livny. Condor Technical Summary. Technical Report 1069, University of Wisconsin–Madison, Computer Science Department, October 1991.

[9] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, April 1989.

[10] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 12–25, December 1995.

[11] David R. Cheriton. The V Distributed System. In *Communications of the ACM*, pages 314–333, March 1988.

[12] Mark Crovella, Prakash Das, Czarek Dubnicki, Thomas LeBlanc, and Evangelos Markatos. Multiprogramming on multiprocessors. Technical Report 385, University of Rochester, Computer Science Department, February 1991. Revised May.

[13] David E Culler, Andrea Dusseau, Seth C. Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Ketherine Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, 1993.

[14] D.L. Dill, A. Drexler, A.J. Hu, and C.H Yang. Protocol Verification as a Hardware Design Aid. In *International Conference on Computer Design: VLSI in Computers and Processors*, October 1992.

[15] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software—Practice and Experience*, 21(8):757–85, August 1991.

[16] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of the 1996 ACM SIGMETRICS Conference*, 1996.

[17] Dror G. Feitelson and Larry Rudolph. Gang Scheduling Performance Benefits for Fine-Grained Synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–18, December 1992.

[18] Douglas P. Ghormley, David Petrou, and Thomas E. Anderson. SLIC: Secure Loadable Interposition Code. Technical Report CSD-96-920, University of California at Berkeley, November 1996.

[19] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–82, February 1988.

[20] Yousef A. Khalidi, Jose M. Bernabeu, Vlada Matena, Ken Shirriff, and Moti Thadani. Solaris MC: A multi computer OS. In *Proceedings of the 1996 USENIX Conference*, pages 191–203, Berkeley, CA, USA, January 1996. USENIX.

[21] James R. Larus and Eric Schnarr. EEL: Machine-independent Executable Editing. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, volume 30(6), pages 291–300, June 1995.

[22] Steven S. Lumetta and David E. Culler. The mantis parallel debugger. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 118–26, Philadelphia, Pennsylvania, May 1996.

[23] Steve Maguire. *Writing Solid Code*. Microsoft Press, 1993.

[24] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the Performance of Log-structured File Systems with Adaptive Methods. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[25] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite Network Operating System. *IEEE Computer*, 21(2):23–36, February 1988.

[26] John K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Third International Conference on Distributed Computing Systems*, pages 22–30, May 1982.

[27] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of the 1995 USENIX Summer Conference*, pages 213–223, January 1995.

[28] Sun Microsystems. XDR: External Data Representation Standard. Technical Report RFC-1014, Sun Microsystems, Inc., June 1987.

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

[30] The MPI Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883, November 1993.

[31] Marvin M. Theimer. Personal communication, June 1994.

[32] Thorsten von Eicken, David E. Culler, Steh C. Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, May 1992.

[33] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.

[34] D. Walsh, B. Lyon, G. Sager, J. M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss. Overview of the Sun Network File System. In *Proceedings of the 1985 USENIX Winter Conference*, pages 117–124, January 1985.

[35] Neil Webber. Operating System Support for Portable Filesystem Extensions. In *Proceedings of the 1993 USENIX Winter Conference*, pages 219–228, January 1993.

[36] Sognian Zhou, Jingwen Wang, Xiaohn Zheng, and Pierre Delisle. Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computing Systems. Technical Report CSRI-257, University of Toronto, 1992.

[37] Songnian Zhou. LSF: Load Sharing in Large-scale Heterogeneous Distributed Systems. In *Proceedings of the Workshop on Cluster Computing*, December 1992.