# Comparing Windows NT, Linux, and QNX as the Basis for Cluster Systems

Avi Kavas Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University, 91904 Jerusalem, Israel

**Abstract**

Clusters use commodity hardware and software components to provide an environment for high-performance parallel processing. A major issue in the development of a cluster system is the choice of the operating system that will run on each node. We compare three alternatives: Windows NT, Linux, and QNX — a real-time microkernel. The comparison is based on expressive power, performance, and ease-of-use metrics. The result is that none of these systems has a clear advantage over the others in all the metrics, but that each has its strong and weak points. Thus any choice of a base system will involve some technical compromises, but not major ones.

## 1 Introduction

Rapid improvements in network and processor performance are causing clustered commodity workstations and PCs to become an increasingly popular platform for executing parallel applications. In the past, Unix was used as the platform for almost all parallel systems implementations. Recently, however, it is becoming more common to use Windows NT as the base platform.

The decision which operating system to use involves many considerations, including the operating system's cost and personal experience with the different systems. But there are also technical implications. Our goal is to illuminate these technical issues, by providing a broad comparison of the capabilities and characteristics of the different systems. Throughout, the emphasis is on those features deemed to be important for the implementation of high-performance computing clusters.

Surprisingly, very little work of this nature has been done before. Tanenbaum has compared three microkernels for use in parallel machines [43], but these have not withstood the test of time and the fact remains that today systems such as Linux and Windows are preferred. Lancaster and Takeda have compared these systems, but only in terms of performance [29]; we consider many other aspects as well. There have been some direct comparisons of Linux and Windows for the commercial server market [27, 31], but not in the context of computational clusters.

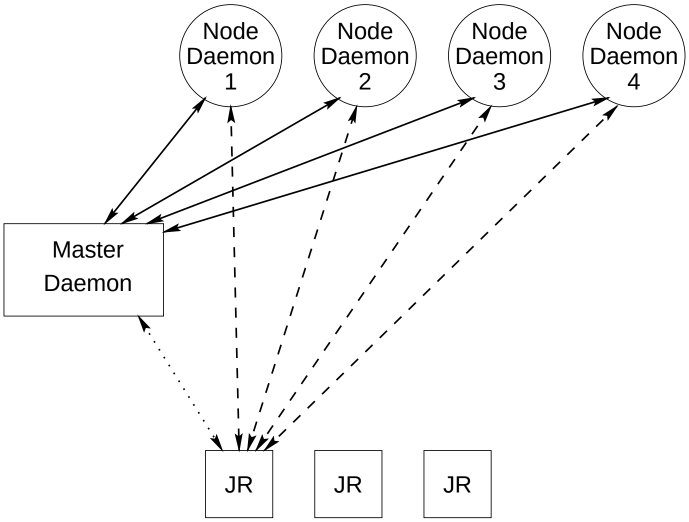

Figure 1: *General cluster architecture. The master daemon handles resource management with the job representative processes (JR), and controls the node daemons.*

## 1.1 Cluster Architecture and Mode of Operation

A cluster is a parallel processing system which consists of a collection of interconnected standalone computers working together as a single, integrated computing resource [41, 33]. The rationale for clusters is the desire to leverage the commodity computing market, and ride the technology curve. These considerations lead to the use of commercial off-the-shelf components, both for hardware and for software. In particular, each node typically runs a conventional operating system such as Windows or Linux. They are tied into a cluster by some user-level processes, often operating as daemons.

Due to their wide appeal, clusters are being used for many different tasks: transaction processing, web-page service, graphics and rendering, parallel scientific computation, and throughput computation. Our focus is on dedicated clusters that are used for parallel computations, similar to large scale MPPs. Thus we emphasize system services that are useful in this context, and relate to the control and efficient execution of multiple parallel jobs. However, we try to provide a broad coverage of features that may be useful for other modes of operation as well.

At a high level of abstraction, many computational cluster systems use a similar architecture: There is a single system-wide control process, and an additional process on each node of the cluster (Fig. 1). The central controller is responsible for configuration management, resource allocation, and job control. The per-node processes collect local data and implement the decisions of the central controller. Example systems based on this design include the Berkeley NOW [24] and ParPar [20]. In addition, every parallel job is represented and controlled by a separate process.

The next section presents a survey of system features that are needed in order to implement such a structure. These include communication facilities used among the different processes, and facilities to spawn and control user processes. In some cases, it is possible to spawn processes remotely without using another process as a local agent. The implications are discussed where specifically relevant. We then go on to discuss performance issues and ease of use. But first we start with some background information about the three systems.

## 1.2 Operating Systems Compared

We focus on the comparison of three systems: Windows NT, Linux, and QNX.

**Windows NT** was introduced in July 1993 and was aimed at the enterprise market, for use on high-end workstations and servers. It was the first version of Windows to support the 32-bit programming model of the Intel 80386, 80486, and Pentium microprocessors (it can also run on Alpha processors, but support for Alpha has been discontinued). Windows NT has a 32-bit flat address space, provides the NTFS file system, C2 compliance security model, Remote Access Server (RAS), and OS/2 and POSIX subsystems. However, it does not provide full POSIX compliance, and cannot use both the Windows API and POSIX API in the same application.

Windows 2000, introduced in February 2000, is the follow-up for Windows NT. It provides better reliability and some new features such as Plug-n-Play, AGP support, Active Directory, scripting tools, native ATM support and more. It also provides improved performance in critical areas such as networking and web and print service. In this paper we focus on Windows NT, which was the dominant version at the time the work was done, and was also available to us for measurements. However, we also mention new features introduced in Windows 2000 where relevant.

Windows XP [44] is the follow-up of Windows 2000, expected in 2002. It features a new user interface design, which is largely meaningless for clusters, and various updates and improvements in different areas, notably networking. It is not reviewed in this paper.

Both Windows NT and Windows 2000 use a microkernel architecture. However, this is not a pure microkernel. Only the operating system environments execute in user mode as discrete processes, including DOS, Win16, Win32, OS/2, and POSIX. The basic operating system subsystems, including the process manager and the virtual memory manager, are compiled with the kernel. They can therefore communicate with one another by using function calls for maximum performance [40].

Several cluster systems are based on Windows NT, including the Velocity cluster at Cornell (`www.tc.cornell.edu/AC3/tech/systems.html`), the HPVM system (`www-csag.ucsd.edu/projects/clusters.html`, [12]), and Millipede [23].

**Linux** is a completely free re-implementation of the POSIX specification, with system V and BSD extensions, which is available in both source code and binary form. The first version of the Linux kernel was made available on the Internet by Linus Trovalds in November 1991. A group of Linux activists quickly formed, and continues to spur on the development of this operating system. Numerous users test new versions and help to clear the bugs out of the software, making Linux the model of open-source development [9] (`www.opensource.org/`).

Linux, like most Unix systems is monolithic, that is, the whole operating system is a single executable file that runs in kernel mode. This binary contains the process management, memory management, file system and the rest.

We chose Linux over other Unix variants (such as Solaris or Tru64 Unix) because it is the most popular basis for the construction of clusters. In fact, according to the clusters sublist at the Top500 web site, the vast majority of high-performance computational clusters employ Linux, and only a few use other Unix variants (even less use Windows NT). The reason for this is the rapid development of Linux and the fact that it is free in source form. This state of affairs has made Linux a major player in the enterprise market, and it is poised to become the standard Unix implementation.

As noted above, many clusters have been built based on Linux. Examples include the LosLobos Supercluster (`www.ahpcc.unm.edu/Systems/Hardware/LosLobos/`), the NPACI Rocks effort (`rocks.npaci.edu/`), The RWCP Score system (`pdsw3.trc.rwcp.or.jp/clusters/`), Beowulf [38], the Berkeley NOW [24], and ParPar [20].

**QNX** is a real-time commercial operating system, developed by QNX Software Systems. The QNX real-time operating system provides applications with a network-distributed, real-time environment that delivers nearly the full device-level performance of the underlying hardware. The architecture consists of a real-time microkernel surrounded by a collection of optional processes (called resource managers) that provide UNIX-compatible system services. By including or excluding resource manager processes at run time, the developer can scale QNX down for ROM-based embedded systems, or scale it up to encompass hundreds of processors connected by various LAN technologies [25]. We focus on QNX version 4.0.

QNX Neutrino, a new version of QNX currently in Beta stage, is also covered in those cases where it provides new features (such as SMP support and POSIX threads support).

While QNX does support a networked environment, few if any computational clusters have been built using it (the nearest example we know of is the Makbilan system, in which nodes ran Intel's RMK real-time kernel [21]). However, the use of a real-time kernel as the basis for a cluster is intriguing, as it might be expected to provide various benefits. Examples include low and predictable overhead for process management and communication events, and the ability to manipulate process priorities and ensure that desired services are not interrupted. We therefore include it in the discussion, with the goal of assessing the degree to which this promise is realized.

# 2 Kernel Services and API Comparison

This section presents a comparison of the set of operating system services and API calls related to parallel systems development as provided by each of the compared operating systems kernels. In both Linux and QNX, functions classified as kernel functions or system calls were compared (those that appear in section 2 of the on-line manual). In Windows, the Win32 API was used for the comparison. The NT system-call interface, called the Native API, is hidden from programmers and largely undocumented. The API that the majority of NT applications write to is the Win32 API, which translates many Win32 APIs to native APIs [37].

## 2.1 Process Control

One of the most basic capabilities required from a parallel system is controlling the processes of parallel jobs. This includes spawning them on the nodes allocated for the job, suspending them in order to schedule another job to run, and resuming them afterwards. Process control also includes the ability to kill or to send a user defined signal to all the processes of a parallel job.

### 2.1.1 Process Creation

CreateProcess() is the fundamental system call used for creating new processes in Windows NT. It creates both a process object and the main thread object of an application. CreateProcess()

allows the parent process to set the operating environment of the new process, including its working directory, security attributes, file handle inheritance, environment variables, priorities, and the command line it is passed [39, 32].

Win32 does not provide the capability to clone a running process (and its associated in-memory contents) as is done by the Unix fork() system call (which is used on both Linux and QNX). This is not such a hardship, since most Unix code forks and then immediately calls exec() [32]. The Linux implementation of fork() does not actually clone the parent process. Instead, it uses the copy-on-write optimization so that common virtual memory pages are shared with read-only permissions. If either of the parent process or the child process tries to modify one of the shared pages, then the kernel duplicates it. An important member of the fork() family of functions is vfork(). This is used to create a new process without fully copying the address space of the parent, and can be useful when the child won't reference the parent's address space and will call exec() to run a new program [42]. Linux also has a __clone() function, which allows the child to share parts of the execution context with its parent; it is used mainly to implement threads.

Besides supporting the fork() family of functions, QNX has a qnx_spawn() system call. This allows the programmer to modify various parameters for the new process, e.g. scheduler type and process priority. The most powerful feature of qnx_spawn() is the option to create the child process on a remote node.

Detailed options of process creation are compared in Table 1.

As noted, QNX allows processes to be spawned on a remote node. Windows has a similar capability as part of the Distributed Component Object Model (DCOM). This is a protocol that enables software components to communicate directly over a network in a reliable, secure, and efficient manner. However, there are two main reasons for not using DCOM to run processes remotely. One is that a significant performance overhead might occur because it uses multiple software layers and interfaces. The other is that it would require users to implement their parallel applications as DCOM objects (need to implement special DCOM interfaces), something which is not desirable since it limits the user to a certain implementation of his application. Also, direct process control (sending signals for example) is not available when using DCOM objects.

### 2.1.2 Process Groups

Sometimes it is more convenient (or even necessary) to treat a set of processes as a single group, e.g. to perform a collective operation on them or to set restrictions on the whole group.

Windows NT 4.0 does not support process groups. Only Windows 2000 offers a new job kernel object that allows the programmer to group processes together. This is used only to create a sandbox that restricts what the job's processes are allowed to do [3]. A detailed description of these restrictions is given in Section 2.1.6.

Linux and QNX support two levels of grouping: sessions and process groups. At the top of the hierarchy are sessions, each of which consists of one or more process groups [28]. In principle, these are useful for the distribution of signals. Unfortunately, this is largely irrelevant for parallel applications, since groups and sessions are limited to processes on the same machine.

| | Windows NT | Linux | QNX |
|---|---|---|---|
| create process on a remote node | not supported | not supported | qnx_spawn() |
| create process on behalf of a user | CreateProcessAsUser()<br>CreateProcessWithLogonW() | not supported | not supported |
| create suspended process | CreateProcess() using CREATE_SUSPENDED flag | not supported | qnx_spawn() using _SPAWN_HOLD flag |
| inherit address space from parent | not supported | fork() | fork() |
| inherit open handles / file descriptors from parent | CreateProcess() | fork() | all in fork(), 10 in qnx_spawn() |
| parent needs to wait() for children to die (to avoid zombies) | never | always | always with fork(), never with qnx_spawn() and _SPAWN_NOZOMBIE flag |
| instruct file system to place executable in cache | not supported | not supported | qnx_spawn() with _SPAWN_XCACHE |

Table 1: *Process creation options.*

### 2.1.3 Process Termination

An important service in a parallel system is the option to kill all the processes in a parallel job. In Windows NT TerminateProcess() causes all the threads within a process to terminate and the process to exit. However, Microsoft documentation [3] suggests to use TerminateProcess() only in extreme circumstances since it does not clean all the resources attached to the process. Specifically, DLLs attached to the process are not notified that the process is terminating, and the process object is not necessarily removed from the system. In Linux and QNX, The kill() system call is used to terminate a process by sending it a SIGKILL or SIGTERM signal. SIGKILL is distinguished by the fact that it cannot be caught nor ignored. QNX allows kill() to send signals to processes on remote machines.

### 2.1.4 Process Suspension and Resumption

Windows NT has no direct API for suspending or resuming processes. The probable reason is that Windows uses thread scheduling rather than process scheduling. Win32 supports thread suspending or resuming using SuspendThread() and ResumeThread(). Single threaded processes can therefore be handled easily by suspending/resuming their primary thread which is returned in the LPPROCESS_INFORMATION structure after calling CreateProcess(). But in order to handle multi-threaded processes, all the threads in the process have to be suspended/resumed individually. Unfortunately there is no direct API for enumerating all the threads of a given process. The Windows registry contains data about all the running threads in the system, so it is possible to list all the threads in the system and check each one's process ID. In Windows 2000, the ToolHelp library supplies a more convenient API for doing this enumeration without digging in the Windows registry, but still all the threads in the system have to be enumerated in order to find the thread handles of a given process.

In Linux and QNX the kill() system call is used to send SIGSTOP or SIGCONT to suspend or resume a process.

### 2.1.5 Process Scheduling and Priorities

Windows NT, Linux, and QNX all implement a priority-driven, preemptive scheduling system. The scheduler selects the next process to run by looking at the priority assigned to every process (thread in Windows NT) that is in the READY state.

Windows NT scheduling is done at the thread granularity. By default, threads can run on any available processor unless processor affinity is used (see Section 2.3.2) [35]. The priority of each thread can be in the range from zero (lowest priority) to 31 (highest priority), as determined by combining its base priority with dynamic adjustments. The base priority, in turn, is a combination of the priority class of its process (IDLE, NORMAL, HIGH PRIORITY, and REALTIME) and the priority level of the thread within the priority class of its process (IDLE, LOWEST, BELOW_NORMAL, NORMAL, ABOVE_NORMAL, HIGHEST, TIME_CRITICAL). Only the system's zero-page thread can have a priority of zero [3].

Linux and QNX perform scheduling at the process level. Both systems offer the following scheduling policies:

1. FIFO scheduling (SCHED_FIFO) in which the current process continues to hold the CPU until it blocks or terminates. The highest priority READY process is always selected.

2. Round-robin scheduling (SCHED_RR) in which processes are given equal time quanta in turn. This allows the processes that share the highest priority level to share the CPU. These two schemes are useful for real-time control.

3. Adaptive scheduling (SCHED_OTHER) is which the CPU is shared as above, but the priorities are adjusted according to a predefined policy that takes CPU usage into account. This is meant to support interactive (desktop) applications.

In QNX, when a process consumes its entire time slice, its priority is lowered by one (only once). If in the next time slice the process will use its whole time slice again, it will stay at that priority. If it didn't use up its entire time slice, the kernel will increase its priority by one. In Linux, the priority calculation takes into account the nice level (set by the nice() or setpriority() system call). The priority is increased for each time quantum the process is ready to run but not running, and decreased when the process is running.

QNX also offers a feature called “client driven priority”, which allows a server to change its priority according to the highest priority of the clients it serves. This feature can be used to prevent the server from serving low priority clients in a high priority.

### 2.1.6 Placing Restrictions on Processes and Users

In Windows NT 4.0 there is no way to set restrictions on a process or a group of processes. On the other hand, Windows 2000 provides a rich API for setting restrictions on jobs or processes (recall that a job is essentially a process group). These can be used to prevent processes from monopolizing system resources. However, disk quotas are still missing.

Linux has mechanisms to support filesystem quotas and process limits. You can define storage quota limits on each mountpoint for the number of blocks of storage and/or the number of unique files (inodes) that can be used by a given user. A “hard” quota limit is a never-to-exceed limit, while a “soft” quota can be temporarily exceeded (using quota(), quotactl(), and quotaon()). The rlimit mechanism supports a large number of process quotas, such as file size, number of child processes, number of open files, and so on. In this case the “soft” limit (also called the current limit) cannot be exceeded, but can be raised to the “hard” limit (also called the upper limit) using setrlimit(). The setrlimit() system call is used in order to set resources limits. It can be used in parallel systems daemons during the creation of a new process by being called after the fork() but before the exec().

The capabilities of Linux and Windows 2000 are compared in Table 2. QNX provides no way to set restrictions on a process or a group of processes.

### 2.1.7 Stdio/stderr Redirection

One of the capabilities required from a computing cluster is to redirect the standard output/error of a job’s processes to the user’s terminal, and to redirect standard input from the user to processes. This is typically done by establishing two sockets for each process, one for stdio and the other for stderr. The module that spawns the processes redirects the stdio/stderr file descriptors of the processes to the established sockets. The I/O handling application can use select() on the established sockets to determine if any new stdout/stderr messages have arrived from one of the processes.

When creating a new process in Windows NT using CreateProcess(), one of the arguments used is a pointer to a STARTUPINFO structure. Among other information, this structure contains handles to standard input, standard output, standard error, and a process creation flags field. When a parent process wants to redirect the stdio/stderr of the child process to predefined stdio/stderr sockets it has to fill up a STARTUPINFO structure with the handles to the sockets, and specify that they be used by setting the appropriate flag.

In Linux, stdio/stderr redirection can be performed using the dup2() system call. This system call duplicates file descriptors, and can be used to duplicate a socket file descriptor and replace the original stdin, stdout, or stderr. This is used to set up the descriptors of a new process between the fork() and exec() calls.

QNX has a shortcut for doing the dup as part of qnx_spawn(). One of the arguments to this function is an array iov that should contain file descriptors 0 through 9. For example, if the value 5 is placed in iov[0], the new process would have its file descriptor 0 (stdin) replaced by file descriptor 5 of the calling process.

|  | Windows 2000 | Linux |
|---|---|---|
| CPU time | user time per process/job in 0.1$\mu$s | CPU time in s |
| number of processes | active processes (and future children) associated with a job | maximum number of processes per user |
| processor affinity | for all processes associated with the job | not supported |
| priority and scheduling | priority class and scheduling class for all the processes associated with the job | not supported |
| memory restrictions | memory limit per-job or per-process | maximum resident set size,<br>maximum data size,<br>maximum stack size,<br>maximum locked in memory |
| GUI restrictions | creating desktops and switching desktops,<br>changing display settings,<br>exiting windows,<br>change system parameters,<br>avoid interaction with windows outside the job,<br>interaction with the clipboard | not supported |
| security restrictions | disallow administrator access,<br>disallow unrestricted token access,<br>force a specific access token,<br>disable certain security identifiers and privileges | user permissions on files (or other devices represented by files) |
| file system restrictions | not supported | number of open file descriptors,<br>maximum core file size,<br>disk quota per user |

Table 2: *Available restrictions in Windows 2000 and Linux.*

### 2.1.8 Process Termination Detection and Error Handling

Once a process is spawned, it's parent has to detect when it exits. The process might exit because it finished its work or because of some unexpected error. It is therefore helpful is the system provides the user with the process exit status and the reason for termination.

In Windows, the way to detect that a child process has exited is to create a thread which polls periodically for the exit status of the child process. This is done with GetExitCodeProcess(). If the process is still running, it returns the value STILL_ACTIVE, otherwise it returns its exit status. No additional information can be retrieved besides the process exit status.

In both Linux and QNX (as POSIX based operating systems) the parent process receives a SIGCHLD signal when a child process dies. The parent must set a signal handler in order to catch the SIGCHLD signal and handle it (using system calls signal() or sigaction()). In the handler one of the wait() family of system calls can be used to determine the exit status and the cause of termination.

### 2.1.9 Deamons

The software architecture of cluster systems often employs daemons — system processes that participate in system administration, rather than running user programs. It is necessary to start these processes when the system is booted. In Unix this is done by the daemon command, which can be placed in a script that is executed upon bootup.

Windows equivalent to a Unix daemon is called a service. A service application conforms to the interface rules of the Service Control Manager (SCM). It can be started automatically at system boot, by a user through the Services control panel applet, or by an application that uses the service functions included in the Win32 API. Services can execute even when no user is logged on to the system [3].

## 2.2 Memory Management

### 2.2.1 Process Virtual Address Space

Many parallel applications require large amounts of memory. To support them cluster nodes are often fitted with lots of physical memory, ranging up to several gigabytes. However, there is a limit to the amount of memory that the operating system can handle.

In Windows (32 bit version) the size of the address space is 4GB ($2^{32}$). The top half of the address space (2GB) is reserved for operating system needs, including kernel and device driver code, I/O buffers, and system tables. Over the years, there has been a large outcry from developers for a larger user-mode address space. Therefore Microsoft has allowed the x86 version of Windows 2000 Advanced Server and Windows 2000 Data Center to increase the user-mode partition to 3GB [35, 37]. In addition, Windows 2000 introduced a new memory management feature called Address Windowing Extensions (AWE) which supports the allocation of more RAM than fits within the process's address space (32 bit), up to 64 GB of memory. The memory blocks are allocated using AllocateUserPhysicalPages(), But these blocks are not visible in the process's address space. The application needs to reserve a region of address space using VirtualAlloc(), which becomes the address window. It can then call MapUserPhysicalPages() to assign one RAM block at a time to the address window [35, 1]. In effect, this is simply support for overlays.

In the x86 architecture, Linux allocates three gigabytes to the process address space. The remaining gigabyte is reserved for memory used by the kernel. The three available gigabytes are split into memory regions used by the process [11].

In QNX 4.0, the user can use all of the available free physical memory in a 4GB (physical) address space, using standard memory allocation functions. While QNX 4.0 supports virtual memory, it does not use swap files for reasons of real-time response/performance. It should be noted that the efficiency of the operating system and Watcom compiler provide relatively small processes in terms of memory requirements.

### 2.2.2 Pinning Memory Pages

In order to improve applications performance, it is sometimes needed to keep the data in physical RAM and reduce disk paging. In clusters, this is also required in order to support the send and receive buffers of user-level communication libraries [18]. On the other hand, locking too many pages into memory may degrade the performance of the system by reducing the available RAM and forcing the system to swap out other critical pages to the paging file.

The Windows VirtualLock() function locks the specified region of the process's virtual address space into physical memory (RAM), ensuring that subsequent access to the region will not incur a page fault. By default, a process can lock a maximum of 30 pages. The default limit is intentionally small to avoid severe performance degradation. Applications that need to lock larger numbers of pages must first call the SetProcessWorkingSetSize() function to increase their minimum and maximum working set sizes. The maximum number of pages that a process can lock is equal to the number of pages in its minimum working set minus a small overhead [3].

In Linux mlockall() disables paging for all pages mapped into the address space of the calling process. This includes the pages of the code, data, and stack segments, as well as shared libraries, user space kernel data, shared memory, and memory mapped files. All mapped pages are guaranteed to be resident in RAM when the mlockall() system call returns successfully, and they are guaranteed to stay there until they are unlocked, the process terminates, or it calls exec(). The mlock() system call locks a specified memory range. Child processes do not inherit page locks across a fork.

Due to real time considerations QNX does not use swap files and therefore locking of virtual memory pages is not needed.

### 2.2.3 Shared Disk Access

Windows NT allows users to specify access permissions for other users to their folders (directories) or disks, by setting up "shares". This is implemented using the SMB (Server Message Block) protocol. Accessing these shares is typically done using the Universal Naming Conventions (UNC), where the server and share are identified in front of the file path (e.g. \\server\share\folder\file). Alternatively, it is also possible to map a share to a single-letter drive identifier. Accessible folders and drives will show up in the "network neighborhood" and "my computer" windows. This methodology is cumbersome for large systems. Windows 2000 introduced Dfs, the distributed file system, that allows administrators to hierarchically connect shares to each other. The end user no longer needs to be aware of the physical location of the files.

In Linux (and all other Unix variants) file systems can be shared among nodes by cross-mounting using the NFS protocols. With NFS, a file system on one node can be grafted onto the local namespace on another. Files in the remote system thereafter appear as local files. This facility, introduced by Sun in the mid ’80s, predates the similar services provided by Windows 2000 by some 15 years. The main limitation on NFS is that the user accounts have to be consistent across the sharing machines. Unix systems can also use SMB via the Samba implementation, but this is typically done only to access files stored on Windows NT servers.

QNX also supports NFS and SMB. In addition, its proprietary file system, Fsys, supports access to remote files just by adding the node number before the filename. For example, /usr/local/bin on node 1 is accessed as //1/usr/local/bin.

## 2.3 Support for SMP nodes

### 2.3.1 Architectural Restrictions

Symmetric multiprocessing (SMP) refers to machines with several processors that share a common main memory and I/O devices. This architecture is commonly used for high-performance servers, and is also useful for the nodes of a cluster. However, it requires special operating system support.

Microsoft offers a few versions of Windows for different market sectors, and each supports different SMP capabilities. Specifically, they recommend that Windows NT Server be used on regular servers, and Windows NT Server Enterprise on cluster systems; this version supports more processors per node. Different versions of Windows 2000 also support different numbers of processors.

Linux SMP support was introduced with kernel version 2.0, and has improved steadily ever since. The kernel locking granularity is much finer in 2.2.x than in 2.0.x, which enables better performance when processes are accessing the kernel. Processes and kernel-threads are distributed among processors. User-space threads are not so distributed, as the operating system is not cognizant of their existence.

QNX 4.0 does not support SMP at all. Only the latest version of QNX for embedded systems, QNX Neutrino 2.0, supports SMP.

Architectural limits on SMP support are compared in Table 3.

### 2.3.2 Processor Affinity

Specifying which processor should run a specific thread or process can improve performance by reducing the number of times the processor cache is reloaded. In clusters it can also be used to control the contention between different parallel jobs. The association between a processor and a thread or a process is called processor affinity.

In Windows, the programmer can define processor affinity for a thread or for a process. SetProcessAffinityMask() specifies the mask of processors on which all the threads of a process are allowed to run. SetThreadAffinityMask() specifies the mask of processors on which the current thread is allowed to run. Windows also provides a weaker level of processor affinity: using Set-

| | Windows | Linux | QNX |
|---|---|---|---|
| max processors | NT 4.0 Workstation: 2<br>NT 4.0 Server: 4<br>NT 4.0 Server Enterprise: 8<br>Win2K Server: 4<br>Win2K Advanced Server: 8<br>Win2K Data Center: 32 | kernel $< 2$: 1<br>kernel 2.0: 16 | QNX 4.0: 1<br>QNX Neutrino 2.0: 8 |
| architectures | Intel<br>Alpha (not supported anymore) | Intel<br>Sparc<br>Alpha<br>Power PC | Intel |

Table 3: *Architectural limits on SMP support.*

ThreadIdealProcessor() the programmer can specify a preferred processor for a thread. The system schedules threads on their preferred processors whenever possible.

In general, Microsoft does not encourage thread affinity: “Setting thread affinity should generally be avoided, because it can interfere with the scheduler’s ability to schedule threads effectively across processors. This can decrease the performance gains produced by parallel processing. An appropriate use of thread affinity is testing each processor” [3]. Instead, they suggest the use of SetThreadIdealProcessor(). However, experimental evidence suggests that when designing parallel algorithms it is actually much better to explicitly assign a thread to each processor [6].

In Linux there is no way to force a process onto specific CPUs, but the Linux scheduler has a processor bias for each process, which tends to keep processes tied to a specific CPU. The Linux community is currently working on a project called “PSET”: Processor Sets for the Linux kernel. The goal of this project is to make a source compatible and functionally equivalent version of PSET (as defined by SGI but partially removed from their IRIX 6.4 kernel) for Linux. This enables users to determine which processor or set of processors a process may run on. The interface is based on the sysmp() system call, Which allows one to specify the binding of a process to a specific CPU, restricting the set of processes that can run on a CPU, and creating sets of processors [30, 5].

As mentioned, QNX 4.0 does not support SMP. QNX Neutrino supports hard processor affinity using the ThreadCtl() system call.

## 2.4 Security

### 2.4.1 Security Model Overview

#### *Windows NT*

Windows NT security is based on access tokens and security descriptors (SD). Every process or thread possesses an access token. When a process is first created, the kernel gives it and its primary thread an access token which contains identifiers that represent the user and any group to which the user belongs. The access token can be passed to other processes as described below. This access token is checked against the SD of an object to determine the permissions that the user has with respect to that object.

An object's security descriptor is essentially an access control list (ACL) that specifies who is and isn't allowed to do things to the object. There are two types of ACLs. The Discretionary ACL is controlled by the owner of an object and specifies the access particular users or groups can have to that object. It contains an access control entry (ACE) for each user, global group, or local group that is either allowed or forbidden to access the object. An SD for an object is initially set to have a DACL with no ACEs, meaning that there is no access for any user. To give access to all users or groups, the DACL for the SD must be explicitly set to NULL. The System ACL is controlled by the system administrator, and allows system-level security to be associated with the object [10].

Whenever a thread requests to create or use another kernel object, it specifies the operations it wishes to perform on that object. The kernel checks the object's SD to see if the requested operations are allowed. If so, then a handle to the object is granted, with only the permissions requested by the thread. When the thread subsequently attempts to perform a certain operation on the object using the handle, the kernel verifies through the permissions attached to the handle that the thread really has the required permissions.

Windows NT and Windows 2000 support C2-level security as defined by the U.S. Department of Defense. Apart from access control as described above, this requires that erased data will not be readable by other programs, that users need to identify themselves, that security events will be audited, and that the system be protected from tampering [2].

### *Linux*

The Linux security model is superficially similar to that of Windows NT: processes have a user ID (UID) that specifies their rights, and filesystem objects have an ACL (other objects, e.g. processes, can only be manipulated by their owners). However, this is a very limited version of an ACL, and only contains three entries: the permissions of the owner of the object, the permissions of members of the owner's group, and the permissions of all others. When a new filesystem object is created, its default access rights are set according to the umask of the creating process.

To allow processes different capabilities, they can actually have several UIDs. The real UID identifies the user on whose behalf the process is running. The effective UID is used for access control checks (see below). Linux, as opposed to other variants of the Unix system, also has a filesystem UID which is used for filesystem access control. Finally, there is the saved UID, which is used to support switching permissions on and off.

UID 0 is a special privileged user (role) traditionally called "root" who can overrule most security checks and is used to administer the system. To allow some splitting of the privileges held by root, POSIX has decreed that processes have three sets of capabilities: the effective, inheritable, and permitted capabilities. This was added to Linux 2.2, but is not universally supported by other Unix-like systems [45].

### *QNX*

As a POSIX compliant operating system, the QNX security model is very similar to Linux. However, it is considered very insecure by many developers in the QNX community because of the fact that once a user (or process) has root permissions on one of the network machines, he has root permissions on all of the network machines.

### 2.4.2 Impersonation

In client-server systems, any client accessing the system through the server might have the same access rights as the server, probably the access rights of the system administrator. Clearly, this kind of access can cause trouble. Impersonation provides a means of limiting the degree of access to that of the client attempting to access the system. In clusters, this is needed when processes are created on behalf of the user on remote nodes.

In Windows, a thread can impersonate a user by receiving that user's access token. The impersonating thread thereafter enjoys the user's access rights, and is prevented from accessing objects that are not allowed for this user. Of particular interest in a cluster environment is the ability to impersonate a client connected to a named pipe, provided the client has given permission for impersonation. This gives the thread almost all of the privileges and abilities of that client. However, it can't subsequently connect to another machine as the client, or create additional processes in the name of the client [10].

In Linux and QNX, impersonation is done by the setuid() system call. Only processes running with root privileges (e.g. daemons) can set their real UID to some chosen user. Another alternative is that programs may allow whoever runs them to impersonate their owner by having the set-UID bit set in their permissions. When such a program is exec'd, the effective UID is copied to the saved UID, and the effective UID becomes the program's owner's UID. This allows the program to access the owner's files on behalf of whoever is running it.

The problem with using setuid() is one of authentication: how does the daemon know that the process requesting it to run as a certain user is trustworthy? One solution is to use low-numbered port numbers, which are by convention reserved for system processes, for the communication among daemons. However, this is inapplicable when receiving the initial request from a remote user process. Another approach is to use a challenge based on the file system security mechanisms, whereby the daemon challenges the requesting process to read the contents of a file that is accessible only to the claimed user [47].

The major drawback in this solution is that it is NFS based, and limits the usage of the system to NFS based configurations. A better alternative is using a secured authentication protocol such as MIT Kerberos. Windows 2000 and also some recent Linux distributions implement the Kerberos v5 authentication protocol, which defines how clients interact with a network authentication service. Clients obtain tickets from the Kerberos Key Distribution Center (KDC), and they present these tickets to servers when connections are established. Kerberos tickets represent the client's network credentials [3].

### 2.4.3 Security Auditing

Windows NT can record a range of event types from a systemwide event, such as a user logging on, to an attempt by a particular user to read a specific file on an NTFS drive. Both successful and unsuccessful attempts to perform an action can be recorded. When an audited event occurs, an entry is added to the Windows NT security log. The security log is viewed by using the Event Viewer application. In the context of clusters, the most important events are probably making connections over the network and indirect object access.

The most common "audit" mechanism currently available on Linux and QNX is the system's logger (syslogd()). The logger enables the operating system and applications to write logging information to the system's log according to their priority. It can be configured to automatically log certain security events such as users logging in.

### 2.4.4 Protecting Access to Cluster Nodes

In cluster environments, it may be necessary to prevent users from running independent processes on the cluster nodes without submitting them through the cluster management software.

In Linux and QNX access to remote nodes is mediated by a set of system daemons (rshd, telnetd, etc.). These daemons can be disabled in order to prevent remote users from using the cluster machines. However, once the remote daemons are disabled, the system administrator might also not be able to remotely administrate the machines. In order to handle this issue, a terminal server can be used. Using the terminal server, the administrator can access the cluster machines remotely using the network.

In Windows running processes on a remote machine is not a standard feature. Special daemons are needed to be installed for this task. So the problem of remote access to the machines is not so relevant in a Windows environment.

## 2.5 Collecting Information

The processes involved in managing a cluster system often have to collect various types of information. Initially the configuration and capabilities of each node must be identified, especially in heterogeneous clusters in which nodes can be used in a dynamic fashion. It is often also necessary to collect resource usage information in order to support resource management functions.

Windows provides various system calls that can be used to determine the system's configuration and resource usage. Examples include GetSystemInfo() for global system information such as the number of CPUs, the processor architecture, level, and revision, and the page size, GetDiskFreeSpace() and GetLogicalDriveStrings() for disk and logical drives information, and GetComputerName() to get the computer's name.

An alternative interface is the Windows registry. In fact, some of the hardware configuration is accessible *only* from the Windows registry. For example, the CPU related information (including CPU speed, manufacturer and revision) is in the registry entry HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System\CentralProcessor\⟨cpu number⟩ [15].

Windows also provides a rich API for creating and accessing various counters associated with system events and performance data. Performance counters can be used for monitoring system resources, application bottlenecks, and program efficiency. Common uses for performance counters are to monitor how much memory an application is using, how badly a computer is paging, and how much CPU time a process is taking. Such counters are used by the Windows NT Performance Monitor tool (PerfMon) [7], which can log the data, send alert messages to the Windows NT event log when a counter exceeds a preset bound, and even run a program when a counter goes over a predefined limit.

Windows performance counters data is stored in the registry. Retrieving the data from the registry and interpreting it requires registry traversals which involves serious programming efforts. This can be eased by Performance Data Helper (PDH) library [17, 7].

Linux also provides system calls to access system information. Examples include sysinfo() which provides information about the system hardware and software (except CPU speed), and vmstat() which provides current load and activity information. An alternative interface is the Linux /proc filesystem. This is an illusionary filesystem that does not exist on a disk. Instead, the kernel creates it in memory as needed. This provides an alternative interface for viewing kernel information, by using the read() system call, instead of a host of other specialized system calls. For example, CPU information can be read from /proc/cpuinfo, and includes model, speed, and other information for each installed CPU. The ps command also reads /proc directly to get information about the state of the system and the running processes [22, 46]. The disadvantages of using /proc is that the data is provided in textual form and has to be parsed, whereas the system calls provide it in predefined structs.

In QNX, resource usage and system configuration information is mostly available via system calls. The important ones are qnx_osstat(), qnx_psinfo() and qnx_info(). qnx_osstat() returns status information of a specified node. The information contains the number of processes ready to execute at each priority level and the average processor load at each priority level. qnx_info() provides information about the system configuration.

## 2.6 Time Measurement and Timers

In cluster systems accurate time measurements and timer events can be useful for job scheduling and for profiling application execution times.

Time measurement is done by the functions QueryPerformanceCounter(), gettimeofday(), and clock_gettime() in Windows, Linux, and QNX, respectively. The resolution of time measurement depends on two things: what is specified in the API, and what is supported in practice. Both Linux and Windows can measure short time intervals down to microseconds resolution, and single microsecond differences do indeed occur when time is measured repeatedly. In QNX the time structure (timepec struct) supports nanoseconds resolution, but the system actually supports only millisecond resolution. In fact, the shortest time interval that can be measured is 10 milliseconds by default. This occurs due to real-time considerations. The system uses a value called “ticksize” to determine the granularity of all software system timers. All time requests will be rounded up to this granularity. For example, if the tick size is 10 milliseconds, a request to wait for 1 millisecond may wait for up to 10 milliseconds [34]. Decreasing the ticksize value to the minimum value of 0.5 millisecond will result in better time measurement granularity. However, changing the ticksize value is not recommended because it will affect all the timers in the system, and might degrade the overall system performance.

System timers can be used for setting periodic events which can be used for scheduling jobs. In the Windows operating system, “regular” timers are associated with windows, and thus with interactive applications; they are set with the SetTimer() function. In addition, there are “waitable” timers, which are created by CreateWaitableTimer() and set with SetWaitableTimer(). These are synchronization objects whose state is set to signaled when the specified due time arrives. There are two types of waitable timers that can be created: manual-reset and synchronization. A timer of either type can also be a periodic timer [3].

In Linux, the system provides each process with three interval timers, each decrementing in a distinct time domain. When a timer expires, a signal is sent to the process, and the timer (potentially) restarts. Timers are set using setitimer(), with one of the three flags: ITIMER_REAL decrements in real time, and delivers SIGALRM upon expiration; ITIMER_VIRTUAL decrements only when the process is executing, and delivers SIGVTALRM upon expiration; and ITIMER_PROF decrements both when the process executes and when the system is executing on behalf of the process. Coupled with ITIMER_VIRTUAL, this timer is usually used to profile the time spent by the application in user and kernel space. SIGPROF is delivered upon expiration.

In QNX, two mechanisms can be used for setting timers. Standard timers are created by timer_create(), and their expiration time is set by timer_settime(). In addition, it is possible to schedule hardware interrupts periodically every 50 milliseconds using qnx_hint_attach().

# 3 Performance Comparison

Performance is one of the most important factors that influence the decision which operating system should be selected for cluster system development. This section presents a performance comparison of Windows, Linux, and QNX, with an emphasis on performance topics relevant for cluster systems software such as process control and networking.

## 3.1 Methodological Issues

In order to measure short time intervals with high precision, a special library called the "Time Stamp Counters library" was used. This library offers a tool for measuring time without the overhead of a system call. It uses a Pentium op-code that reads the Pentium's clock cycles counter, at user level [19]. A good example for the library's added value is QNX time measurements. In QNX, only intervals of 10,000 microseconds can be measured by default. Using the Time Stamp Counters library, intervals as low as 0.16 microseconds could be easily measured.

The hardware and software configuration details of the machines used for the measurements are given in Table 4.

The benchmarks we used for our measurements are available at URL `www.cs.huji.ac.il/~feit/papers/benchmarks.tar.gz`.

## 3.2 Process Life Cycle

Some of the fundamental tasks required from parallel systems are running parallel jobs and scheduling parallel jobs on the cluster nodes. In this section we try to measure the overhead involved in some of the basic process control operations: process creation and termination and process suspension/resumption.

In the measurements presented below results can be distorted by random system events (such as long context switches). These distortions affect the standard deviation of the results. Therefore, in order to minimize the impact of these random events, each test was run 5 times and the test which resulted in the smallest standard deviation is presented here.

| | | |
|---|---|---|
| hardware | model | IBM PC 300GL |
| | processor | Intel Pentium III 500 MHz |
| | memory | 128MB RAM |
| | hard disk | 10GB IDE |
| | network card | Intel 82555 100Base-Tx PHY Pro/100 |
| software | Windows version | NT Server 4.0 SP 5 |
| | services | Alerter, Computer Browser, EventLog, Messager, NAV Alert, License Logging Service, Server, Norton Program Scheduler, Net Logon, NAV Autoprotect, Plug and Play, Protected Storage, RPC Locator, RPC Service, SOFF, Spooler,TaskScheduler,workstation, TCP/IP NetBIOS Helper |
| | Linux version | Linux 2.2.5-22 |
| | daemons | amd, atd, crond, gpm, inet, keytable, linuxconf, netfs, network, nfs, portmap, random, sendmail, site, snmpd, sound, syslog, xfs |
| | QNX version | QNX 4.25 |
| | daemons | nameloc, syslog, portmap, inet, Photon |

Table 4: *System configurations used for measurements.*

### 3.2.1 Process Creation and Termination

This benchmark tries to measure the overhead involved in process creation and termination in each of the compared operating systems. Pseudo code for this benchmark is as follows:

```
for (50 iterations) {
    measure time
    create a process
    wait for the process to die
    measure time and save time interval
}
calculate average time and standard deviation
```

The following system calls were used for process creation: CreateProcess() in Windows, fork() in Linux, and qnx_spawn() in QNX. The spawned process is empty and exits immediately. In order to avoid inconsistent results due to buffer cache hits/misses, one warmup iteration was used for loading the executable into the buffer cache.

As the time to create a process also depends on the size of its data, we used executables ranging in size from 2KB to 10MB. This was achieved by using static allocation and initialization of an array. In addition, we checked a second version in which the process steps once through this array at 4KB increments, to ensure that it is all paged into memory.

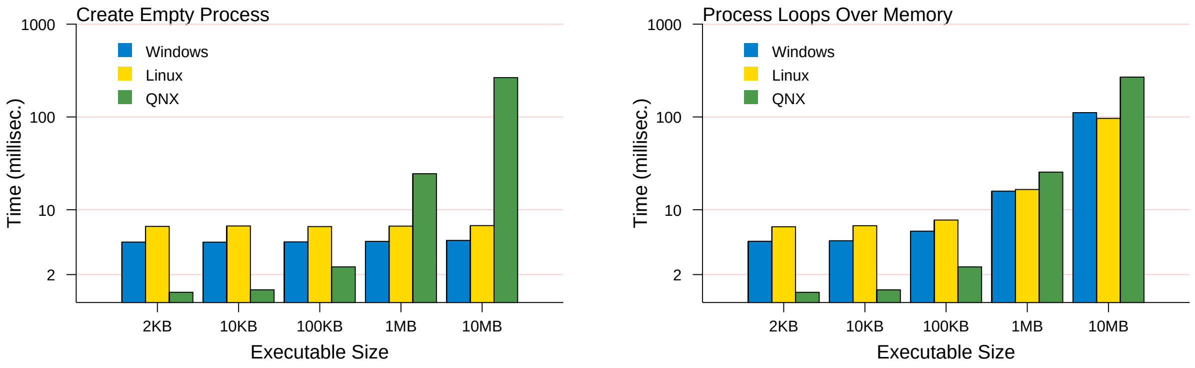

Figure 2: *Process creation and termination times.*

As shown in Fig. 2, Windows and Linux indeed create processes with their memory swapped out, and only perform the real allocation if it is accessed. QNX, on the other hand, always allocates the memory at once when the process is created, in order to avoid paging during execution. Comparing the systems, we can see that although QNX is faster for executable sizes of up to 100KB, from 1MB QNX becomes slower than Windows and Linux.

### 3.2.2 Process Suspension and Resumption

Implementations of gang scheduling on clusters typically perform their scheduling of parallel jobs by suspending all the processes which constitute one job and resuming all the processes of another job. This benchmark tries to evaluate the overhead involved in suspending a process and resuming another. Pseudo code for this benchmark is as follows:

```
turn = 1
for (500 iterations) {
    measure start time
    if (turn == 1) {
        suspend process A
        resume process B
        turn = 0
    } else {
        suspend process B
        resume process A
        turn = 1
    }
    measure end time and save time interval
}
calculate average time and standard deviation
```

The results were that suspending and resuming a process takes a moderate amount of time on all three systems: about 2 microseconds in Linux, 3 microseconds in Windows, and 3.5 in QNX.

## 3.3 Networking

Networking performance is one of the important factors that influence computing clusters performance and distributed applications in general. Since TCP/IP is the most common networking protocol, it was selected for the networking performance comparison. However, we also compare the QNX native protocol performance (FLEET protocol) against QNX TCP/IP performance, and check Linux and Windows UDP performance.

In order to measure TCP/IP performance, a standard network performance benchmark called "Netperf" was used (`www.netperf.org`). Netperf was configured to run a TCP stream test. The stream test sends a stream of TCP packets from one machine to another and measures the bandwidth for various size packets. The measurements were done using two identical machines. The communication medium was a 10MB/s switch shared by other users during the test, or 10MB/s and 100MB/s dedicated hubs.

The major conclusions from these tests were (Fig. 3):

1. When using Windows NT and Linux out of the box, Windows outperforms Linux on 10MB/s connections but Linux outperforms Windows on 100MB/s connections.

2. After manually finding the optimal buffer sizes for a given network configuration and running the test using these buffer sizes, the Linux bandwidth increased significantly on 10MB/s connections and matched Windows. Both improved significantly for 100MB/s connections, with Linux still outperforming Windows by a small margin. This agrees with previously published results for 100MB/s connections between Alpha-based machines [29].

3. QNX does not require any buffer size modifications. Its performance is comparable to Linux and Windows using 10MB/s connection. In 100MB/s connection, the QNX performance is somewhat lower than Windows and Linux.

4. When using optimized buffer size, Linux achieves its peak bandwidth using message sizes smaller than Windows and QNX. For example: when using 10MB/s switch connections Linux bandwidth stabilizes at 4 byte messages vs. 16 byte messages in Windows or 32 byte messages in QNX.

Repeating the measurements with a UDP stream test and 32K buffers led to the following results. For Linux, the peak performance was slightly higher than for TCP, but this was achieved only for large messages of 512 bytes or more. Windows NT achieved similar results for messages of 2KB and higher. However, it suffered from significant system instability for smaller messages, leading to automatic reboots and network connection failures.

FLEET is QSSL's fault tolerant, load balancing LAN protocol built into QNX. It allows message passing between processes on separate nodes and supports multiple LAN cards connected to multiple LANs. Using multiple networks in this way can increase bandwidth and provide fault tolerance. If a cable or network card in one network fails, FLEET automatically reroutes data through another network. This happens on the fly, without involving application software [25].

As shown in Figure 4, QNX FLEET protocol maximum bandwidth is approximately the same as TCP/IP bandwidth. The difference is that TCP/IP protocol reaches it's maximum bandwidth

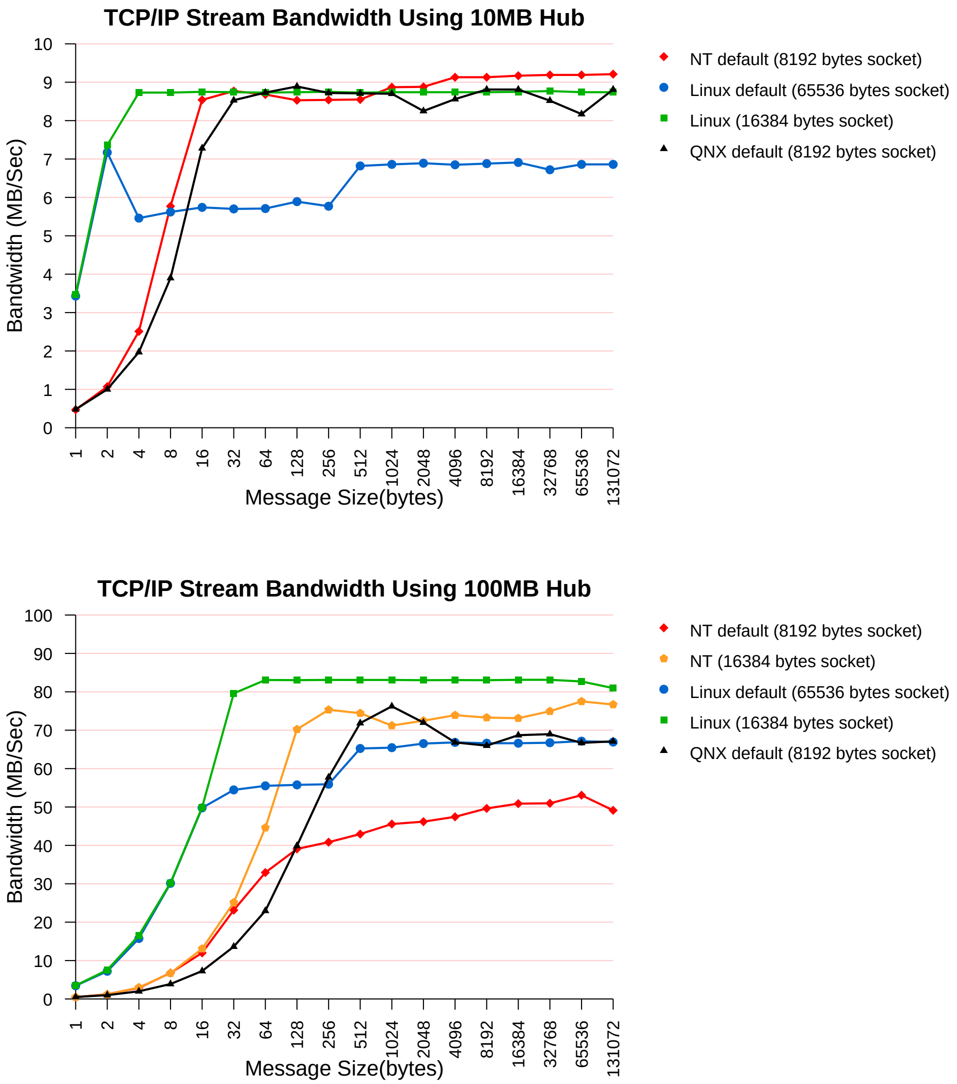

Figure 3: *Bandwidth for TCP/IP stream using Netperf benchmark (10MB/s and 100MB/s hub connections).*

already at 64 byte messages as opposed to FLEET which reaches its maximum bandwidth only for 8192 byte messages (Fig. 4).

## 3.4 System Overheads

The operating system has various overheads that, while not very significant on a desktop, become amplified in a cluster with dozens of nodes. These include the boot and shutdown times, and the memory footprint. The measurements were made using two modes: windows mode (X in Linux, Photon in QNX, and regular Windows NT mode), and console mode where available (Linux and QNX). Results are summarized in Fig. 5.

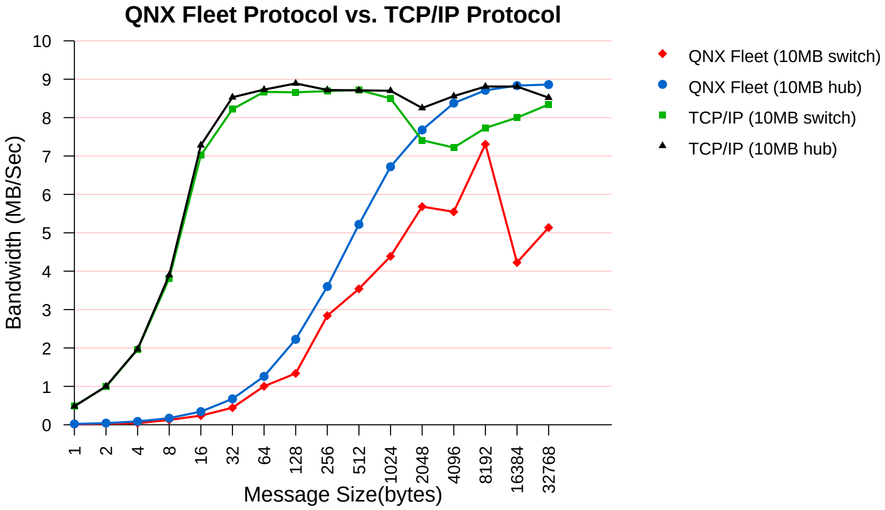

Figure 4: *QNX FLEET protocol vs. TCP/IP protocol.*

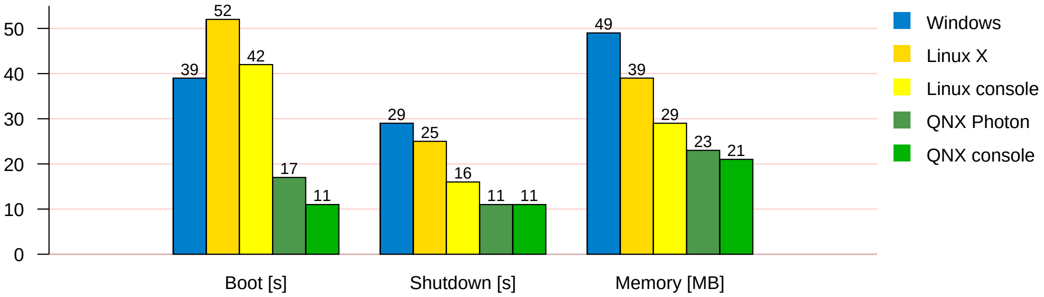

Figure 5: *Boot time, shutdown time, and memory footprint.*

The boot processes is considered as finished when the operating system is ready to log-in users. QNX is the fastest Operating system both to boot and to shutdown. This is due to the fact that QNX is the lightest Operating system in terms of memory footprint. Also, it does not use a swap file that has to be flushed to disk once the operating system is being shut down. Windows boot time is faster than Linux boot time even when Linux is run in console mode. On the other hand, Linux shutdown process is faster than Windows in both configurations.

## 4 Ease of Use

In this section we try to evaluate the ease of use of the compared operating systems. Evaluating operating systems ease of use is not easy, and therefore it discussed in a broad sense. Metrics for evaluation of operating system's ease of use include API calls count, documentation, Web resources count, and ease of administration.

| system | kernel calls | higher API |
|---|---|---|
| Windows NT | 1049 (ntdll.dll) | 1940 (Win32) |
| Linux | 190 | |
| QNX | 21 | 338 (kernel+Posix) |

Table 5: *API calls count.* *(Linux system calls from /usr/include/bits/syscall.h. QNX system calls from /usr/include/sys/kernel.h. Windows system calls retrieved using dumpbin /exports ntdll.dll. Win32 API functions count from list in the MSDN library.)*

## 4.1 API Calls Count

One of the significant factors that influences a programmer's learning curve is the number of API functions needed for implementing a specified programming task. In some of the operating systems, the kernel API (system calls) is enough for developing parallel systems. In others the development should be done using a higher API. Counting the number of functions supported by the lowest usable API can be used as a metric for measuring the ease of development. Fewer functions to learn can be considered better.

In Linux, system calls are basically enough for developing computing clusters. As shown in Table 5, Linux provides the smallest number of of functions needed to be learned in order to develop parallel systems (190 calls). In Windows, the kernel system calls are undocumented and therefore cannot be used for development. Windows exposes the Win32 subsystem API which is built on top of the kernel API. In fact, Win32 is the lowest API level that can be used for developing applications. QNX as a microkernel operating system exposes only 21 kernel functions (see Table 5), the rest of the functions are implemented as separate libraries (POSIX, WATCOM and ANSI).

When trying to understand the variance of the API calls count, we need to take into account the different functionality supported by the API. It is obvious that Win32 offers by far the largest number of services to the developer which can explain the large number of API calls. The flip side of this argument is that Windows is much more complex, and therefore more susceptible to performance and reliability problems. For clusters, a large part of this complexity — e.g. the support of graphical interfaces — is unwarranted.

## 4.2 Standard Documentation

An important factor that influences the ease of development is the standard documentation provided with the operating system. Good documentation is essential for effective software development, especially for the non-expert developers, and should include everything from the simplest commands up to detailed technical articles including code samples. Another parameter for good documentation is the time it takes to find it.

The most comprehensive resource for Windows development documentation is MSDN (Microsoft Developer's Network). It is available both on a Web site (`msdn.microsoft.com`) and on a CD as a part of the Microsoft development environment distribution (Visual Studio). The MSDN Library contains more than a gigabyte of technical programming information, including code samples, documentation, technical articles and the Microsoft Developer Knowledge Base. When trying to compare the quantity and quality of Windows documentation to Linux and QNX documentation, Windows is clearly superior. The MSDN contains a huge number of articles and

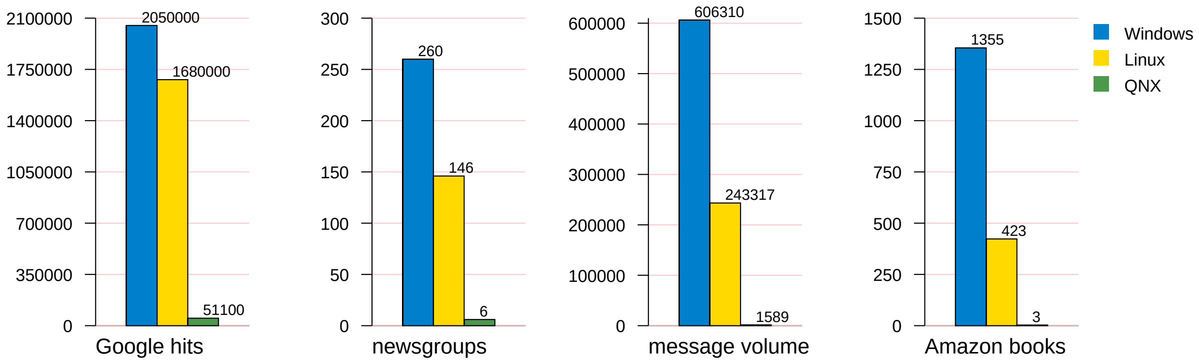

Figure 6: *Results of Web search regarding the three operating systems. (Newsgroups retrieved from the Hebrew University's news server which contains 42,126 newsgroups. Some may not be active. Message volume retrieved from Deja News. Valid as of 25/07/00.)*

code samples.

Linux has two main sources for standard documentation: that provided with the distribution, and the Linux documentation project. Documentation in the distribution includes kernel whitepapers, man pages, and various other miscellaneous documentation (howto, FAQs, tutorials, installed software documentation). In addition, the distribution includes the source code itself, which is the ultimate reference when trying to figure out intricate details of the system's behavior.

The Linux Documentation Project (LDP) (`www.linuxdoc.org`) is working on developing free, high quality documentation for the GNU/Linux operating system. The overall goal of the LDP is to create a canonical set of documentation. Being online (and downloadable), the documentation can be frequently updated in order to stay on top of the many changes in the Linux world. The effort is collaborative with minimal central organization, just like the development of Linux itself. Recently, some of the Linux distributions have started providing the LDP documentation CD along with the operating system installation CD.

In general, the volume of QNX documentation is very low relative to the other two systems. It includes the QNX Helpviewer application for the display of online help, man pages, and manuals, whitepapers, and data sheets located in the QSSL site (`www.qnx.com/literature/`).

## 4.3 Web Resources Availability

Apart from the "official" resources available for each system, development can benefit from help from the on-line community. To get some notion of what resources of this type are available, we tabulate search engine hits, newsgroup activity, and books (Fig. 6).

Search engines can be very useful when looking for documentation, technical articles, online books, and source code samples. We checked the number of hits when looking for "Windows NT"+"Windows 2000", "Linux", and "QNX" keywords in popular search engines. More hits implies more resources and a shorter time until the required information is found. We found that Windows and Linux number of hits is comparable, and in some cases (e.g. AltaVista) there was even a significant advantage for Linux. QNX hit counts, on the other hand, are relatively low.

Newsgroups can be considered as a powerful tool for finding information and solving problems using the Internet community. Finding newsgroups relevant to a specific topic is easier when a large number of newsgroups is available. Windows has almost twice the number of newsgroups as Linux. QNX has only six newsgroups. Similar differences are observed regarding the volume of messages in these newsgroups.

Books about Operating systems and related software are another important resource for users, especially for the non-advanced users who wish to study more about the system. Since the vast majority of the literature is available also in on-line book shops, running queries in the Web sites of these shops can be used to measure the amount of literature available. The number of books about Windows is significantly higher than about Linux in all of the book shops checked. QNX literature is poor by any standard.

## 4.4 Source Code Availability and Extensibility

As mentioned above, an important benefit of operating system source code availability is that it can be used to gain a better understanding of the operating system internals and behavior. But even more important is the option of modifying the operating system. For example, the Mosix system provides a load balancing capability among the nodes of a cluster, based on a process migration facility [8]. This is implemented within the kernel of the base system, which is Linux. Likewise, the initial development of Beowulf hinged on kernel modifications that allowed several physical networks to be used as a single logical network — a technique known as “channel bonding” [38] (although later availability of faster networks reduced its importance).

In Linux, all the source code is available under the terms of the GNU General Public License. Both Windows and QNX are commercial operating systems and their source code is unavailable for the public. However, Microsoft is cooperating with academic research and has shared Windows source code with a few research institutes.

A separate issue is extending the kernel with added functionality. Linux allows kernel modules, including drivers, to be loaded and unloaded while the system is operational [16]. This obviously has significant advantages in a server that strives for continuous availability, but also in a cluster serving multiple parallel applications. In Windows, every such modification requires a reboot to take effect. However, these modifications typically occur only during setup, and not during production.

## 4.5 Software Availability

A separate issue is the ability of software that works with the different systems. Two important types of software are device drivers and parallel programming environments, including application libraries and cluster management software.

Due to the dominant market share enjoyed by Windows, nearly all device manufacturers write device drivers for Windows. In addition, the Microsoft Device Drivers Development Toolkit eases the development of new drivers by providing various templates [4]. Device drivers for Linux are less plentiful, but drivers for popular hardware can often be found on the Internet even if they were not provided by the hardware manufacturer, and extensive documentation to help in the development of drivers is available [36]. QNX has very few device drivers available by comparison.

Libraries that are important for high-performance cluster computing include MPI for message passing and ScaLAPACK for linear algebra. The most common implementation of MPI is MPICH, which is available for both Unix and Windows platforms (www-unix.mcs.anl.gov/mpi/mpich/). As MPICH has a version that runs over TCP/IP, it can be ported to QNX as well with relative ease. ScaLAPACK is written above MPI, so any machine that supports MPI automatically supports ScaLAPACK as well (www.netlib.org/scalapack/). Other, more specialized libraries, may be expected to be available for Unix systems and probably also for Windows, but maybe not for QNX.

Popular cluster control systems include Platform Computing's LSF (Load Sharing Facility, www.platform.com/platform/platform.nsf/webpage/LSF?OpenDocument), Varidian's PBS (Portable Batch System, www.pbspro.com/), Codine, and the Maui Scheduler (which provides policy decisions more than actual control over processes; see supercluster.org/ [26]). These were typically developed on Unix systems, and are all available for Linux. Some have also been ported to Windows (e.g. LSF), and there are also some non-commercial research environments that use Windows. However, Unix (and especially Linux) remains the dominant platform for which cluster management systems are developed. As far as we know, they are not available on QNX, but a port to QNX should be possible due to QNX's support of POSIX.

## 4.6 Administration

Windows NT uses administration tools based on graphical user interfaces. This has the well-known advantages of being able to provide hints, context-sensitive help, and certain consistency checks. Windows 2000, on the other hand, also provides command-line capabilities for many administrative tools. In Unix (including Linux and QNX) most of the tools used to manage the system are command-line based, although some tools also have graphical interfaces [13].

In the context of clusters command-line interfaces are more robust in the sense that it is usually possible to access a remote machine via some text terminal, but not always via a graphical interface. Also, text-based programs are easier to automate so administrators can make the same changes on several machines using a single script. Unix comes with various scripting facilities such as the shell and Perl. Windows does not. Moreover, Unix uses textual configuration files that are easy to edit and reproduce, whereas Windows registry uses a proprietary binary format that is created by the system. It can be copied en-masse, but not manipulated.

One of the most attractive features of Unix is the ability to control machines remotely. Since Linux and QNX come with a telnet daemon built in, administrators can telnet virtually any machine running the telnet daemon, regardless of operating system, to do all administrative tasks. To remotely administer a Windows server, you must purchase a separate application to allow remote control. Two of the most popular remote administration products for this are PC Anywhere by Symantec and SMS (Systems Management Solution) from Microsoft. Using commercial administration tools can become costly because you must purchase a copy for the server to act as the host, and a copy for each computer that needs to remotely control the server. VNC from AT&T Research is also an excellent product, which is available for free (www.uk.research.att.com/vnc/). It allows desktops to be viewed across different platforms, including Windows and Unix.

# 5 Conclusions

Our comparison of Windows NT, Linux, and QNX as the basis for building cluster systems uncovered a host of differences among these systems, but no clear winner. Each system supports some important features that do not exist in the others. For example, Windows NT can create suspended processes, which Linux cannot, and has better support for impersonation. Linux, on the other hand, can easily suspend a process, and supports disk quotas. While the lack of such features can cause headaches, they are typically not show-stoppers.

Likewise, contrary to the common belief that Linux outperforms Windows, and the expectation that QNX would outperform both, we found no clear-cut advantage for one system over the other. This agrees with the results of Lancaster and Takeda, which indicate that the compiler is more important than the base system in terms of performance [29].

Thus it seems we can say that all three systems provide similar support for the construction of clusters. We verified this claim by implementing a reduced version of ParPar on all three of them, using the same basic architecture. Overall, the implementation are rather similar and required commensurate effort. In fact, they were so similar that we decided it was pointless to try and measure the differences between them in terms of code complexity or other software metrics.

While our results indicate that the systems are similar in-the-large, it may be that the differences are amplified in certain niches. For example, familiarity with Unix may tip the scale in favor of Linux for scientific computing, whereas the availability of commercial software or drivers can promote Windows for other applications. In fact, our findings indicate that such considerations should indeed dominate the choice of which base system to use.

A somewhat indirect metric that nevertheless provides important insights is the number of cluster installations that actually use the different systems. This can be checked using the clusters sub-list of the Top500 project (`clusters.top500.org/`). Of the top 100 clusters listed, 88 are based on Linux, and an additional 7 on other variants of Unix. Only 2 are based on Windows NT. None are based on QNX. Granted, this data is incomplete, and it may be a result of historical preference rather than technical merit alone, but the disparity is very striking.

Of course, the comparison done in this research could be extended to other topics. The benchmarks written could be improved, and new ones can be written for comparing other topics. One of the most important aspects which was not covered in this research was operating system stability. Determining how stable the operating system is for a long period of time under different types of load is not an easy task and can be used as a subject for an independent research. Alternatively, it is possible to try and glean some information from other comparisons, especially between Windows NT and Linux [27, 31, 14]. However, it should be remembered that such comparisons — especially those published by companies that have an interest in the results — may be tainted by the fact that they have huge economic implications. In addition, they focus on the server market which is only partly relevant to high-performance computing clusters. Finally, it should be remembered that the operating system is just one player in creating a usable cluster. Application development tools are also very important. Thus there is reason to survey the parallel application development environments that are available for each of the systems.

## Acknowledgements

This work was supported in part by the Israel Science Foundation and by the Ministry of Science.

# References

[1] “*Address windowing extensions and Microsoft Windows 2000 DataCenter Server*”. URL http://msdn.microsoft.com/library/backgrnd/html/awewindata.htm, Mar 1999.

[2] “*C2 level security*”. Platform SDK documentation, MSDN, Jan 2000.

[3] “*Microsoft Developer Network*”. Jan 2000.

[4] “*Microsoft windows driver development kits*”. URL http://www.microsoft.com/DDK/.

[5] “*Pset - Processor Sets for Linux/SMP*”. URL http://isunix.it.ilstu.edu/˜thockin/pset/.

[6] B. Alkire, “*Parallel computation of Hessian matrices under Microsoft Windows NT*”. *SIAM News* **31(10)**, Dec 1998.

[7] R. Anderson, “*Finding leaks and bottlenecks with a Windows NT PerfMon COM object*”. URL http://msdn.microsoft.com/library/techart/perfmon.htm, Jan 1999.

[8] A. Barak, O. La’adan, and A. Shiloh, “*Scalable cluster computing with MOSIX for Linux*”. In *Linux Expo*, pp. 95–100, May 1999.

[9] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner, *Linux Kernel Internals.* Addison-Wesley, 2nd ed., 1998.

[10] P. Bosch, “*Windows NT security system basics*”. URL http://msdn.microsoft.com/library/periodic/period97/vc0197.htm, 1997.

[11] R. Card, E. Dumas, and F. Mével, *The Linux Kernel Book*. Wiley, 1998.

[12] A. Chien, S. Pakin, M. Lauria, M. Buchanan, K. Hane, L. Giannini, and J. Prusakova, “*High performance virtual machines (HPVM): clusters with supercomputing APIs and performance*”. In 8th *SIAM Conf. Parallel Processing for Scientific Comput.*, Mar 1997.

[13] Q. P. Coldiron, “*Replacing Windows NT Server with Linux*”. URL http://citv.unl.edu/linux/LinuxPresentation.html, 1997.

[14] D. H. Brown Associates, Inc., “*Operating system scorecard*”. URL http://www.dhbrown.com/dhbrown/opsys.cfm.

[15] T. Daniels, *1001 Secrets for Windows NT Registry*. 29th Street Press, 1997.

[16] J-M. de Goyeneche and E. A. F. de Sousa, “*Loadable kernel modules*”. *IEEE Softw.* **16(1)**, pp. 65–71, Jan/Feb 1999.

[17] A. Denver, “*Using the performance data helper library*”. URL http://msdn.microsoft.com/library/techart/msdn_pdhlib.htm, Mar 1997.

[18] P. Druschel, "*Operating system support for high-speed communication*". *Comm. ACM* **39(9)**, pp. 41–51, Sep 1996.

[19] Y. Etsion and D. G. Feitelson, *Time Stamp Counters Library - Measurements with Nano Seconds Resolution*. Technical Report 2000-36, Inst. Computer Science, The Hebrew University of Jerusalem, Aug 2000.

[20] D. G. Feitelson, A. Batat, G. Benhanokh, D. Er-El, Y. Etsion, A. Kavas, T. Klainer, U. Lublin, and M. A. Volovic, "*The ParPar system: a software MPP*". In *High Performance Cluster Computing, Vol. 1: Architectures and Systems*, R. Buyya (ed.), pp. 754–770, Prentice-Hall, 1999.

[21] D. G. Feitelson, Y. Ben-Asher, M. Ben Ezra, I. Exman, L. Picherski, L. Rudolph, and D. Zernik, "*Issues in run-time support for tightly-coupled parallel processing*". In 3rd *Symp. Experiences with Distributed & Multiprocessor Syst.*, pp. 27–42, USENIX, Mar 1992.

[22] J. Fink, "*An overview of the proc filesystem*". *Linux Gazette* **46**, Oct 1999. URL http://www.linuxgazette.com/issue46/fink.html.

[23] R. Friedman, M. Goldin, A. Itzkovitz, and A. Schuster, "*Millipede: easy parallel programming in available distributed environments*". *Software — Pract. & Exp.* **27(8)**, pp. 929–965, Aug 1997.

[24] D. P. Ghormley, D. Petrou, S. H. Rodrigues, A. M. Vahdat, and T. E. Anderson, "*GLUnix: a global layer Unix for a network of workstations*". *Software — Pract. & Exp.* **28(9)**, pp. 929–961, Jul 1998.

[25] D. Hildebrand, "*An architectural overview of QNX*". URL http://www.qnx.com/literature/whitepapers/archoverview.html.

[26] D. Jackson, Q. Snell, and M. Clement, "*Core algorithms of the Maui scheduler*". In *Job Scheduling Strategies for Parallel Processing*, Jun 2001.

[27] J. Kirch, "*Microsoft Windows NT Server 4.0 versus UNIX*". URL http://www.unix-vs-nt.org/kirch/, Aug 1999.

[28] R. Krten, *Getting Started with QNX 4*. Parse Software Devices, 1998.

[29] D. Lancaster and K. Taked, "*Comparative performance of a commodity Alpha cluster running Linux and Windows NT*". In *IEEE Workshop Cluster Comput.*, May 1999.

[30] D. Mentré, "*Linux SMP HOWTO*". URL http://www.phy.duke.edu/brahma/smp-faq/, Sep 1999.

[31] Microsoft Corp., "*Linux myths*". URL http://www.microsoft.com/ntserver/nts/news/msnw/LinuxMyths.asp, Oct 1999.

[32] Microsoft Corp., "*Moving Unix applications to Windows*". MSDN, Jan 2000.

[33] G. F. Pfister, "*Clusters of computers for commercial processing: the invisible architecture*". *IEEE Parallel & Distributed Technology* **4(3)**, pp. 12–14, Fall 1996.

[34] QNX Software Systems, *QNX Operating System Utilities Reference*. QSSL, 1998.

[35] J. Richter, *Programming Applications for Microsoft Windows*. Microsoft Press, 4th ed., 1999.

[36] A. Rubini, *Linux Device Drivers*. O'Reilly, 1998.

[37] M. Russinovich, "*NT vs. UNIX: is one substantially better*". *Windows NT Magazine*, Dec 1998. URL http://www.winntmag.com/Articles/Index.cfm?ArticleID=4500.

[38] D. F. Savarese and T. Sterling, "*Beowulf*". In *High Performance Cluster Computing, Vol. 1: Architecture and Systems*, R. Buyya (ed.), pp. 625–645, Prentice Hall, 1999.

[39] R. J. Simon, *Windows NT WIN32 API Superbible.* Waite Group Press, 1998.

[40] D. S. Solomon, *Inside Windown NT*. Microsoft Press, 2nd ed., 1998.

[41] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Parker, "*Beowulf: a parallel workstation for scientific computation*". In *Intl. Conf. Parallel Processing*, vol. I, pp. 11–14, Aug 1995.

[42] W. R. Stevens, *Advanced Programming in the Unix Environment.* Addison Wesley, 1993.

[43] A. S. Tanenbaum, "*A comparison of three microkernels*". *J. Supercomput.* **9(1/2)**, pp. 7–22, 1995.

[44] P. Thurrott, "*Windows XP beta 2 review*". URL http://www.winsupersite.com/reviews/windowsxp_beta2.asp.

[45] D. A. Wheeler, "*Secure programming for Linux and Unix HOWTO*". URL http://www.dwheeler.com/secure-programs/.

[46] L. Wirzenius, "*The linux system administrators' guide, version 0.6.2*". URL http://www.linuxdoc.org/LDP/sag/book1.html.

[47] S. Zhou, X. Zheng, J. Wang, and P. Delisle, "*Utopia: a load sharing facility for large, heterogeneous distributed computer systems*". *Software — Pract. & Exp.* **23(12)**, pp. 1305–1336, Dec 1993.