GLUnix: a Global Layer Unix for a Network of Workstations

Douglas P. Ghormley
University of California at Berkeley
ghorm@cs.berkeley.edu

David Petrou
Carnegie Mellon University
dpetrou@cs.cmu.edu

Steven H. Rodrigues
Network Appliance Corporation
steverod@netapp.com

Amin M. Vahdat
University of California at Berkeley
vahdat@cs.berkeley.edu

Thomas E. Anderson
University of Washington
tom@cs.washington.edu

February 23, 1998

Summary

Recent improvements in network and workstation performance have made workstation clusters an attractive architecture for diverse workloads, including interactive sequential and parallel applications. 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, operating system middleware for a cluster of workstations. GLUnix was designed to provide transparent remote execution, support for interactive parallel and sequential jobs, load balancing, and backward compatibility for existing application binaries. GLUnix was constructed to be easily portable to a number of platforms. GLUnix has been in daily use for over two and a half years and is currently running on a 100-node cluster of Sun UltraSPARCs.

This paper relates our experiences with designing, building, and operating GLUnix. We discuss three important design tradeoffs faced by any cluster system and present the reasons for our choices. Each of these design decisions is then re-evaluated in light of both our experience and recent technological advancements. We then describe the user-level, centralized, event-driven architecture of GLUnix and highlight a number of aspects of the implementation. Performance and scalability measurements of the system indicate that a centralized, user-level design can scale gracefully to significant cluster sizes, incurring only an additional 220 µsec of overhead per node for remote execution. The discussion focuses on the successes and failures we encountered while building and maintaining the system, including a characterization of the limitations of a user-level implementation and various features that were added to satisfy the user community.

Key words: Distributed Operating Systems, Networks of Workstations, Single System Image, Transparent Remote Execution.

Introduction

Historically, computing models have been separated into three distinct paradigms, each optimized for a particular class of applications: desktop workstations used primarily for interactive jobs, centralized compute servers such as supercomputers and mainframes used for compute-intensive or I/O-intensive batch jobs, and massively parallel processors (MPPs) used for batch parallel jobs. However, technology trends of the past two decades have steadily eroded the boundaries of these paradigms. The narrowing performance gap between commodity workstations and supercomputers as well as the availability of commodity high-speed local area networks [10, 18, 5] has led to the concept that networks of workstations (NOWs) can support all three types of workloads [2].

By leveraging commodity high-performance workstations and networks, the primary challenge in building a NOW shifts to the system software needed to manage, control, and access the cluster. Of course, to be positioned as an attractive alternative

to both workstations and supercomputers, NOWs must support all the features available from either system, including interactive parallel and sequential jobs, transparent remote execution of existing applications, and dynamic load balancing across the cluster. To keep pace with exponential improvements in hardware performance, the software should also be quickly portable to new platforms.

When we set out in 1993 to select the system software for the Berkeley NOW, some of the features listed above were present in a few existing systems, but none provided a complete solution. LSF [47] and Utopia [46] provided support for remote execution of interactive sequential jobs, but did not support parallel jobs or dynamic load balancing. PVM [38] supported parallel jobs but lacked the gang scheduling necessary to run closely synchronized parallel jobs efficiently [21]. PVM also did not integrate the parallel job environment with interactive sequential jobs or provide dynamic load balancing. Condor [11] could dynamically balance cluster load through job migration, but did not transparently support existing binaries: jobs were restricted to a limited set of operating system features and had to be re-linked with a special library. Further, Condor only operated in batch mode. Sprite [31] provided dynamic load balancing through process migration [20] and maintained full UNIX I/O and job control semantics for remote jobs, but did not support parallel jobs. In addition, it was a complete operating system, requiring significant effort to keep up with workstation evolution.

GLUnix (*G*lobal *L*ayer *Unix*) was designed to address these limitations by providing a user-level layer to transparently glue a network of workstations into a single 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. The system is built at the user level to minimize the porting effort required to run on new platforms. For reasons discussed in this paper, GLUnix was unable to completely fulfill this vision while remaining a portable, user-level solution. To overcome these limitations, we are developing a non-intrusive kernel extension system called SLIC [23] which will enable GLUnix to fulfill its design potential.

The current implementation of GLUnix performs remote execution, but lacks complete transparency. It maintains most, but not all, UNIX I/O semantics and provides intelligent load balancing at job startup, but does not migrate processes. GLUnix clusters are not completely fault-tolerant: a central master node must always be available, though the system can survive failures to any number of other nodes. GLUnix runs existing application binaries as well as parallel programs written for GLUnix; parallel jobs are gang-scheduled when needed. GLUnix has been in daily use for over two and a half years, with 160 users throughout the Berkeley campus and 1.7 million jobs run on the cluster. User experience with GLUnix has been 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 built at the user level or attempting to provide access to network-wide resources. We begin by describing in 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 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 needs. The paper concludes with a discussion of related work and future directions for GLUnix.

Goals and Features

History

GLUnix was conceived in 1993 as the global operating system layer for the U.C. Berkeley NOW project. A primary goal of the NOW project was to construct a platform that can simultaneously execute interactive parallel and sequential jobs with negligible slowdown for either class of workload. A NOW should guarantee one workstation's worth of performance to each user at all times, with the possibility of recruiting all available cluster resources (CPU, disk, memory, network) for parallel programs when needed. The feasibility of supporting integrated parallel and sequential workloads was established by an early simulation study [6] of workloads measured from a 32-node CM-5 at Los Alamos National Laboratory 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 studied with minimal slowdown.

Given our target NOW environment, we identified the following set of goals for our cluster software system:

- *Portability:* To keep up with the rapid pace of hardware evolution and to enable heterogeneous clusters, the system should be written to be quickly and easily portable to new operating systems or hardware platforms.
- *Timeliness:* An initial implementation of the software should be up and running user jobs as soon as possible. The complete system should be finished by the end of the NOW project.

- *Binary Compatibility:* The system should provide remote execution, load balancing, and coscheduling without requiring application modification or relinking. Existing applications should be able to benefit transparently from new features.
- *High Availability:* When a node crashes, the system should continue to operate. However, jobs with a footprint on a failed node can be terminated. The system may provide hooks for application fault-tolerance mechanisms.
- *Performance*: Sequential and interactive applications should run as if a single dedicated workstation were available to them. Similarly, parallel workloads should run as if a dedicated supercomputer were available.

We then identified the set of features that the system would need to provide:

- *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 job placement policies to determine the best location to initially execute each job. The system should also dynamically migrate jobs either as cluster load becomes unbalanced or as nodes are dynamically added to 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 [33, 6, 21].

Architecture Alternatives

We faced three major design decisions in the development of GLUnix: kernel level vs. user level, centralized vs. decentralized, and multi-threaded vs. event-driven.

Kernel Level vs. User Level

The first and perhaps most important choice was to decide how much functionality to leverage from the underlying system. The options were to (i) build a new operating system, (ii) modify an existing operating system, or (iii) "build on top" of existing systems (now called "middleware"). Though it would likely provide the best performance, the first option of building a new operating system was immediately rejected because we felt that building a new OS was completely incompatible with our goals of portability and timeliness, far outweighing any potential performance improvements. Modifying an existing operating system enables the use of all in-kernel services, not just those available via the user application programming interface (API), and allows changing the implementation of those services to provide additional functionality or better performance. Existing applications can transparently take advantage of these modifications. The tradeoff is that making kernel modifications requires an understanding of an existing kernel—a large and complex system containing hundreds of thousands of lines of code. Kernel modifications are often not portable between successive releases of the same operating system [45], and not at all between different operating systems. In addition, operating system kernels typically lack state-of-the-art debugging and development tools, hindering development. Finally, understanding and modifying operating system kernels is a time-intensive task[1].

Building a middleware layer essentially reverses the advantages and disadvantages of modifying the operating system. Programming at the user level only requires an understanding of the kernel's API, which is considerably simpler than understanding kernel internals. Development is also assisted by numerous development and debugging tools which are frequently unavailable for kernel modifications. User-level layers can be more portable than kernel modifications, although to ensure portability, code must be restricted to a widely-available, standardized set of features (such as POSIX). However, middleware layers are limited by the functionality and performance of the API provided by the underlying operating system. In particular, we were concerned that the limitations of previous user-level systems [11, 47, 34] were an indication that the functionality required for transparent remote execution was simply not available at the user level. We concluded that the advantages of portability and timeliness made a user-level middleware layer the best option at the time. To alleviate the limitations of a user-level implementation while retaining compatibility with existing binaries, we planned to explore binary rewriting techniques [43, 26] or develop novel kernel extension techniques.

Given the choice to implement a user-level solution, we then considered the question of whether all programs on a node would run under GLUnix or only GLUnix-aware programs. We chose a multi-phased approach: initially, programs must specifically request to run under GLUnix, either by user command or by application control; as we identified and solved transparency issues, we would gradually subsume more programs under GLUnix control until all programs on each node were run

[1] In 1993, the only operating systems available for our hardware were commercial products; simply obtaining the source code is often a time-consuming task.

under GLUnix. This path enabled 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 the system was stable).

Centralized vs. Decentralized

Our next design decision concerned the high-level system architecture of GLUnix: a centralized or decentralized design. Centralized systems, with a single master node and a collection of client nodes, are relatively straightforward to design, build and debug. However, the master can be a single point of failure as well as a performance bottleneck. Fully decentralized (peer-to-peer) systems do not depend on any single node and consequently are more resilient to failure and can scale more gracefully to large clusters. Decentralization, however, complicates state management and synchronization, making such systems more difficult to design and debug. The scalability of previous centralized systems [40, 14] led us to decide to begin with a simple, centralized design and to later decentralize system components as needed.

Multi-threading vs. Event-driven

Our final design decision concerned whether to use a multi-threaded or event-driven programming model for individual GLUnix components. Multi-threaded designs have two primary advantages over event-driven programming: increased concurrency with the potential for improved performance, and the relative simplicity of dealing with blocking operations [9, 32]. Maintaining performance in a distributed system with event-driven programming requires that network message handlers use only non-blocking communication operations, treating reply messages as separate events. Event-driven architectures, however, are simpler to design and debug, since they avoid the deadlocks and race conditions of multi-threaded programming. Ultimately, the lack of a portable kernel thread package and thread-aware debuggers in 1993[2] convinced us to adopt an event-driven architecture for GLUnix.

Evaluation

This section provides a retrospective look at our goals and design decisions in light of recent technological developments and our experience. Since the GLUnix design began in 1993, there have been a number of technological advances which would have influenced the goals and design choices of GLUnix. In addition, implementing, debugging, and maintaining GLUnix for over two years has given us valuable experience with a user-level, centralized, event-driven system.

While GLUnix was initially designed to provide NOWs with many of the features available in supercomputers and massively parallel processors (MPPs), in recent years symmetric multi-processors (SMPs) have become increasingly cost-effective and more widely used. The tight processor integration offered by SMP operating systems presents a number of advantages over traditional MPP operating environments: dynamic load balancing, low-cost shared memory, and support for interactive jobs. SMPs also 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 must provide comparable services. 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 because of the user level limitations of today's commodity operating systems (see the section on "User-Level Challenges and Tradeoffs"). To remove these limitations and enable GLUnix to achieve full functionality while retaining the advantages of a user-level solution, we are developing SLIC, a system to enable the transparent interposition of trusted, portable extension code into existing commodity operating system kernels [23]. By overcoming the limitations of developing at the user level, SLIC will enable GLUnix to provide a transparent single system image.

A second technological advancement that has occurred in the past four years is that kernel modification has become considerably simpler. A number of operating systems now provide interfaces for new kernel modules such as file systems, scheduling policies, and network drivers. These modules are dynamically linked into the kernel without a kernel recompilation or even system reboot, and distributing them does not require distribution of source code. However, such interfaces still tend to be undocumented, making them difficult to develop against without the availability of kernel source code. With respect to debugging, an increasing number of operating systems now include advanced kernel debugging support which was unavailable in 1993. While these factors simplify the implementation of new operating system functionality in the kernel, kernel modules are not yet portable among operating systems. Given our goals of portability and timeliness, we believe that the user level approach which we took remains the best available alternative.

With respect to the decision to implement a centralized GLUnix master, measurements of the implementation indicate that the centralized master is not the performance bottleneck of the system. The master requires only 220 μsec of processing per

[2] For example, one of our target architectures, HP-UX, did not have kernel support for threads until 1996.

node to start a parallel program. Thus the master requires less than a quarter of a second to process a 1000-node execution request. Current scalability bottlenecks of the system are BSD sockets over TCP/IP and the 10 Mb/sec switched Ethernet used by GLUnix. Performance improvements can be expected by porting GLUnix to use higher performance communication layers such as Active Messages [29] and faster networks such as the 1.28 Gb/s Myrinet [10].

As development of GLUnix proceeded, maintaining all event handlers as non-blocking routines became increasingly frustrating and considerably reduced code readability, maintainability, and correctness. Distributed GLUnix components often interact in a remote procedure call (RPC) [8] like fashion, issuing requests and needing replies. To maintain system responsiveness, GLUnix must return to the main event loop after issuing a request to a remote node. This requires the programmer to manually encapsulate and store all local event state before returning to the event loop, only to manually restore that same state when the reply arrives. The multi-threaded programming model maps more naturally onto RPCs — threads may issue RPC requests and block awaiting replies; local state is automatically encapsulated on the thread's stack, requiring no additional effort from the programmer to save and restore it. Given our experience and the increased availability of multi-threading support, today we would implement GLUnix as a multi-threaded program.

Using GLUnix

The primary service provided by GLUnix is remote execution of both parallel and sequential jobs. This section describes the abstractions provided by GLUnix, how they operate within the service of remote execution, and the interfaces to access these abstractions and services.

GLUnix Abstractions

To provide remote execution of both parallel and sequential jobs, GLUnix extends some existing UNIX abstractions and introduces new abstractions, borrowing heavily from MPP environments such as that of the CM-5. The new abstractions include *network programs* and globally unique *network program identifiers* (NPIDs) for GLUnix jobs and *virtual node numbers* (VNNs) to name the nodes running a network process. The existing abstractions of signal delivery to remote applications and I/O redirection were extended to support parallel and remote jobs. GLUnix provides both programming and command-line interfaces to access these abstractions.

Network Programs

A *network program* is an executing parallel or sequential job that is controlled by GLUnix. Network programs can be located anywhere in the cluster and are identified using a 32-bit, cluster-unique *network program identifier* (NPID) which is assigned and tracked by GLUnix. Using a cluster-wide, location-independent identifier provides the level of virtualization necessary to enable transparent remote execution and dynamic process migration.

Parallel Programs and Virtual Node Numbers

GLUnix supports parallel programs that use the Single-Program-Multiple-Data (SPMD) programming model. In the SPMD model, a collection of N separate processes run the same application binary and work cooperatively on a single computation. A parallel program is identified by a single NPID. The number of individual processes comprising the parallel program is referred to as the *parallel degree* of the program[3]; processes within a parallel program are distinguished by the use of *virtual node numbers* (VNNs), numbered $0 \ldots N-1$. Just as with NPIDs, VNNs are location-independent and enable transparent process placement and migration within the GLUnix cluster.

Signal Delivery

In UNIX, signals are used to indicate exceptional conditions (page faults, I/O errors, etc.) or for process control (to suspend or terminate a running program). By default, signals in GLUnix are delivered to an entire NPID; that is, each individual process in a parallel program will receive the same signals in the same order. GLUnix also supports additional signaling semantics, for example, the signaling of individual processes of a parallel program.

[3] GLUnix treats sequential programs as parallel programs with a parallel degree of 1.

I/O Redirection

GLUnix extends the standard UNIX abstractions of input and output byte streams to remotely running sequential and parallel programs, including file redirection and pipes [35]. Input given to a GLUnix program, either from the console keyboard, a file, or a pipe, is replicated by default to all processes of a parallel program. Analogously, output from network programs is multiplexed to the user's console, a file, or a pipe in the normal UNIX manner. The `stdout` and `stderr` streams from remote processes are kept distinct as in traditional UNIX, allowing the user to pipe `stdout` to a file while `stderr` remains connected to the console. GLUnix further adds additional I/O modes to network program I/O. Each line of output from remote processes of a parallel program may be prepended with the VNN of the process which generated that line.

Parallel Program Support

Just like MPP operating systems, GLUnix provides specialized scheduling support for parallel programs in the form of barriers and coscheduling [33]. 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. 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 and resume execution.

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 typically take this approach (e.g., PVM [38]). However, a number of studies [16, 22, 7, 21] demonstrate that the lack of coscheduling 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.

GLUnix Interfaces

`int Glib_Initialize();`	Enable GLUnix services for this program.
`int Glib_Spawn(int numNodes, char *progPath, char **argv);`	Start program `progPath` on `numNodes`.
`int Glib_CoSpawn(Npid npid, char *progPath, char **argv);`	Start up program `progPath` on the same nodes that network process `npid` is running on.
`int Glib_GetMyNpid();`	Get the NPID of this job.
`int Glib_GetMyVNN();`	Get the Virtual Node Number of this node within the program.
`int Glib_GetIpByVNN(VNN vnn);`	Get the IP address of the virtual node `vnn` in this process.
`int Glib_Signal(Npid npid, VNN vnn, int sig);`	Send a signal to the virtual node `vnn` within the network process `npid`.
`int Glib_Barrier();`	Wait for all other nodes in the network process to make this call, then proceed with execution.

Table 1: Selected functions of the GLUnix API.

Accessing GLUnix services is done through either the GLUnix API (for programmers) or through command-line utilities (for users of the cluster). Table 1 shows a portion of the GLUnix API. Initializing GLUnix services is done via `Glib_Initialize();` this function creates sockets, locates the per-cluster master and per-node daemon processes, and prepares to run a program. `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; it is used by the Mantis [27] 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 all of the VNNs of a program running under GLUnix. The `Glib_Barrier()` function is used to synchronize parallel applications.

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 2. The `glurun` utility allows unmodified applications to be run under GLUnix; `glurun` distributes jobs across one or more nodes using GLUnix to select the least loaded

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 2: The GLUnix tools provided to users. Each of these utilities is linked with the GLUnix library to provide a service.

nodes in the cluster. The `glukill` utility enables users to signal their jobs from any node 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. `glush` is a modified `tcsh` shell which uses `Glib_Spawn()` instead of `exec()` to run programs on the cluster. `glups` and `glustat` are used to query cluster status.

GLUnix Architecture

This section provides an overview of the GLUnix architecture. We begin with the basic assumptions of the design, and then discuss the design itself and give some illustrative examples of how the system works. We then share some software engineering lessons we learned from constructing the system.

Assumptions

In constructing GLUnix, we made two primary assumptions concerning the target environment: shared filesystems and homogeneous operating systems. Since the Berkeley NOW project included a separate group implementing 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 file to store the network address of the centralized GLUnix master for bootstrapping. Our cluster currently uses an NFS-mounted file system for this purpose.

The second assumption was that operating systems and binary formats would be homogeneous 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 simplified the implementation of GLUnix by eliminating difficulties with byte ordering, data type sizes, and data structure layout for internal GLUnix 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 [37] and developing a mechanism for supporting multi-platform binaries.

System Architecture Overview

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., 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 remote 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 request GLUnix services. Unmodified applications which have not been linked with the GLUnix library can still be run remotely via GLUnix and can still use standard UNIX facilities such as I/O and job control, but will not have access to advanced cluster features (such as parallel execution) provided by the GLUnix API.

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.

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,

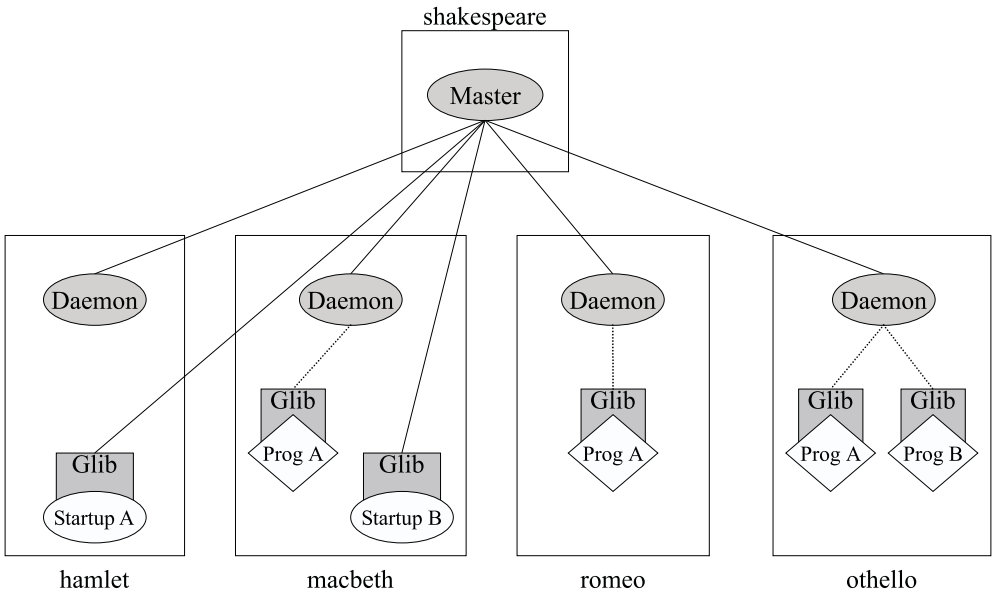

Figure 1: The basic structure of a GLUnix cluster. The solid lines indicate message connections between GLUnix components; the dotted lines indicate parent-child process relationships. In this example, the master is located on the machine named `shakespeare` and daemons are located on `hamlet`, `macbeth`, `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`. The I/O connections from the startup to the remote processes are not represented.

though it is designed to be easily replaceable by a module using faster communication primitives, such as Active Messages [42].

The main event loop is located in the `Comm` module. GLUnix events consist of both messages and signals. When a message arrives, the `Msg` module unpacks the message into a local data structure and then invokes the appropriate 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 signal. Each module registers handlers for the messages or signals they are interested in.

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.

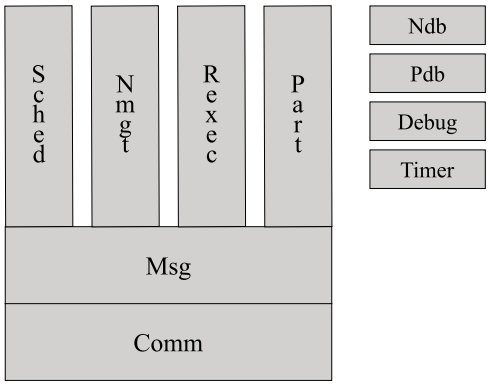

Figure 2: The logical internal structure of the GLUnix master, daemon, and library. Although the module structure is the same in all three components, most modules have different code for each.

Failure Recovery

GLUnix is currently able to tolerate failures of the GLUnix daemons, startup processes, and, of course, user programs, or the machines where any of these are running. 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. Although not currently implemented, 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 check-pointing or restarting applications, though applications are free to use check-pointing packages such as `libckpt` [34] if necessary.

Transparent Remote Execution

When users run GLUnix programs from the command-line, the process `exec`'ed by the user's shell calls the `Glib_Spawn()` routine in the GLUnix library. `Glib_Spawn()` sends the GLUnix master a remote execution request containing the command-line arguments, the parallel degree of the program, and the user's current environment including groups, current working directory, umask, and environment variables. `Glib_Spawn()` then converts the calling process into a GLUnix startup process. When the master receives the execution request, it uses load information periodically sent from the daemons to select the *N* least loaded nodes in the system. Users can influence this selection process by supplying a description of the candidate nodes to select from (e.g., "`u0..u50,u73`") and can even disable the load sorting process to support controlled experimentation (e.g., "`=u0..u50,u73`"). Once the nodes have been selected, the `Rexec` module of the master registers the new network program 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 establishes I/O connections with the startup process for `stdin`, `stdout`, and `stderr`. The `Rexec` module then performs a `fork()`, recreates the environment packaged up by the startup process, and finally does an `exec()` of the program with the original command-line arguments. The daemon will report success or failure of the execution 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 looks up the target network program in the `Pdb` module and forwards the signal request to the daemons which are managing the processes of this network program. Those

daemons then signal the appropriate local processes using the `kill()` system call.

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 [38] 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.

Barriers and Coscheduling

The GLUnix barrier is implemented using a standard binary-tree algorithm. A process of a parallel program invokes the `GLUnix_Barrier()` library call which sends a barrier request message to the local daemon. That daemon identifies the other daemons which are managing the processes for this parallel program. The daemons then use a standard binary tree reduction algorithm to determine when all processes of the parallel program have entered the barrier. When the daemon at the root of the barrier tree detects completion of the barrier, the release message is broadcast back down the 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 [33] to determine a time-slice order for all runnable parallel programs. Periodically, the master sends a message to each daemon identifying which NPID should be scheduled on that node. Each daemon maintains a NPID to local UNIX `pid` mapping in its process database. The daemon uses this mapping to send a UNIX `SIGSTOP` signal[4] 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 solution is, however, only an approximation of coscheduling. 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 interactive, non-GLUnix jobs which will run during a parallel program's time slice.

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 bugs more difficult to locate and fix. The book "Writing Solid Code" [28] led us to a adopt a less tolerant internal structure. System debugging became significantly easier and proceeded more quickly. The GLUnix library and the communication protocols used between GLUnix components both remain tolerant to errors in order to maintain overall system stability.

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 of a parallel program 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. As GLUnix became more stable, it was run on 35 nodes and finally transfered over to a cluster of 100 UltraSPARCs. 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 the "Performance and Scalability" section.

GLUnix Usage

GLUnix has been running and in daily use by the Berkeley CS Department since 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. GLUnix runs on Solaris versions 2.3–2.6 and a basic port to Linux on x86's has also been completed. As of January, 1998, GLUnix has had over 160 users during the course of its lifetime and has run 1.7 million jobs. GLUnix has been in daily use by both production users as well as system developers; Figure 3 shows the number of GLUnix jobs per week since October, 1995 while Figure 4

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

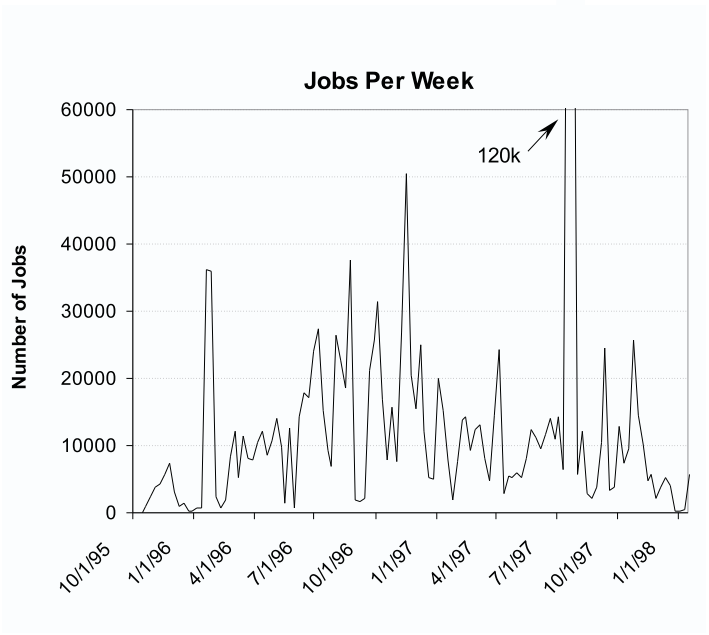

Figure 3: Jobs run per week from October, 1995 through January, 1998.

shows a histogram of the parallel degrees of jobs run under GLUnix. Monitoring statistics taken over the course of 100 days indicate that GLUnix was operational over 97% of that time.

User feedback concerning GLUnix has been positive. GLUnix has enabled or assisted systems research and program development in a number of areas. GLUnix has been used by the graduate Parallel Architecture and Advanced Architecture classes and by other research departments here at Berkeley. Implementations of the parallel programming language Split-C [17] and the Message Passing Interface (MPI) [39] 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 set a record for the world's fastest sort time (8.4 GB in under one minute), p-murphi [19], 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 at the time[5].

The largest use of GLUnix and the Berkeley NOW cluster has been as a compute server for distilling thumbnail images off of the web[6]. 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 system research [30], two-dimensional particle in cell plasma [41], three-dimensional fast Fourier transforms, and genetic inference algorithms.

GLUnix was also found to be extremely useful for testing and system administration. Initially, parallel stress testing of the Myrinet network and Active Messages was accomplished using scripts built around `rsh`. As GLUnix became stable and was put into daily use, the `rsh` scripts were quickly abandoned in favor of the increased performance and job control facilities of GLUnix. From that point, parallel testing of cluster components proceeded at a much faster pace. 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 the section on "Social Considerations, Batch Jobs") has been used for managing very large numbers of short and medium simulation runs, such as the thousands of simulations needed during the investigation of implicit

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

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

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

coscheduling [21].

Performance and Scalability

This section evaluates the performance and scalability of GLUnix and relates some of our experiences in performance tuning. All of the data presented in this section are measured with GLUnix running on 167MHz Sun UltraSPARCs running Solaris 2.5.1. Executed binaries are stored on a standard Auspex NFS server which is shared by all machines. Although the cluster has a 1.28 Gb/s Myrinet network which is used by parallel programs running under GLUnix, the Active Messages networking software has only recently supported the client/server model needed by GLUnix, and we have not yet modified GLUnix to use it. Consequently, all measurements presented here are for GLUnix using BSD sockets over TCP/IP on a 10 Mb/sec switched Ethernet.

Figure 5 and Table 4 identify the various components of the remote execution latency of GLUnix for a sequential job. This breakdown was generated by inserting `gettimeofday()` calls at strategic points in the code and averaging the times over 50 runs. The `gettimeofday()` system call has a resolution of less than 1 µsec and an overhead of approximately 1 µsec, causing negligible perturbation of the system's performance. Segment A, "get master hostname", represents the time

Graph Label	Time	Description
A	16.7 msec	get master hostname over NFS
B	3.6 msec	misc initialization
C	9.0 msec	ping the master
D	4.3 msec	`getcwd()`
E	4.7 msec	network latency
F	1.7 msec	master receives message
G	3.9 msec	network latency
H	3.0 msec	misc initialization
I	11.0 msec	setup `tty` master, connect to daemon
–	1.6 msec	`fork()`
J	6.2 msec	`chdir()`
K	34.8 msec	setup `tty` slave

Table 4: Time spent on various activities during remote execution. The graph labels correspond to the segments of Figure 5. Times are averaged over 50 runs.

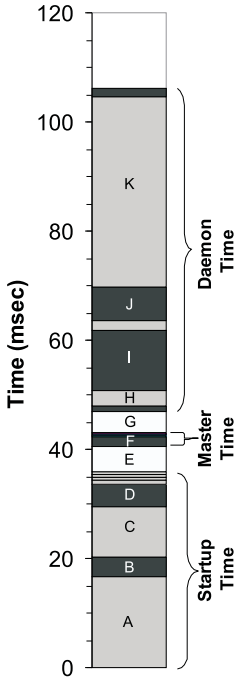

Figure 5: An analysis of the remote execution time of a null sequential program run under GLUnix. The labels correspond to the rows in Table 4. The total time depicted is 106.0 msec. Only the components which take 1.0 msec or more are labeled. The unlabeled segments amount to 3.8 msec, or 3.6%, of the 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.

required to read the master's hostname from an NFS-mounted filesystem. Segment C, "ping master", was not part of the original GLUnix design, but was added later in response to user feedback. When a program first initializes the GLUnix library by calling `Glib_Initialize()`, the library sends a ping message to the master to check 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. Segment D, "`getcwd()`", represents the time required for the daemon to change to the NFS-mounted current working directory of the startup process; network traces revealed that this operation was generating two NFS RPCs to the file server. Segments E and G represent the time spent in the TCP/IP protocol stacks as well as physical link transmission time; the remote execution message transmitted was 2.2 kB. Segment I includes both the time to setup a master `tty` and to make three I/O connections from the remote child to the startup process (a separate connection is made for `stdin`, `stdout`, and `stderr`). Section J shows the cost of creating the slave `tty` and attaching the child process to it.

Figure 5 and Table 4 shows that the dominant factor in remote execution latency is the `tty` support which accounts for 45.8 msec, or 43.1%, of the total time. The second largest factor is the network file system which accounts for 27.2 msec, or 25.6%, of the total time. Neither of these factors occur in the master. Very little of a sequential program's latency is due to the master. The total time spent in the master is 2.4 msec, or only 2.2%, of the total time.

Scalability

One of the main concerns with a centralized master architecture is that the master will become a performance bottleneck. The upper line in Figure 6 shows the total elapsed wall-clock time for running a null program of different parallel degrees under GLUnix. These times were measured using `tcsh`'s built-in `time` command and averaged over 20 consecutive runs. To eliminate file cache effects, each run of the same parallel degree was sent to the same set of nodes and the caches were warmed with an initial untimed run. The size of the remote execution message sent to each daemon was 2.2 kB. GLUnix can run a sequential process in 0.24 sec and a 95-node parallel program in 1.3 seconds. For comparison, an `rsh` of a null application under similar conditions takes approximately 2.2 seconds to a single remote node, and over 90 seconds to 95 remote nodes.

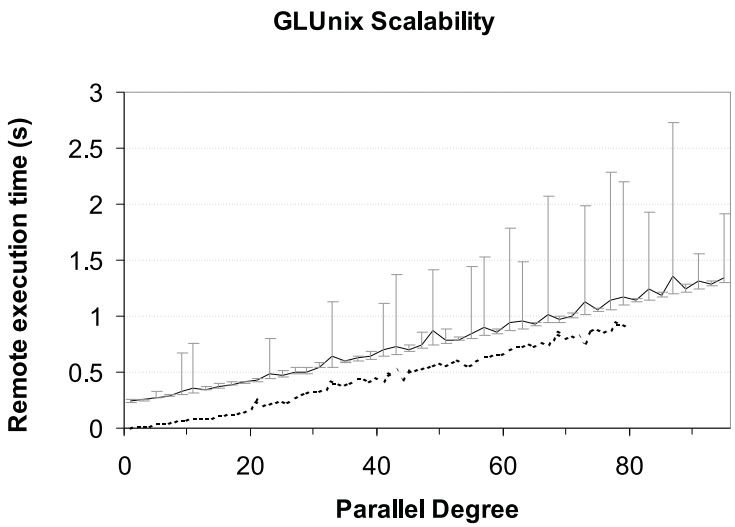

Figure 6: The upper line is the time taken to run parallel programs of various sizes to completion. Each data point is averaged over 20 runs. The error bars indicate the minimum and maximum time of those 20 runs, while the solic line graphs the average time. The lower line is the time taken by a microbenchmark to simply establish I/O connections to the startup process. The slope of both lines is 12 msec/node.

The lower line in Figure 6 represents the time required just to establish the I/O connections from the parallel processes to the GLUnix startup process. These times was measured using a microbenchmark comprised of a single server with multiple clients, each client establishing three TCP/IP socket connections to the server. The slopes of each line are approximately the same, indicating that I/O connection establishment to a startup process is the primary scalability bottleneck for remote execution. Figure 5 identifies many of the costs which account for the gap between the two lines.

To illustrate more directly the scalability of the centralized master, independent of network stack overhead or link contention, Figure 7 presents the computation time spent in the master when running jobs of different degrees of parallelism (the conditions were similar to Figure 6). When running parallel applications, the time spent in the master increases by an average of 220 µsec/node. In contrast, Figure 8 graphs the time spent in the network stacks and on the physical network for the same test. The switched Ethernet begins to show serious contention after a certain threshold. Running `netstat` on the master node indicates a significant increase in collisions of outgoing Ethernet packets during remote execution. Remote execution messages in GLUnix are sent from the master to each daemon which will run a parallel job. Each of these daemons generate a reply message, indicating success or failure. When performing remote execution to many nodes, the replies from earlier daemons collide on the Ethernet with the remote execution requests sent to later daemons.

Figure 9 evaluates the skew when coscheduling parallel programs. Coscheduling in GLUnix does not generate replies from the daemons and hence will not experience the Ethernet collisions discussed above. The coscheduling skew graphed 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 that the signal delivery and rescheduling times through each node's local operating system are statistically equivalent for different parallel degrees, a reasonable assumption given that each operating system performs the actions independently. The times measured include network stack overheads in the master and daemon as well as physical link transmission times. The coscheduling time slice used for these tests was one second.

The scalability data here reveal two points. First, the processing time required by a centralized master to handle remote execution is relatively low, at only 220 µsec/node. Additionally, one-to-many communication patterns also scale relatively well, such as coscheduling in Figure 9 at 203 µsec/node. Second, many-to-one communication patterns using TCP/IP over the switched Ethernet exhibit poor scalability, as demonstrated by Figures 6 and 8. Many of these network contention effects can be alleviated by modifying the GLUnix `Comm` module to use Active Messages [29] across the Myrinet [10].

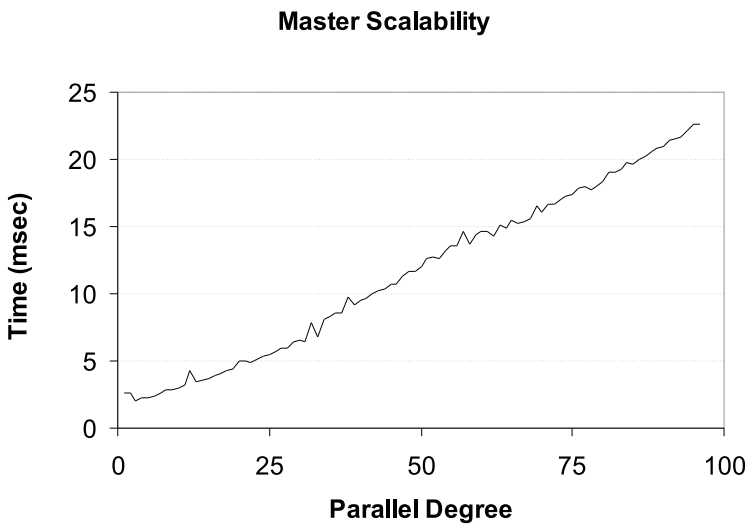

Figure 7: This graph measures the processing time spent in the master when handling execution requests for programs of various parallel degrees. This time does not include any time spent in the startup or daemon processes or in the network. The slope of the line is 220 µsec/node.

Performance Tuning

In this section, we will describe some of our experiences with improving the performance of remote execution. These changes were driven by the needs of the parallel NOW-Sort application [4], which set a 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. Unfortunately, initial runs revealed that it took more than 60 seconds simply to start the program on 100 nodes.

Investigation revealed that GLUnix was flooding the local NIS server during 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, which accesses the cluster's NIS server to retrieve the requested group information. Running a parallel program on 100 nodes thus flooded 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. GLUnix was modified to use the new system calls.

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. However, the default connection queue length in Solaris 2.5.1 defaults to 32. Thus, the first few 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 had originally staggered the execution of the GLUnix daemons, effectively serializing the I/O connections to the startup and reducing the probability of over-running the accept queue.

Limitations on Scalability

To pipe the input and output between the remote program and the user's shell, GLUnix sets up TCP connections between each remote process of the parallel and the startup process. To properly keep `stdout` and `stderr` separated so that shell piping will work as expected, GLUnix requires a separate connection for each. 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

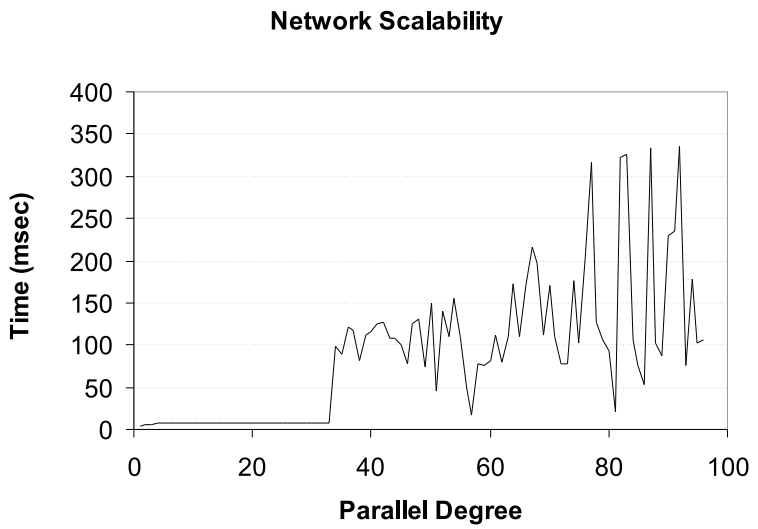

Figure 8: This graph shows the delay between when the GLUnix master begins sending the execution request messages and when the last request message arrives at a daemon. The time includes network stack delays on both sender and receiver as well as network transmission time. Measurements are taken on a 10 Mb/sec switched Ethernet.

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 requires 300 I/O connections to the GLUnix startup process. In Solaris 2.5.1, 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 a parallel degree of 341 or less[7]. Furthermore, since the GLUnix master maintains a connection to each GLUnix daemon and startup process in the system, this file descriptor limit also restricts the total number of GLUnix processes in the system at any one time.

Active Messages provides a light-weight connection establishment abstraction called an endpoint [15]. Porting GLUnix to use Active Messages would dramatically reduce the I/O connection establishment costs and would remove the limit on the number of connections. However, using Active Message endpoints for the I/O connections would require application modification, since endpoints cannot be accessed through native UNIX file descriptors. The FastSockets [36] library provides a way of mapping standard UNIX sockets onto Active Messages, but requires applications to be relinked, which violates the GLUnix goal of binary compatibility.

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, device access, coscheduling, and NFS. 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, they have two disadvantages. First, such facilities are not yet standardized and are not generally available in other operating systems. Second, the `/proc` filesystem 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 in the

[7] $\lfloor 1024/3 \rfloor = 341$.

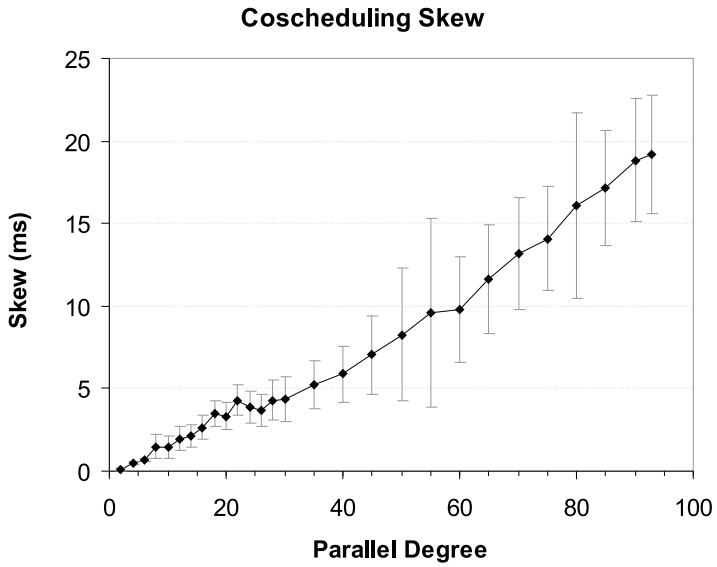

Figure 9: This graph measures the skew arising from coscheduling. The skew is defined as the difference between the earliest and latest times that two processes in a parallel program are started or stopped. Each data point represents the skew for a particular parallel degree averaged over 200 coscheduling events. The error bars indicate one standard deviation above and below the average. The slope of the line is 203 µsec/node.

process, reducing the performance of all application I/O, not just terminal I/O.

Terminal I/O

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 standard user level services. Implementing this behavior would require the GLUnix startup to know how many characters of input were being requested by the remote process. While this can be easily accomplished if the remote program has been relinked with the GLUnix library, this solution is not compatible with the goal of supporting 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.

A more efficient 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.

A second problem with terminal I/O is that GLUnix does not properly handle `tty` behavior for `stdout` and `stderr`. Many applications depend on `tty` support to execute properly. Even applications that do not appear to need a `tty` often behave in unexpected ways in their absence. The standard C library on many different operating systems, for example, performs an `isatty()` system call to determine whether `stdout` should be line or block buffered. Without a remote `tty`, a great many

programs frequently appear to be working incorrectly since they do not produce output for an unusual length of time, or not at all if the program crashes before flushing the output streams. This difference can be easily observed by running "`du /`" and "`rsh remote du /`" on most varieties of UNIX. Initially, GLUnix did not install remote `tty`s when executing jobs; this functionality was added in response to considerable user complaint. However, simply installing a remote `tty` is insufficient. In order to keep `stdout` and `stderr` separate, GLUnix must set up a separate pseudo-terminal for each. This violates standard terminal semantics. In UNIX, if both `stdout` and `stderr` are tied to terminals, they are assumed to be tied to the same terminal. Programs such as `more` and `less` often rely on this fact and will not work as expected when two separate pseudo-terminals are used. 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.

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, when delivering the signal remotely, the GLUnix daemons cannot know if the signal should be sent only to the remote process or to the process and its children. GLUnix therefore 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 for signals originating in the kernel or device drivers (in particular the terminal device driver which is used for job control signaling from a shell).

Device Access

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 may behave differently. Again, although 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 (up to a maximum), the format of which are specified by the device being accessed. Consequently, 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 involves issuing the `ioctl()` system call on the process's home node (where any special devices are located) and intercepting all subsequent device accesses to user memory. Modern operating systems provide special copy routines to move data between the kernel and user level. Intercepting these copies would enable the system to determine the location and length of data read or written by the device to the application. The kernel's copy routines would effectively have to be replaced with distributed routines which perform remote reads and writes. Such functionality is not supported by either standard user-level services or Solaris's `/proc` API.

Coscheduling

GLUnix currently approximates coscheduling by using `SIGSTOP`/`SIGCONT` signal pairs. Unfortunately, this technique is not transparent to some programs. The delivery of the `SIGCONT` signal will generally interrupt any system call which the process may have been blocked in, returning an `EINTR` error code to that process. While this is the same behavior that a `Ctrl-Z`/`fg` combination would have from a shell, we have encountered a small number of programs and communication libraries which did not properly check for this condition and required modification to execute correctly when scheduled using this method.

NFS

The weak cache consistency policy of NFS [44] for file attributes limits the utility of the `glumake` program. 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 distributed computing environments. The presence of these stale attributes, however, can break the illusion of transparent remote execution for programs that depend on them, such as `glumake`. When `glumake` spawns off a remote child to perform a compilation, the file attributes are properly updated on the node where compilation

```
1:  file:  file.o
2:         cc -o file file.o
3:
4:  .o.c:
5:         cc -c $^
```

Table 5: Sample `Makefile` which exposes inconsistency of NFS when run under `glumake`.

takes place, but the old attributes remain cached on other nodes. When `glumake` then 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. A typical manifestation of this problem is when a source file is recompiled into an object file, but the resulting executable is not built. Table 5 shows a sample `Makefile` that exhibits this problem. Lines 4 and 5 will cause `file.c` to be compiled into `file.o`. However, the time-stamp of `file.o` may not update right away on all nodes in the cluster, causing the dependency check for line 1 to access the old time-stamp rather than the new one. Consequently the executable will not be relinked. 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 `Makefile`s to avoid consistency problems in the filesystem is neither transparent nor reasonable. This argues for a strongly-consistent cluster file system [24, 3].

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 shell operations (e.g., `Ctrl-C`, `Ctrl-Z`, pipes, and file redirection) with parallel programs is extremely useful. A second 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.

User-Initiated System Restart

As the GLUnix user community increased, it became increasingly important to have the system continuously available. Periodically, GLUnix would fail, either as a result of system bugs, machine crashes, or network file system problems. Initially, only system developers had the permissions needed 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 arbitrary users to restart the system.

This decision led to an interesting, yet subtle consequence. Users were mistakenly restarting the system when nothing was wrong with it. With the introduction of the reservation system discussed below, 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 continued to grow, there were an increasing number of users who were not familiar with the infrastructure. These users often encountered errors in other aspects of the NOW (such as the NFS file servers or the AM-II network) and, being unsure of the source of the error, would restart GLUnix as a reflex. These two factors led to an explosion of entirely unnecessary GLUnix restarts. We have since reverted to the scenario where only system developers can restart the system, but are exploring other approaches to increasing system availability.

Batch Jobs

Although the original goal of GLUnix focused 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 batch queuing system. Users could submit an arbitrary number of jobs to the batch system which would monitor the load of the GLUnix cluster to determine how many machines were idle. The batch system would then regulate the submission of the batch jobs to GLUnix, being careful not to overrun the cluster. The batch system also had an option which would cause it to terminate submitted batch jobs if the load of the cluster exceeded a certain threshold; this allowed low priority "background" jobs to use all available idle resources without significantly impacting the availability of the system for other users.

GLUnix Reservations

When GLUnix was initially deployed, it had no support for naming or reserving individual nodes. However, as the system became more heavily used, the need became apparent. System developers as well as researchers often need exclusive access to a particular set of machines to take performance measurements or to test experimental kernels. Initially, these reservations were simply made by sending e-mail to a user distribution list. However, 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 this problem, at one point an exchange of roughly 40 e-mail messages over the course of two days was required to establish a reservation for 32 nodes.

To alleviate this problem, we developed a naming and reservation system for GLUnix. The system works by associating three lists with each machine: *aliases*, *users*, and *owners*. The alias list simply contains a list of valid names for 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 a *node specification*[8] to indicate the set of nodes which should be used to run the program. GLUnix expands the node specification into a set of machines, 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, the user supplies a specification of the nodes to reserve, the start and end times of the reservation, and a name for the reservation. When the time for the reservation arrives, GLUnix filters the reservation's node specification by the machines' owner lists, 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. Users then indicate their desire to run on the reserved nodes by placing the reservation name in the node specification.

In addition to dramatically simplifying the process of reserving machines, 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 resource characteristic rather than having to remember the current configuration. When the cluster configuration is modified, the aliases are simply updated, allowing subsequent uses of the alias to properly use the new configuration.

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 became extremely unhappy when demanding jobs were 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 are 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.

Related Work

When we began the 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

[8] The node specification is actually an equation consisting of aliases and set operations on those aliases. For example: "(myrinet2.3∧solaris2.6)−desktop".

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.

Both V [14] and Sprite [31] are research operating systems written from scratch, providing in-kernel support for remote execution and dynamic process migration. Both systems were successful at providing remote execution and process migration, though at an admittedly high implementation cost in the Sprite case [20].

More recently, the Hive [13] project modified Irix to improve the failure characteristics of distributed shared memory machines by tolerating certain hardware failures (fault-containment). Since the project began with an SMP operating system, it will naturally provide a fully transparent Single System Image (SSI), but it significantly less portable to other architectures than is GLUnix.

Two related projects which have taken the approach of minor modifications to existing systems are the Newcastle Connection and Solaris MC. Newcastle Connection [12] was an early distributed system constructed by interposing a distributed software layer on top of the Vax VMS operating system. It required very few changes to the underlying operating system, but intercepted application/kernel interactions and forwarded them to remote nodes to provide remote execution. The Solaris MC [25] group made small modifications to the Solaris kernel to implement cluster facilities, similar to the Newcastle Connection. Solaris MC provides execution-time remote execution and a SSI, but does not do dynamic process migration.

A number of other projects have attempted to provide cluster facilities at the user level. PVM (Parallel Virtual Machine) [38] provides support for parallel programs on clusters of distributed workstations. Whereas PVM supports heterogeneous clusters, GLUnix is currently constrained to homogeneous cluster. PVM was initially designed for wide-area distributed applications across different administrative domains and was not designed to provide the same level of node coupling that GLUnix does. Instances of PVM are typically created per-user, preventing users from seeing or interacting with each other's jobs. Consequently, GLUnix provides a closer approximation of a SSI than does PVM. Interaction with PVM is done through a special shell whereas GLUnix jobs can be run from unmodified UNIX shells. PVM also has only recently begun to support coscheduling [33] which GLUnix was designed to do from the start.

LSF (Load Sharing Facility) [47] provides command line integration with a load balancing and queuing facility. Like GLUnix, when running remote processes, LSF recreates a substantial portion of the user's runtime environment, including current working directory, file creation mask, resource limits, etc. LSF also is very similar to GLUnix in that it uses a startup process to redirect signals and I/O to remote jobs. When the GLUnix project first began, LSF did not provide support for parallel jobs, although parallel support is now available via integrated with PVM. LSF supports node selection based on static and dynamic resource requirements, while GLUnix is only currently capable of supporting static resource requirements via the naming and reservation system. One difference is that GLUnix uses an integrated NPID model for all remote programs, enabling job control and querying to be done from any node in the system and providing the virtualization necessary to implement process migration. In contrast, only LSF batch jobs can be manipulated from any node in the system. Thus, the interface provided by GLUnix is closer to the SSI of SMPs than the interface provided by LSF.

Condor [11] and `libckpt` [34] provide migration and check-pointing, respectively, at the user level. Both systems require application modification or relinking and therefore do not work with existing application binaries. Both systems have a number of limitations in that target applications are restricted from using certain system features, such as network communication, signals, terminal I/O, and shared memory.

Future Work

There are two further research efforts that have arisen out of GLUnix. The first is an effort called Implicit Coscheduling [21] which enables communicating processes to coschedule themselves using various heuristics. One such heuristic is 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 [23] which is a method for alleviating the restrictions of implementing system facilities at the user level. SLIC enables trusted extensions in commodity operating systems by transparently interposing on common UNIX interfaces such as system calls, signals, and virtual memory events. SLIC requires only minimal (and in some cases no) modification to existing operating systems. By loading trusted extensions into a commodity kernel, system software developers can eliminate many of the user-level limitations discussed earlier while retaining the majority of advantages that user-level software development affords.

The second research effort is SLIC [23], a system designed to alleviate the restrictions typically associated with system facilities implemented at the user level. SLIC enables developers to easily insert trusted extensions into commodity operating systems by transparently interposing on common UNIX interfaces such as system calls, signals, and virtual memory events. SLIC requires only minimal (and in some cases no) modification to existing operating systems. SLIC supports a split model in which the bulk of a system may remain at the user level while a small portion of the system is located in kernel extensions, coordinating with the user-level portion through a SLIC interface. Systems such as GLUnix can use SLIC while retaining a considerable degree of portability for two reasons. First, the bulk of the code remains portable user-level code. Second, SLIC interposes extensions on standard UNIX interfaces which have substantially similar functionality across a wide spectrum of operating systems. Although the details of each operating system can vary, a great many of those details can be abstracted away beneath an extension library, enabling SLIC extensions to be considerably easier to port than general kernel modifications. We plan to use SLIC to work around the limitations we faced in implementing a single system image (e.g., tty handling and signal delivery) in GLUnix while retaining the advantages that user-level software development affords. Although initially designed for GLUnix, SLIC will also enable general purpose extension of commercial operating systems.

Conclusions

The GLUnix operating system layer was designed and built as a user-level, centralized, event-driven system to provide system software for networks of workstations. Measurements of the centralized GLUnix master indicate that a centralized design can scale gracefully on networks of workstations, requiring only 220 µsec/node of computation. However, measurements also indicate that traditional TCP/IP protocols over switched Ethernet scale poorly.

GLUnix was originally intended to support interactive sequential and parallel programs through transparent remote execution and load balancing. Though the system has supported both the production and research work of over 160 users over two and a half years, it did not fully 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.

Availability

The GLUnix source code and documentation are publically available at `http://now.cs.berkeley.edu/Glunix`.

Acknowledgments

There are a number of people 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. Many 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. Special thanks go to Joshua Coates for his invaluable help in developing a number of the scripts used in evaluating the performance of GLUnix.

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] Anonymous. Gigabit Ethernet. *Computer*, 29(2), February 1996.

[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. In *Proceedings of Performance/Sigmetrics*, May 1995.

[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] Andrew Birrell and Bruce Nelson. Implementing remote procedure calls. *ACM Trans. Computer Systems*, 2(1):39–59, February 1984.

[9] Andrew D. Birrell. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, January 1989.

[10] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

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

[12] D. Brownbridge, L. Marshall, and B. Randell. The Newcastle Commection. *Software—Practice and Experience*, 12:1147–62, 1982.

[13] 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.

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

[15] Brent Chun, Alan Mainwaing, and David Culler. Virtual Network Transport Protocols for Myrinet. In *Proceedings of the 5th Hot Interconnects Conference*, August 1997.

[16] 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.

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

[18] Martin de Prycker. *Asynchronous Transfer Mode: Solution for Broadband ISDN*. Ellis Horwood Publishers, second edition, 1993.

[19] 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.

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

[21] 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.

[22] 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.

[23] Douglas P. Ghormley, Steven H. Rodrigues, David Petrou, and Thomas E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the 1998 USENIX Conference*, June 1998.

[24] 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.

[25] 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. USENIX, January 1996.

[26] 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.

[27] 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.

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

[29] Alan Mainwaring and David Culler. Active Message Applications Programming Interface and Communication Subsystem Organization. Technical Report CSD-96-918, University of California at Berkeley, October 1996.

[30] 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.

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

[32] John Ousterhout. Why Threads Are A Bad Idea (for most purposes). Invited Talk at the 1996 USENIX Technical Conference, January 1996. See http://www.sunlabs.com/~ouster.

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

[34] 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.

[35] Dennis M Ritchie and Ken Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, 1974.

[36] Steven H. Rodrigues, Thomas E. Anderson, and David E. Culler. High-performance local area communication with fast sockets. In *Proceedings of the 1997 USENIX Conference*, pages 257–274, January 1997.

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

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

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

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

[41] J. P. Verboncoeur, A. B. Langdon, and N. T. Gladd. An object-oriented electromagnetic PIC code. In *Computer Physics Communications*, pages 199–211, 1995.

[42] 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.

[43] 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.

[44] 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.

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

[46] 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.

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