Performance Evaluation of Commodity Systems for Cluster Computing

Imed Chihi <imed.chihi@ensi.rnu.tn>

December 4, 2003

Abstract

The rise of the Personal Computer has dramatically changed the modern computer scene. Along with the proliferation of commodity networks, a new breed of computing was born: Parallel Distributed Processing. The large distribution of commodity components caused a huge drop in prices even for some high end machines. Some research work explored the case of using a collection of machines linked by a network as one single processing facility, one major outcome of these tendencies was the Scientific Computing Cluster. The emergence of free, low cost and fast operating systems like Linux and FreeBSD has accelerated these tendencies.
This work attempts to explore the capabilities of a single processing node. A special emphasis is put on I/O sub-systems for we believe they are the determinant metrics for the performance of the modern scientific workloads. We study the case of the Intel x86 machines, Ethernet networks and the Linux operating systems: the *de facto* commodity components. We present a short summary of some related research and industry efforts that deal closely with our topic, then we present some architectural details that would help in the appreciation of later measurements, next we present the operating system as the major software component driving a cluster. A couple of chapters are dedicated to some in-depth study of the storage and network IO subsystems with interesting findings.

Keywords

Operating systems, Clusters, High-performance computing, Unix, Linux, Benchmark, Scientific computing, File system, Zero-copy.

Résumé

La popularité de l'ordinateur individuel a profondément changé la paysage informatique moderne. En outre, avec la prolifération des réseaux bon marché, un nouveau type de traitement a vu le jour: le calcul parallele distribué. La grande quantité des ventes des comosants a réduit les coûts même pour certains équipements haut de gamme. Certains traveaux de recherche ont exploré la possibilité d'utiliser un ensemble de machines comme un seul noeud de calcul, et l'émergence de systèmes d'exploitation libres et gratuits comme Linux et FreeBSD a accéléré cette tendence.
Ce travail essaye d'étudier les capacités d'un noeud de calcul, le point est mis sur le sous-système des entrées/sorties puisque les applications scientifiques manipulent souvent de gros volumes de données. Nous étudions l'architecture Intel x86, les réseaux Ethernet et Linux: les composantes bon marché de fait. Nous présentons un bref aperçu des traveaux qui relatent à notre sujet, puis nous présentons quelques détails sur l'architecture en question qui faciliterons l'appréciation des mesures faites. Nous introduisons le système d'exploitation comme la composante logicielle majeure tournant un cluster. Deux chapitres sont dédiés à une étude des systèmes de stockage et des communications réseaux avec des résultats intéressants.

Mots clés

Systèmes d'exploitation, Clusters, Calcul à haute performance, Unix, Linux, Benchmark, Calcul scientifique, Système de fichier, Zero-copy.

Contents

1 Related Work **10**
1.1 Distributed computing . . . 11
1.1.1 Job schedulers . . . 11
1.1.2 Process checkpointing and process migration . . . 11
1.1.3 Distributed and parallel programming environments . . . 11
1.2 Micro kernels . . . 12
1.3 Unix enhancements . . . 12
1.3.1 File systems . . . 12
1.3.2 In-kernel processes . . . 13
1.3.3 Zero-copy networking . . . 13
1.3.4 Asynchronous I/O . . . 14
1.3.5 Raw I/O . . . 14
1.3.6 Vector I/O . . . 14
1.3.7 Channel bonding . . . 14
1.4 Conclusion . . . 15

2 Architectural Overview **16**
2.1 The low-end computer system . . . 16
2.1.1 The CPU . . . 16
2.1.2 The memory . . . 17
2.1.3 The input/output . . . 18
2.1.4 The address spaces . . . 18
2.2 Components interconnection . . . 19
2.2.1 The bus . . . 19
2.2.2 Programmed IO . . . 21
2.2.3 Interrupts . . . 21
2.2.4 Memory-mapped IO . . . 21
2.2.5 Direct Memory Access . . . 21
2.3 Today's technologies . . . 22
2.3.1 The CPU . . . 22

2.3.2 The memory . . . 25
2.3.3 The bus . . . 25
2.3.4 The storage devices . . . 27
2.3.5 Networking . . . 28
2.4 Conclusions . . . 29

3 The Operating System **31**
3.1 Architecture . . . 31
3.2 Virtual memory . . . 32
3.3 Kernel space and user space . . . 33
3.4 Kernel performance . . . 34
3.4.1 System calls, file handling and signaling . . . 34
3.4.2 Process spawning . . . 36
3.4.3 Context switching . . . 37
3.4.4 Inter-process communications . . . 39
3.4.5 File IO and virtual memory . . . 41
3.4.6 libc code . . . 42
3.5 Conclusions . . . 42

4 Interconnecting Nodes **45**
4.1 Introduction . . . 45
4.2 Dissecting network latency . . . 46
4.3 Transport Overhead . . . 47
4.4 Zero copy network IO . . . 49
4.5 Conclusion . . . 50

5 The Storage Sub-system **51**
5.1 The file system — part of the kernel . . . 51
5.2 RAID 0 . . . 52
5.3 NFS — the Network File system . . . 54
5.3.1 User space NFS . . . 54
5.3.2 Kernel space NFS — Shortening the critical path . . . 56
5.4 Zero copy.. again . . . 56
5.5 Conclusion . . . 59

A CPU Optimizations **63**

B An Example Supercomputer: Intel Paragon **65**

C Source code **67**

C.1 The pf_packet benchmark 67

C.1.1 The sender (rp_send.c) 67

C.1.2 The receiver (rp_receive.c) 69

C.2 The bonnie-mmap patch 70

D The test environment **72**

D.1 lmbench tests 72

D.2 GFS and RAID tests 72

D.3 Raw Ethernet RTT 73

D.4 kNFSd tests 73

D.5 Bonnie tests 73

List of Figures

2.1 The memory hierarchy. 17
2.2 Illustration of address spaces. 19
2.3 Bus hierarchy. 20
2.4 Composition of a typical PC. 22
2.5 Integer performance of various processors compared [6]. 24
2.6 Floating point performance of various processors compared [6]. 24

3.1 The UNIX kernel architecture 32
3.2 Virtual memory management 32
3.3 A system call path in the Unix kernel. 33
3.4 System calls and file handling latencies 35
3.5 Signal handler installation and signal catching latencies 36
3.6 Process spawning latencies 37
3.7 Context switching times between communicating processes 38
3.8 Inter-process communication latencies 40
3.9 File system creation latencies 41
3.10 Hand-coded bcopy() bandwidth 42
3.11 bcopy() bandwidth through libc 43

4.1 Network latency. 46
4.2 TCP and raw Ethernet round-trip times 48

5.1 Throughput of different file systems and RAID arrays compared 52
5.2 CPU usage of different file systems and RAID arrays compared 53
5.3 Response time of user-space NFS vs. in-kernel NFS [20] 55
5.4 Throughput of user-space NFS vs. in-kernel NFS [20] 56
5.5 Memory mapped files for a Linux shell (/bin/bash) 57
5.6 Bonnie results with and without mmap() 58
5.7 mmap() latencies 59

B.1 Architecture of a node in Intel Paragon. 66

B.2 Paragon interconnect. 66

List of Tables

2.1 CISC and RISC approaches compared. 17
2.2 Popular bus technologies compared. 25
2.3 SCSI throughput. 28

4.1 Cost of send and receive operations per word on Gigabit Ethernet 49

D.1 Tested combinations of hardware and operating systems 72

Introduction

Computer technology has evolved toward a large set of interconnected systems. This computing scheme, very different from the centralized approach, has led software manufacturers to fundamentally new ways of designing applications. One of the most noticeable effects of these new trends is that the **network** has become a master piece of a computer system just like the processor or the memory, almost all modern applications make intensive use of this resource, and hence *The Network Is The Computer*[1].

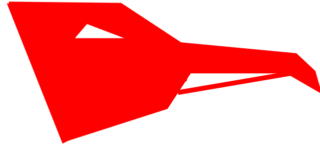
Distributed processing is becoming a common term in today's software industry. It is, to make a long story short, the slicing of an application into multiple pieces running on distinct nodes that cooperate in order to perform a given task. The low costs of personal computer networks along with the relatively good performance of the networking and the processing hardware caused the rise of a new kind of distributed processing: *Distributed Parallel Processing*. For so long, parallel processing was a holy field dedicated to custom hardware. Today's scientific and engineering applications require tremendous computing power and do manipulate huge amounts of data. Having the ability to perform such processing without making enormous investments will help researchers and engineers do better work.

Distributed parallel applications usually run over *Computer Cluster*s, which are a collection of commodity (low end) systems interconnected by a (commodity) network where each node is running an (commodity) operating system. The big question with this new approach is: by piling together many low end systems, would it be possible to achieve an overall processing performance that's comparable to supercomputers? The answer strongly depends upon too many factors, this work focuses on the study of the capabilities of a single processing node, that is the computing and communication performance of a node running a given operating system.

The first chapter presents the major research projects and technologies with direct connection to our topic, the second presents the PC architecture and unveils the performance factors that might be critical to parallel applications. The third chapter introduces the operating system as the major software driving the network and the applications, chapter four discusses the computer networking issues and, just like the

[1] Trade mark of Sun Microsystems.

second chapter, it shows what factors might affect the performance of parallel applications. Chapter five presents the storage subsystem for it is a critical factor when it comes to moving large amounts of data.



Chapter 1

Related Work

Many computer systems research works have focused on performance for both scientific and enterprise needs. The objective being to get the best performance out of a network of commodity computers, some research work have focused on redesigning and reimplementing the operating system running per node based on the argument that Unix was not designed for the needs of high-performance computing, and, therefore, running it on a node would introduce too much overhead. A major trend emerged from these theories: micro-kernels which are totally different operating systems designed to reduce the overhead introduced by the operating system's mechanisms to the lowest.

Another category of research work focused on introducing some Unix enhancements by changing some mechanisms to enhance performance. This trend uses the argument that creating another operating system would be too disruptive and cannot gain wide popularity, the performance enhancements were done mainly in the network and disk IO subsystems [24, 28, 23, 4], some work was done to facilitate program development on computer clusters by extending some Unix semantics like the *Process ID* (PID) [29] or disk file system mounts [30].

A third big trend used a very different approach: add some software layer to different operating systems to facilitate application development and avoid changing existing operating systems because that would introduce incompatibilities in the case of heterogeneous clusters, standardized message passing libraries like MPI ([37]) and PVM ([36]) fall in this category.

"Beowulf" is becoming a popular term in today's cluster computing. It's a collection of software aimed at building scientific computing clusters around Linux, it includes software for Linux kernel and libraries optimizations, device drivers, packaged PVM and MPI toolkits, node management software and many other utilities [41].

1.1 Distributed computing

Many terms (*grid computing*, *multicomputer*, *computer clusters, network of workstation*s, *pile of PCs*, etc.) were coined by the computer industry to refer to a very large variety of technologies to cooperatively perform computing tasks by interconnected machines. The concepts originated in the research on multi-computer process scheduling, distributed systems and parallel programming. Cluster computing describes local area networks of computer systems dedicated to perform high-performance processing, usually scientific computing. It does also refer to high-availability systems used in critical applications, but this will remain beyond the scope of this work.

1.1.1 Job schedulers

Job scheduling was a hot research topic in the early nineties. It consists into optimizing the scheduling of batch and interactive workloads by launching a job on the most appropriate host in the network, the "most appropriate host" can be the one with the lowest relative load. Namely: Condor [31] at Wisconsin, Madison University, Distributed Queuing System (DQS) [32] at Florida State University, LoadLeveler [33] from IBM, Portable Batch System (PBS) [34] from OpenPBS.org, Codine from Genias Software, etc.

1.1.2 Process checkpointing and process migration

Some job scheduling systems offered the ability to dynamically adjust load. Apart from scheduling hosts where to launch applications, the job scheduler can relocate a running process from its host. *Process checkpointing* consists into getting an exhaustive image of the state of a Unix process in order to reconstruct it later on (eventually) a different similar host. A few systems implemented Unix process checkpointing, the Condor Project at the Wisconsin University has probably pioneered this implementation [35]. Process migration is one of the useful uses of process checkpointing: once a process is checkpointed, it is relocated ("migrated") to a different host and reconstructed there, execution should resume where it stopped on the first host.

1.1.3 Distributed and parallel programming environments

A few tools were developed to support application development in distributed environments or to create a service layer atop of legacy systems to provide a parallel processing facility. Java is certainly one of the most wide spread distributed programming environments. The OpenGroup, a consortium of Unix vendors, created the *Distributed Computing Environment* (DCE) which is a set of software tools to support distributed applications on legacy Unix infrastructures.

The *Parallel Virtual Machine* (PVM) [36] and the *Message Passing Interface* (MPI) [37] are very widely used parallel programming libraries. They were initially designed to alleviate the huge burden of porting parallel applications between custom supercomputers, a task so complex that complete rewrites were often needed. PVM and MPI provide the basic functions to create and manage processes on a cluster of homogeneous or heterogeneous systems, they have facilities for data transfer between distant processes and a cluster-wide process naming.

1.2 Micro kernels

A few years ago, distributed systems were a very hot research topic and many projects devoted a lot of effort to writing entire distributed operating systems and development tools. The Mach [38] micro-kernel operating system is probably the best known distributed operating system, it started as a project at Carnegie Mellon University and then moved to Utah University. Mach was designed with the intent to have all the subsystems spread over a network: memory management can be distributed and processes may allocate memory anywhere in the network. Inter-Process Communication is done in-kernel and remote processes can communicate in a seamless fashion. Amoeba [39] from Vrije Universiteit of Amsterdam and Chorus from Sun Microsystems are equally popular distributed operating systems with many design differences.

The theory behind micro kernels design is very elegant but the implementation faced tremendous challenges getting a system to perform reasonably fast. For instance, the TCP/IP implementation of Mach had to be included in the kernel which is against the very nature of micro kernels. Application compatibility caused another array of problems and Mach had to run a Unix emulation layer by implementing the BSD operating system in user space.

1.3 Unix enhancements

The design of the Unix operating system was not geared toward high-performance processing in particular. The major design goals were simplicity and portability. As hardware got better and faster, Unix implementations started to include modified I/O semantics and new system calls.

1.3.1 File systems

A lot of work was dedicated to designing high-performance file systems because Unix relies on file systems for many of its subsystems, file systems are also very solicited in intensive I/O operations. The *Parallel Virtual File System* (PVFS) is a project at Clemson University, USA which is trying to bring a parallel file system, like those

available on commercial supercomputers, to PC clusters. It has the capability of making a file system span multiple nodes and striping files across them. Using the Linux *Network Block Device* (NBD) one can perform striped I/O across multiple hosts but at a low (block) level, NBD doesn't provide file system semantics and, therefore, cannot provide data integrity guarantees in concurrent access.

The *Global File System* (GFS) [30] started as a research project at University of Minnesota, USA, now, a company (Sistina), is offering it as a commercial product with technical support. GFS is a shared-disk file system: multiple nodes can mount the same device containing a file system and perform concurrent read and write accesses on it. It is designed to use any low-level device sharing mechanism like shared (multi-ported) SCSI, Fibre Channel and Linux NBD. It does journaling and striping over multiple devices and it can take advantage of DMEP-capable devices and perform locking on-disk. Therefore a lock server is not needed. GFS has been deployed in real-world applications with success and is heading to become a major cluster file system. The GFS code was initially released as Open Source, but lately Sistina changed the licensing for the more recent versions, a "free" branch of the GFS code was forked as Project OpenGFS to carry on the development using the old free code.

1.3.2 In-kernel processes

The user-kernel separation in Unix is meant to be a protection mechanism to ensure a safe execution environment. Some research work tried to eliminate this separation by running user code in kernel space and, hence, avoiding system calls [46]. System call invocation becomes a simple funtion call. *Kernel Mode Linux* (KML)[46] is a Linux kernel patch that adds some extensions to run user processes in kernel space, the authors report an acceleration by 30 times in the execution of getpid() in KML by opposition to the traditional interrupt-based invocation of system calls.

1.3.3 Zero-copy networking

Normal Unix network I/O happens in too stages: a copy from user-space (the application's address space) to kernel-space, then a copy from that kernel-space to the device's I/O address space which results into the actual transmission. Zero-copy networking is a technique to eliminate the first step and save a memory copy operation, on certain hardware configurations this may yield to substantial network I/O speed up. Major work include the Trapeze Project at Duke University and Solaris Zero-Copy TCP at Sun Labs [24], the Linux kernel developers included a complete zero-copy framework into kernel version 2.5. The U-Net project, initially at Cornell University, has implemented a zero-copy network I/O for Linux on selected ATM and Ethernet devices [23]. The *Virtual Interface Architecture* was born as a U-net-based specification started by an

alliance between Intel, Compaq and Microsoft [42], a few hardware vendors have manufactured VIA-compatible interfaces. U-net and VIA provide a user-level access from applications to the network device. Some modern NIC can perform some high-level networking tasks in hardware like TCP checksum calculation and *TCP Segmentation Offloading* (TSO).

1.3.4 Asynchronous I/O

Normal Unix (character and block) I/O is blocking, that is an application has to wait on a `read()` or `write()` system call. With asynchronous I/O (AIO), it is possible for an application to issue multiple I/O requests to multiple devices and get notified when the operation completes via a callback mechanism [43]. There is a Posix specification for Unix asynchronous I/O and it's available on most large scale Unix implementations, some Linux vendors have included it in their offerings and it should become a part of the official kernel releases soon.

1.3.5 Raw I/O

Raw I/O is a mechanism that Unix implements to allow applications to perform their own I/O scheduling. Unix provides a generic I/O scheduler which may not be suitable for certain workloads. I/O intensive applications, Database management systems for instance, benefit a lot from doing their own scheduling on device accesses.

1.3.6 Vector I/O

Some Unix systems implement a variant of the `read()` and `write()` system calls usually referred to as *scatter-gather IO* or *vector IO* [44]. The corresponding system calls, `readv()` and `writev()`, are very suitable for transferring data from contiguous disk blocks to non-contiguous memory buffers. Assuming that modern file systems make a lot of effort to keep a file's data within contiguous blocks, the use of vector IO avoids to issue multiple system calls and multiple IO requests.

1.3.7 Channel bonding

Channel bonding consists into aggregating multiple network interfaces at the link layer and giving to the networking protocol stacks the illusion of a single interface [40]. This technique in usually implemented in Unix servers and networking hardware like Ethernet switches and IP routers. The same concept is called *Etherchannel* by Cisco and *Trunking* by Sun.

1.4 Conclusion

From our short survey of research work related to high-performance scientific processing, we believe that the tendencies are converging toward commodity operating systems like Linux. Legacy Unix semantics that might impede high performance IO and processing are being replaced by novel techniques. This work is by no means an attempt to create a new operating system or a clustering environment, but a study of how good a commodity system running a commodity operating system performs.

Computer systems and operating systems research have focused on various hardware and software sub-systems. This kind of research is inherently complex and very technical. For the modern systems administrator and researcher it may be hard to sort out all these technologies, for this matter the following chapter is an attempt to dissect a computer system's architecture to shed light on the fundamental sub-systems and to present a clear view of the hardware architecture, the operating system and the networking facilities without too much intricacy.

Chapter 2

Architectural Overview

"A system is anything with no system to it,
hardware is anything that clutters when you drop it,
and software is anything for which there is a logical explanation why it's not working"
— Johannes Leckebusch

The objective of this work is to evaluate the performance of a single computing node in a computer cluster, this implies the performance of both the operating system and the underlying hardware. This chapter starts by presenting the modern computer's architecture and the state of the art of processing, storage and communication hardware. This overview will, hopefully, give a rough idea about the commodity system's hardware performance.

2.1 The low-end computer system

In this chapter we aim to give an overview of a *commodity computer* architecture emphasizing on the performance issues. We mean by "commodity computer", as opposed to *custom computers*, a general-purpose computer system that is built conforming to current industry standards like desktop computers (Personal Computers and workstations). We need such an introduction to understand and explain some future technological choices.

2.1.1 The CPU

Over the last years, the CPU industry knew important advances in terms of integration and clock rate mainly due to the advances in the semi-conductor technology and micro-architecture. Two clans appeared in the early 80's with different tendencies: the *Complex Instruction Set Computer* (CISC) approach that uses a large set of large building-

block instructions, and the *Reduced Instruction Set Computer* (RISC) approach which uses a small but fast set of instructions. Table 2.1 compares the CISC and RISC tendencies.

Feature	**CISC processors**	**RISC processors**
Cache splitting	Uses a single cache for data and instructions	Uses separate caches for data and instructions
Instruction set size	Large (>=120)	Small
Register set	Small (~20)	Large (up to 128)
Cache size	Relatively small (some hundreds of KB)	Large (some MB)
Microprogramming	Uses micro-code	Everything is hardwired, no microprogramming

Table 2.1: CISC and RISC approaches compared.

The *base cycle* of a processor is the interval between two clock ticks. An instruction is executed in one or more cycles depending on its complexity. The clock rate is an important factor affecting performance but it is not the unique one. To speed up the execution, many techniques are deployed by the designers and the manufacturers [1] like: pipelining, out-of-order execution, branch prediction, high clock frequency, software optimizations and cache splitting. Refer to Appendix A for further details about these techniques.

2.1.2 The memory

The memory is organized in a hierarchy as shown in Figure 2.1. A memory at a given level of the hierarchy is characterized by an access time, a size, an access bandwidth and a transfer unit size. The fastest memory, which is the closest to the CPU, is the cache and it is used to temporarily hold data and/or instructions [1].

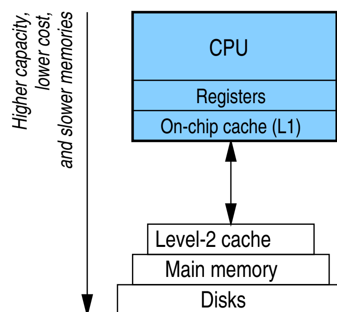


Figure 2.1: The memory hierarchy.

Cache systems are very fast memory chips. They may be implemented as a level 1 (L1) on-chip cache, or as a separate component called a level 2 (L2) cache. As the cache is the innermost memory level, its access time is very low but it has a restricted size because of its high cost. Only very-often-accessed data is stored in the cache. The most influencing parameter in cache design is the *hit-ratio* (as opposed to the *miss-ratio*) which is the average ratio of the queries for which data is available (cache hits) to the total number of cache queries.

The cache design is suggested by the assumption that a program will behave in such a way that the cache will satisfy most of its references. Cache-misses will cost a penalty in terms of access time making it even larger than a direct, non-cached access to the main memory. But, the overall performance is better with a cache than without [1].

Memory management is a very complex task, implementing all its functions in software will make operating systems larger and worse, will slow down the executions. Most of modern computers integrate a hardware support for memory management into special ships called *Memory Management Unit* (MMU).

2.1.3 The input/output

The input and output devices are controlled by special controllers. Such devices include serial communication ports, disks, network interfaces, etc. A controller is a board with a specialized processor, much like the CPU itself which can be viewed as a global manager of the other processors. An IO processor is useful for off loading the CPU from tedious low-level operations when asking for services from the device. Some systems provide a pre-installed set of firmware routines to handle hardware IO operations. IO operations will be further discussed in next sections. The most obvious cases for IO systems are the disk storage and networking.

Networking devices are found on the IO bus, the most important factors in networking performance are: network *bandwidth* and network *latency*. Bandwidth and latency are closely tied to the networking protocol used for communication, the software implementation of the device drivers, the communication software and the networking technology. Networking issues will be discussed further in a next chapter.

2.1.4 The address spaces

The CPU has to address memory locations and IO ports, for that reason there are two distinct address spaces. The CPU can access both address spaces whereas device controllers can only access their IO space, they cannot access memory address space without going through the CPU.

There are typically three different address spaces: physical (or real) memory address space, bus IO address space and bus memory address space, see Figure 2.2. Phys-

ical memory address space is what the CPU sees when it feeds addresses on the memory bus. Bus IO address space is used to access devices' internal registers that are mapped to IO addresses. The bus memory address space contains the memory used on-board by the devices, the CPU addresses this memory to access internal buffers on devices [3].

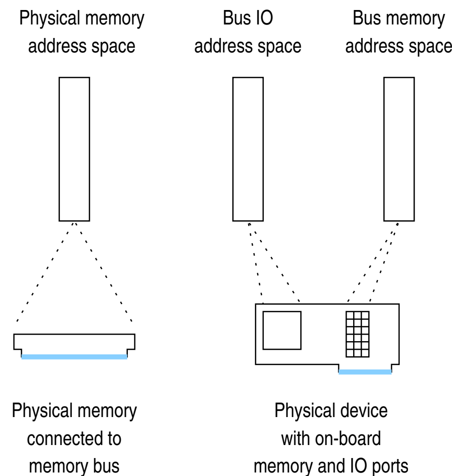


Figure 2.2: Illustration of address spaces.

The CPU uses different instructions to access the memory address space and the IO address space. Indeed, the CPU controls the devices by writing to and reading from their internal registers. The controlling software knows about the IO address space used by a device, some peripherals have conventional addresses but for others, IO settings must be provided. Some architectures use the same instructions to access both address spaces (bus memory and physical memory).

2.2 Components interconnection

The different components in a computer system are connected together with an internal network: the bus. Besides, they use some mechanisms to communicate and to speed up data transfers between them.

2.2.1 The bus

The bus transfers the data, the control messages and the addressing messages between the different components. The address bus specifies the memory locations where data are written or read, the data bus transfers data and the control bus transfers control messages and signals [3][7]. The access to the bus is granted by a bus arbiter that

resolves access conflicts. The amount of data transferred at each access is the *bus width*.

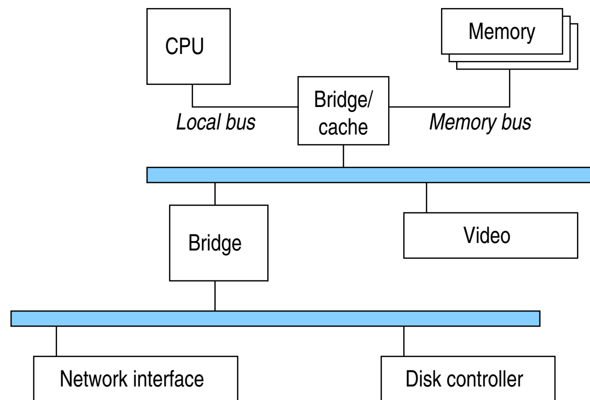


Figure 2.3: Bus hierarchy.

As the main memory is a somewhat particular component and its access time has dramatic effects on the whole system performance, it is accessed through a special bus: the *system bus*. The system bus connects the memory, the CPU and the bridges to other buses. When the CPU wants to send data to a device, say the network interface, it issues a request to acquire the bus then waits till it is acknowledged. Once the bus is granted to the CPU, it starts sending its data to the particular device port.

As there are different bus technologies, computer designers use special chips, called *bridges*, to convert signals from one bus type to another. Bridges are also used to connect buses of the same type. The buses are connected into a bus hierarchy (see Figure 2.3) and access to the main memory depends upon the "position" of the device in the bus hierarchy: devices that are some bridges away from the main memory will spend more time to read/write data than those closer to the memory bus. That's why devices requiring high-speed data transfers, like high-resolution graphics adapters and high-speed network interfaces, are connected to the bus at the first level in the hierarchy. Whereas, slower devices like disk controllers and slow serial interfaces are connected to the local bus through one or more bridges.

A very critical factor in the bus performance is the bus latency which is the amount of time between the issuing of the bus acquisition request by a device and the start of the effective data transfer. Bus latency depends mainly on arbitration protocols used by the bus controller. Though a bus may have a low latency and a high throughput, some devices are unable to use it at full speed. Devices that are able to send chunks of data continuously are said to be *back-to-back* capable, these devices are able to keep the bus operating at its highest performance while they transfer data.

2.2.2 Programmed IO

The simplest way of transferring data between the device and the memory is to let the CPU read one byte from the device into a register, writes it to the memory, checks the device's status register and repeat this process as many times as needed. This mechanism, called *Programmed IO* (PIO) has the benefit of requiring no special hardware and, therefore, no special programming. The major drawback is that the CPU would be spending too much time on trivial IO operations. PIO is used only for low-speed devices like modems and slow hard disks or CD ROM drives.

2.2.3 Interrupts

When the CPU instructs a device controller to perform some particular job, say sending a large data block over the network, the device starts to work while the CPU waits for it. This situation causes the loss of valuable time, as the CPU will be "busy doing nothing". Two mechanisms are implemented to avoid this situation: *polling* and *interrupts*. Polling consists into periodically probing the device to know its status and to find out whether it is ready to receive data or whether it has finished some work. Polling is used on some printers connected with a parallel interface.

Interrupts are hardware signals sent by the devices to the CPU to claim its attention. Once the CPU has submitted the job to the device, it returns to execute something else. When the device is done, it notifies the CPU by sending an interrupt. Some interrupt lines could be *shared* among many devices. That is multiple devices can use the same interrupt to notify the CPU. Of course this sharing causes degradations in the performance of the device and, in practice, it is only used with relatively slow devices. Network interface cards are a good example of devices using interrupts.

2.2.4 Memory-mapped IO

Some systems map a part of the main memory address space to IO ports. A store to address X becomes an IO operation, and generally IO access is performed using the usual FETCH and STORE instructions. This is an elegant way of communicating with devices since the operating systems would use the same primitives as for memory IO, on the other hand it's sometimes complex to operate by the device driver because main memory management may be very different from an IO memory. Memory-mapped IO is used mainly on graphics cards and some network interfaces.

2.2.5 Direct Memory Access

When a device requests an interrupt it is generally to transfer data. The CPU transfers data from the memory to the device in small amounts (1 to 4 bytes in most systems) which ties the CPU for some time. A mechanism to overcome this loss of time is the

Direct Memory Access (DMA) which lets devices transfer data to/from the memory without involving the CPU. DMA benefits are noticeable with large transfers, since without it the CPU becomes too busy to sustain a high rate while with DMA on, higher transfer rates are possible. DMA is used on fast hard disks and CD ROMs and many network interface cards.

2.3 Today's technologies

A typical desktop computer system that is shipped at the time of the writing (late 2002) is comprised of the components outlined in Figure 2.4.

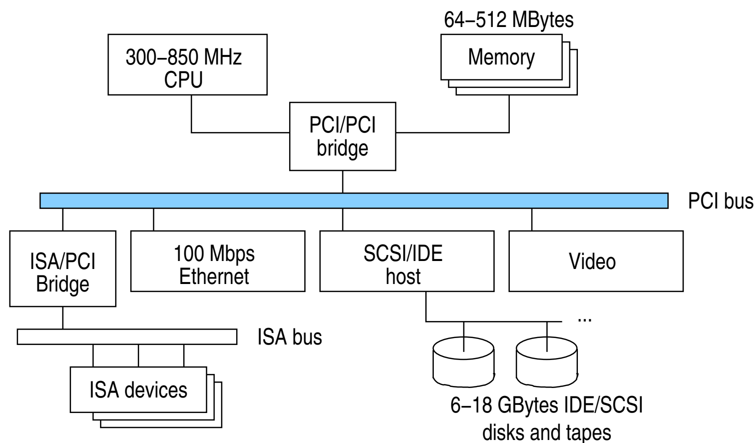


Figure 2.4: Composition of a typical PC.

2.3.1 The CPU

The Intel Pentium and compatible processors are the most used chips in today's computers. The Digital Alpha, the SGI/MIPS MIPS, the HP PA-RISC and the Sun SPARC processors are also widely used among workstations. It is worth noting that all these processors, except the Intel, are RISC processors, but the distinction between RISC and CISC implementations is quite vague and no vendor sticks to strict guidelines. CPU designers seem to implement whatever concept that makes the chip run faster.

- Digital Alpha
  Alpha is a RISC processor designed by Digital in 1992 to support future enhancements and evolutions. A major feature of the Alpha CPU is the *Privileged Architecture Library* (PAL) calls which is a set of programmable macros written in the Alpha instruction set. Alpha has a large instruction set and a small (8Kb)

on-chip cache and, compared to other RISC processors, it supports high clock rates.

- SGI/MIPS MIPS
  MIPS started as a project at Stanford University. The original aim was to build a Microprocessor without Interlocked Pipeline Stages. Silicon Graphics Inc. introduced the MIPS R2000 in 1986 as the first commercial RISC processor. The CPU includes a Memory Management Unit supporting a cache. The latest processor in the MIPS family is the 64-bit R12000 with multiple Floating Point Units and split caches.

- HP PA-RISC
  The *Precision Architecture* was designed by Hewlett-Packard in 1986 as a replacement of the Motorola 680x0 processors used in their workstations and servers. PA-RISC has 32 32-bit registers and 32 64-bit registers that can be seen as 32 32-bit and 16 128-bit registers, it has a particularly large instructions set for a RISC CPU. The PA-RISC 8500 model included a MMU and a Level 1 cache of 1.5 Mb and has outstanding integer and floating point performance.

- Sun (Ultra)SPARC
  The *Scalable Processor ARChitecture* was introduced by Sun Microsystems in 1987. Sun is a manufacturer of workstations, they used to employ the Motorola 680x0 processors but the RISC architecture was promising and they wanted to build their own processor. SPARC is not a chip, it is an open specification that Sun licensed to many manufacturers, hence there are many designs. SPARC uses pipelines and contains about 128 or 144 registers, only 32 are available for a given function: 8 are global and the remaining 24 are allocated in a *window* from a stack of registers. At each function call the window is moved 16 registers down so that the called function has 8 common registers with the previous window (corresponding to the calling function), 8 local registers and 8 common with the next function to be called. The window technique allows for 1-cycle function calls. The Ultra SPARC introduced many modern features and become a fully 64-bit CPU.

- Intel x86
  The 8086 was introduced in 1979 and was based on an awkward design which caused problems with source compatibility in modern Intel processors. The last Intel model is the Pentium Pro which features integrated FPU, MMU, separate Level 1 data and code caches and a small set of general purpose registers. It has up to 512Kb on-chip Level 2 cache and a 14-stage pipeline, it makes extensive use of branch prediction. The old Intel architecture was a barrier to performance enhancements, that's why Pentium clones designed by AMD/Nexgen and

Cyrix/National Semiconductors broke with the old design and used a RISC core with on-the-flight translation of the x86 code.

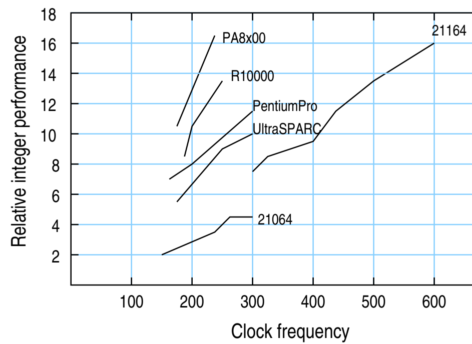


Figure 2.5: Integer performance of various processors compared [6].

The most commonly used processors are the Intel Pentium family, Pentium processors seem to display average performance at similar clock rates but with all the market investments, their design is improving a lot faster than RISC-class competitors. The second most popular processors are UltraSparc and to a lesser extent the PowerPC and the Alpha chips [17].

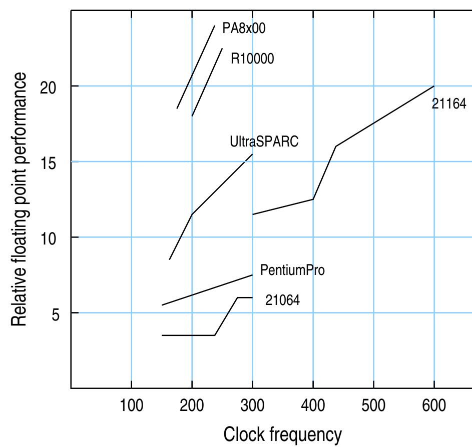


Figure 2.6: Floating point performance of various processors compared [6].

Figures 2.5 and 2.6 show average performance for the Pentium-class processors either for floating point or integer performance. This is not necessarily a determinant factor in the choice of a platform since modern applications perform a lot more IO operations than pure calculations. Besides, in a clustered environment the bottleneck is shifted from the CPU to the IO subsystems.

2.3.2 The memory

Main memory boards are built using Dynamic Memory (DRAM) chips and caches are built using Static Memory (SRAM) chips. Static Memory chips are faster, more expensive, and less dense than Dynamic ones. Today's computers use static memory to build main memory boards. A typical configuration would use between 64 and 512 MB of RAM and the average access time to the main memory is about a few tens of nano seconds on a modern memory chip. It's an interesting fact here that memory chip speed (access time) didn't improve as much as CPU integration or disk speed and capacity over the last 10 years.

2.3.3 The bus

Workstations manufacturers used proprietary bus technologies for a long time. Though the world of Intel-based PCs knew a quite different situation (too many bus technologies), the results were very annoying too. Indeed, many bus technologies were deployed in the PC systems ranging from slow, 16-bit ISA to fast, 64-bit PCI. The most popular bus technologies are summarized in Table 2.2. It's noteworthy that the actual bus throughput varies between half and two thirds of the the theoretical peak bandwidth, and that some motherboards may be overclocked. For example some users reported success with an ISA bus at 26 MHz.

IO Bus	Width	Bus speed	Theoretical Peak Throughput	Found in..
ISA	8/16 bits	5.33/8.33 MHz	5.33/8.33 MBps	PC
EISA	32 bits	8.3 MHz	33 MBps	PC servers
SBus	32/64 bits	16/25 MHz	100/200 MBps	Sun Sparc
PCI 1.0	32 bits	33 MHz	132 MBps	PC, Sun Sparc, Alpha, PowerPC, ..
PCI 2.1	64 bits	66 MHz	528 MBps	PC, Sun Sparc, Alpha, PowerPC, ..

Table 2.2: Popular bus technologies compared.

- ISA: the *Industry Standard Architecture* is a very popular 8- or 16 bit-wide bus that is, for compatibility reasons, still available on many newly designed PCs

despite its low performance, ISA devices can be either 8 or 16 bit. ISA's typical operation speed is 16 MBps at 8.33MHz, but motherboards may ship with variable clock speeds, ranging between some 6 MHz up to 27 MHz.

- EISA: the *Extended Industry Standard Architecture* was pioneered by Compaq Corporation and was meant to be a 32-bit wide replacement that overcomes the weaknesses of ISA. EISA wasn't widely used and was restricted to the high-end server market mainly because of its high cost. It supports up to 33 MBps.

- MCA: the *Micro-Channel Architecture* was introduced by IBM with their PS/2 systems. MCA wasn't widely used because of the strategy of IBM that kept it secret and didn't open the specifications for other manufacturers. MCA's theoretical maximum throughput is 32 MBps.

- VLB: *VESA Local Bus* was a designed to work on i486 systems. It knew a good success but vanished soon. Is supports up to 30 MBps.

- PCI: the *Peripheral Components Interconnect* is the most deployed bus technology among today's computers. It supports 32 and 64 bit bus widths and up to 132 MBps bus throughput (for 32 bit PCI) at up to 33 MHz bus speed. PCI 2.1 allows 64-bit width at up to 66 MHz bus speed (512 MB/s).

- SBus: was developed by Sun and first introduced in the SPARCStation 1 in 1989. SBus is defined by IEEE standard 1496, it runs at 16 or 25 MHz and theoretically achieves up to 200 MBps.

PCI has the merits of being open, upgradeable and processor-independent. Since the *PCI Special Interest Group*[1], an independent group, is responsible for writing specifications for PCI, the technology doesn't depend on a particular brand. Devices connected with a PCI bus can identify themselves to the bus controller and declare the resources they use such as interrupt and DMA channels, this auto-configuration feature contributed greatly to the success of PCI. By opposition to VLB, PCI is independent from the CPU and the host architecture it is connected to, which makes it possible to deploy PCI buses in multiple architectures regardless the processor they use. Indeed, Digital use PCI in their Alpha systems and Sun Microsystems are shipping UltraSPARC workstations with their proprietary SBus bus and with the PCI bus. Apart from the high bandwidth transfer capability, PCI supports low bus latency. A device connected to a PCI bus can explicitly specify the maximum bus latency it can work with. Like in most communication systems, PCI data transfers incur a fixed overhead that's not proportional to the amount of data being transferred.

[1] http://www.pcisig.com

2.3.4 The storage devices

Most of the modern operating systems use a swap space as an extension to the main memory. Swap space could be a file in the file system or a raw disk. Besides, scientific applications make intensive use of the storage devices since they process very large amounts of data. For these reasons the performance of the storage devices has a direct impact on the overall performance of a computer system.

Integrated Disk Electronics (IDE) is the most used storage technology among desktop PCs. The *Small Computer System Interface* (SCSI) technology is well known for workstations and high-end PCs. Both knew many enhancements over the last decade.

The standard IDE supports up to 2 hard disks of no more than 528 Mb, one device has to be explicitly (by setting jumpers) declared as master and the other as slave. The IDE bus just duplicates the ISA bus signals. When a device is being accessed the other is blocked until the operation finishes. The *Enhanced IDE* (EIDE) introduced the support for mixed devices on the same interface (hard disks, CD-ROMs, tapes, etc.), it behaves like two independent IDE interfaces, that is it supports up to four devices. EIDE inherited the interlocking restriction between devices on the same interface, but devices on different interfaces may work in parallel. The Ultra ATA is the latest member of the ATA family, it is capable of transferring 33 MBps. Ultra ATA becomes available on any newly shipped PC as a replacement of the Fast ATA with only 16 MBps.

The standard SCSI connects up to 8 devices on a single interface and supports concurrent operations on different devices, the specifications limit throughput to 5 MBps, see Figure 2.3. SCSI devices work in a peer-to-peer mode. The two major differences directly affecting performance are: bus width and bus rate. Bus width is either narrow (8 bits) or wide (16 bits). The "normal" (or narrow) SCSI operates at 5 MHz and with 8-bit width which allows for a 5 MBps peak throughput for data transfers. SCSI has the benefit of supporting multiple devices operating at a high throughput (up to 160 MBps), the ease of configuration and the support for dynamic addition and removal of disks.

Fibre Channel is making steady progress in large installations, it's a switched network running over fibre optical links and operating at speeds up to 2 GBps. It's largely used in Storage Area Networks (SAN) and by shared-disk file systems.

High-end servers which require high availability and low response times use some storage technologies that greatly improve their performance. The most common technology is RAID which stands for *Redundant Arrays of Inexpensive Disks*. RAID consists into managing a set of disks with some software or firmware algorithms and giving the operating system the illusion that it is dealing with only one disk. There are five basic RAID levels and four hybrid levels, the most interesting basic levels are:

- RAID-0

Type	**Number devices per channel**	**Transfer rate**
SCSI-1	8	5 MBps
Fast SCSI	8	10 MBps
Fast Wide SCSI	16	20 MBps
Ultra SCSI	4 or 8	20 MBps
Wide Ultra SCSI	4, 8 or 16	40 MBps
Ultra2 SCSI	2 or 8	40 MBps
Wide Ultra2 SCSI	2 or 16	80 MBps
Ultra3 SCSI	2 or 16	160 MBps

Table 2.3: SCSI throughput.

This level is a bit special as it doesn't provide any redundancy and is referred to as *striping*. Indeed data is split onto drives into stripes, this helps in lessening concurrent access to a single drive and provides higher throughput to concurrent applications requesting data simultaneously. RAID-0 is widely used among systems processing large data sets.

- RAID-1
  This level is commonly referred to as *mirroring*. On writes, data is written to multiple disks so that the failure of a disk doesn't cause data loss. RAID-1 is widely used on data servers in enterprise networks.

2.3.5 Networking

Communications performance is a key factor when building distributed systems or Networks of Workstations. Many networking technologies were deployed in local area networks in the last two decades, the most popular technology is *Ethernet*. Ethernet was introduced by Digital, Intel and Xerox in the early 80's. It has operated well in LANs with moderated loads, besides it has the advantages of having very good implementations (hardware and drivers have improved over time), a low cost of ownership and simple installation and maintenance procedures. Many theoretical studies showed that Ethernet was not suitable for loads higher than 37%, but Ethernet performs much better in real-world applications. Indeed, though most of theoretical models used too many simplifications they were still too complex for the average reader [10] which led to misunderstanding. The standard Ethernet specified a 10 Mbps capacity, but many extensions appeared like the *Fast Ethernet* (100 Mbps) and the *Gigabit Ethernet* (1000 Mbps), 10 Gbps is under specification. Besides, the *Switched Ethernet* (at all speeds) introduced many performance enhancements to the standard shared Ethernet. Ethernet networks scale quite well up to a few dozen hosts with a latency of about a hundred micro-seconds.

Asynchronous Transfer Mode (ATM) is a very large set of networking protocols

that spans many fields. Is was designed to satisfy all the needs for networking: high-speed backbones, local loops, local area networks, mobile systems, etc. One of the key features of ATM is its ability to provide a deterministic quality of service, since it is connection-oriented it is less vulnerable to the contention problems faced by shared medium networks like Ethernet. The major problem with ATM is its still high cost of ownership and complexity of installation which places it far from a "commodity network".

This has been a very brief glimpse on networking technologies, a whole chapter (chapter 4) is dedicated to discuss network performance issues.

2.4 Conclusions

This chapter, and actually the whole work is far from comparing computer clusters to supercomputers (see Appendix B), still we can make some comments based upon the above study: today's personal computers are still not as powerful as custom supercomputer nodes when it comes to floating point performance, but PCs and workstations have some advantages that make them very "appealing" for applications like scientific computing on local area networks:

1. low cost: investments in PCs and workstations are amortized over a large sales volume,
2. availability in the production environments: a PC or a workstation is a handy hardware in almost any production environment,
3. wide software platform: the software base and the array of operating systems developed for PCs/workstations is quite wide,
4. very low maintenance cost: both for the software and for the hardware.

Because of the wide spread of the PC as an instrument for everyday's work, the shipment volumes are very high. This situation helped in dropping PC prices significantly. Since PCs are using almost the same technologies as workstations (bus, memory chips, network interfaces, etc.), they are perfectly adequate for most jobs when provided with a good operating system [16]. It's hard to argue that the acquisition cost of a reasonably sized cluster is still way below the price of a parallel computer.

This first chapter has shown that the Intel processors aren't as fast as some RISC processors in terms of floating point and integer performance. But the advances in networking technologies have shifted the bottleneck from the network to the host's IO subsystem, that is, it doesn't help a lot to have a fast CPU since network, disks and memory IO will delay the overall operation anyway. Another critical factor in favor of computer clusters is the *scalability*: it's a lot easier to scale a network of computers and

their storage and processing power than to scale a supercomputer which ranges from very hard and very expensive to impossible.

The hardware components used in computer clusters are not incomparable to supercomputers. Actually, many components used in supercomputers are found in workstations. The building blocks are getting normalised and the hardware offerings from different vendors are getting more and more similar: the PCI and the SCSI buses are being used ubiquitously, many hardware designers are planning to use the new Intel Itanium processor in their future offerings.

What's left is the second major component of a computer cluster node: the *Operating System*. The question is: how much overhead would this component introduce? The next chapters will explore the performance issues of the operating system running a node in a PC cluster and its various sub-systems.

Chapter 3

The Operating System

A computer system runs a set of processes. These processes use concurrently the same resources: the CPU, the memory and the IO (network access, disks, video display, etc.). The operating system could be seen as a resource manager, it provides the applications with an interface to request services from the underlying hardware, and it hides the hardware intricacies and specific designs from applications. An operating system provides applications with a set of tools for cooperation, synchronization, etc. and it presents a consistent and uniform view of the computer system. Being the intermediate between the applications and the hardware resources, the operating system has a paramount impact on the overall performance.

In the following we'll discuss some operating systems details assuming the UNIX model and, when appropriate, giving examples from the Linux and the Solaris kernels. We'll try to evaluate the costs introduced by the operating system as the mediator of operations between user applications and the hardware.

3.1 Architecture

The UNIX kernel has a monolithic structure but can be viewed to be composed of major modules as shown in Figure 3.1. The monolithic structure virtually lets any code within the kernel reach any other code using trivial function calls or jumps within a single address space in low-level terms.

Modern applications become increasingly dependent on multimedia, graphics, and data movement, they are spending an increasing fraction of their execution time in the operating system kernel. Most of the engineering and scientific applications follow this pattern, even applications handling small amounts of data and running CPU-intensive calculations tend to perform heavy IO operations when they run on a cluster of computers. This behavior makes the operating system a key element to the performance of distributed parallel applications.

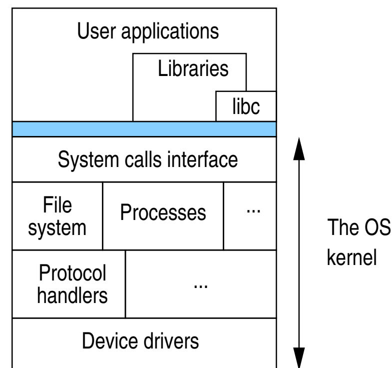


Figure 3.1: The UNIX kernel architecture

3.2 Virtual memory

All memory addresses manipulated by processes belong to a virtual address space: Virtual Memory. Virtual memory, as the name implies, is a virtual linear array of addresses organized in small blocks (pages) that has no direct mapping in hardware. The way this memory is managed is quite complex, Figure 3.2 shows a very sketchy architecture of the virtual memory in the Linux operating system.

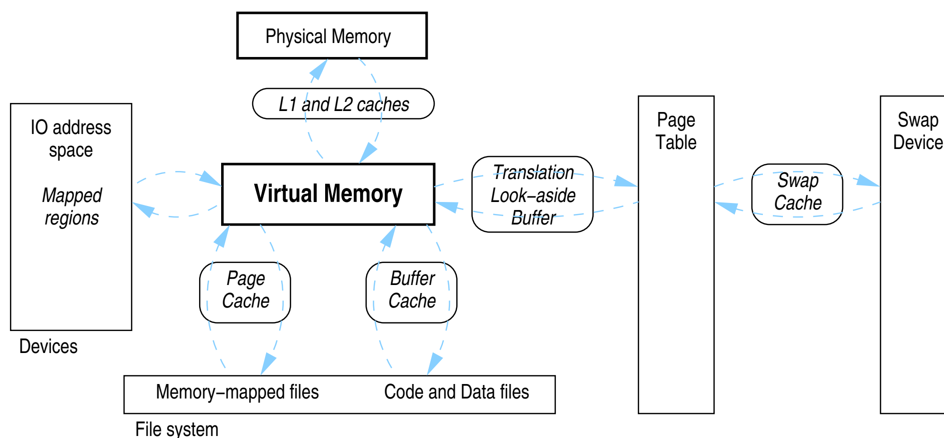


Figure 3.2: Virtual memory management

Caching is definitely an efficient way to gain performance in IO operations, that's why Figure 3.2 shows so many buffering and caching between entities. Some operating systems, like Linux, use dynamic buffer sizes so that to use as much memory for buffering as possible and release it when it's requested by processes. Since almost all CPU instructions involve memory access, it's crucial to optimize the management of this resource.

3.3 Kernel space and user space

Since the kernel code happens to directly handle hardware addresses when processing requests from devices and since it's the kernel code that's running virtual memory, its address space is directly and statically mapped to physical memory. Virtual memory pages hosting kernel data and code are not subject to the normal paging process, they are never discarded from physical memory. For these reasons, the kernel code has to keep a "local" copy of any data it has to process. When a process requests kernel services through a system call no context switching happens, the process just switches execution from user space to kernel space. The kernel code runs with highest privileges, and for that reason very strict boundaries are set to control the access to kernel space. The UNIX kernel also enforces a strict separation between different processes so that the behavior of one doesn't interfere with other processes. This system call scheme provides a flexible and a secure way to run processes but it also implies some overhead that's sometimes considerable. In Linux, for example, system calls are performed as shown below (see Figure 3.3):

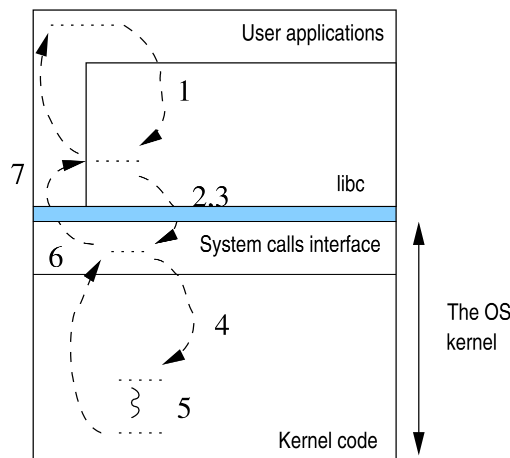


Figure 3.3: A system call path in the Unix kernel.

1. all calls are directed through a stub in the libc library which sets the parameters to be passed to the kernel code,

2. the stub calls an interrupt which is a gate to the kernel, this interrupt is a vector to `_system_call()`, that is invoking this interrupt performs just like a jump to `_system_call()`,

3. the call transfers to the `_system_call()` function which saves registers and makes sure a valid system call is being invoked,

4. the `_system_call()` function fetches the offset of the invoked system call in a table and branches to the corresponding address,

5. the kernel code for the system call executes,

6. back in `_system_call()`, it executes `_ret_from_sys_call()` which might invoke the scheduler to run another process,

7. back in the libc stub, it checks whether a negative value was returned and sets the `_errno` system variable if so.

The kernel space vs. user space duality yields to another overhead in kernel services involving user data movement. User data has to be copied from the process's address space to the kernel space before getting processed by kernel code and then copied again to an IO device memory. This double copy creates a heavy traffic on the memory bus.

3.4 Kernel performance

We have ran some benchmarks to get some insight into the way the operating system copes with the hardware. The overhead introduced by the kernel mechanisms is a particularly important factor. The overall performance can be checkpointed at 5 major stages:

1. application-level and library primitives like `sento()` and `recvfrom()`,

2. high-level operating system primitives like process spawning,

3. low-level operating system primitives like a memory region copy,

4. device driver performance,

5. hardware capabilities like the memory bus speed.

Operating systems developers seem to agree that Linux is one of the fastest UNIX implementations for *Intel Architecture 32* (IA32), commonly referred to as x86 or i386. We have looked into a set of micro benchmarks described in [26] and in [27] against different combinations of hardware and operating systems. The test environment is described in Appendix D.

3.4.1 System calls, file handling and signaling

This is a measure of how fast the system call interface performs, results are shown in Figure 3.4:

Null call — The simplest system call is considered to be `getppid()` which returns the process ID of the parent of the current process, it just reads a value and returns it (null call).

Null IO — The simplest IO operation is assumed to be a read from /dev/zero or a write to /dev/null (null IO). The system call does nothing in the kernel space, it just returns.

File ops. — File handling tests measure the speed of the `stat()` (stat), `open()`, `close()` (open/close) and `select()` system calls, the select test is used to measure how fast the operating system can check and report the status of a set of file handles associated to TCP sockets (TCP select). The open/close test doesn't include the time to read actual inode data from disk since all the file system data must be contained in the buffer cache after the first run.

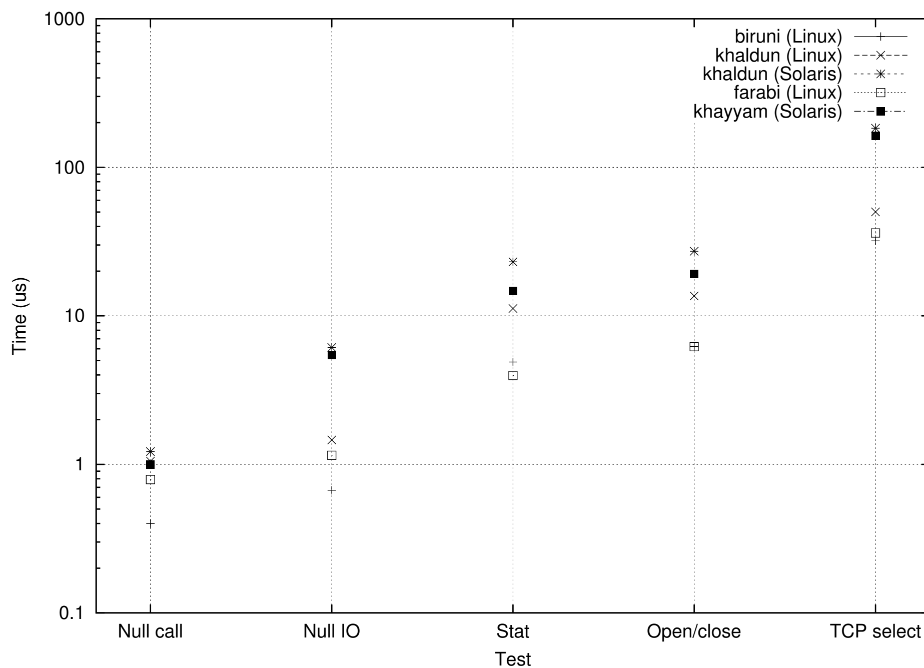


Figure 3.4: System calls and file handling latencies

In this test we notice that on the same architecture Linux performed the null call twice as fast on a 850 MHz CPU than on a 400 MHz one which was the expected result

since this operation uses almost exclusively the CPU. A Pentium III at 850 MHz can execute about 400 Mflops meaning that the time it takes to perform a select() on a set of TCP sockets (~70 μs) is enough to execute 30000 floating point instructions, therefore application slicing has to be done with care and should not go below some granularity.

There is a couple of tests regarding signals, results are shown in Figure 3.5:

Installation Measures how long it takes to install a signal handler with `sigaction()`,

Catch Measures how long it takes the OS to deliver a signal and get the user process to jump to its handler.

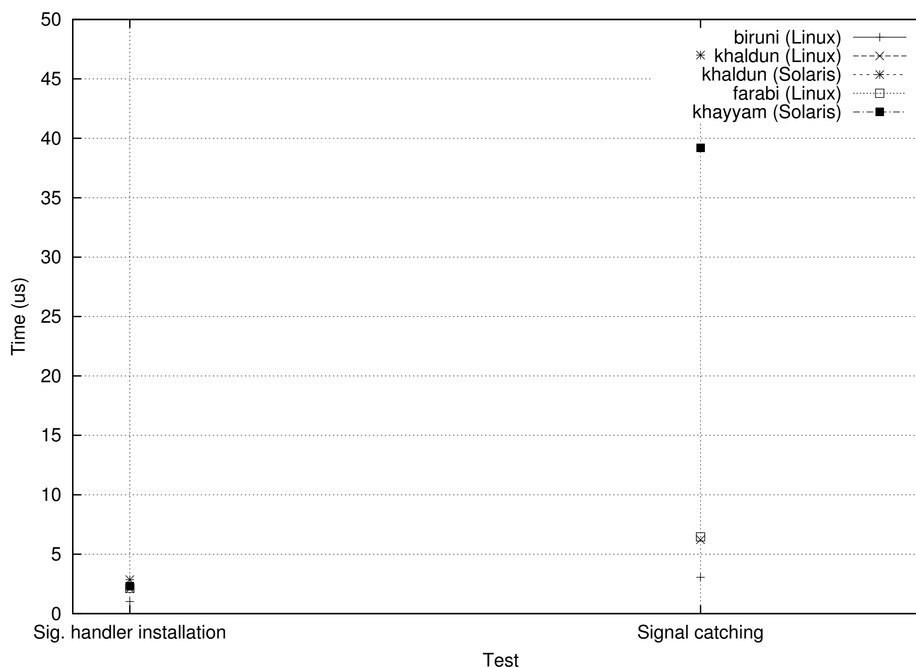


Figure 3.5: Signal handler installation and signal catching latencies

3.4.2 Process spawning

It measures how fast the OS can clone a process with `fork()` and, possibly, run another. Results are shown in Figure 3.6.

Fork The time it takes the OS to make a copy of the currently running process and get one of them to exit,

Fork + execve(prog.) The time it takes the OS to make a copy of the currently running process and overlay the latter with the code of another program and run it,

Fork + execve(sh -c prog.) Same as fork + execve, but get creates a shell which runs a program using the "-c" shell switch.

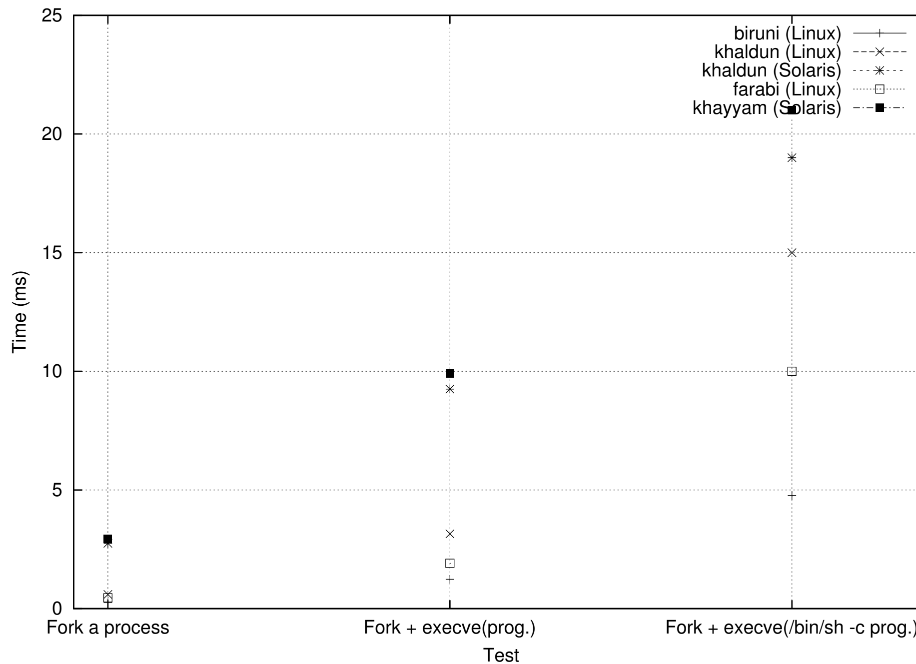


Figure 3.6: Process spawning latencies

Figure 3.6 shows very significant delays for the fork system call. Programmers should pay attention to this matter by avoiding to fork processes on the spot to process requests, it should be possible to run calculations on a dedicated cluster node as a single process. Some Internet servers tend to create a large set of processes initially or to use threads to avoid this effect.

3.4.3 Context switching

This measures context switching time for some reasonable numbers of processes of some reasonable sizes. The processes are connected in a ring of Unix pipes. Each process reads a token from its pipe, possibly does some work, and then writes the token to the next process. Processes may vary in number. Smaller numbers of processes result

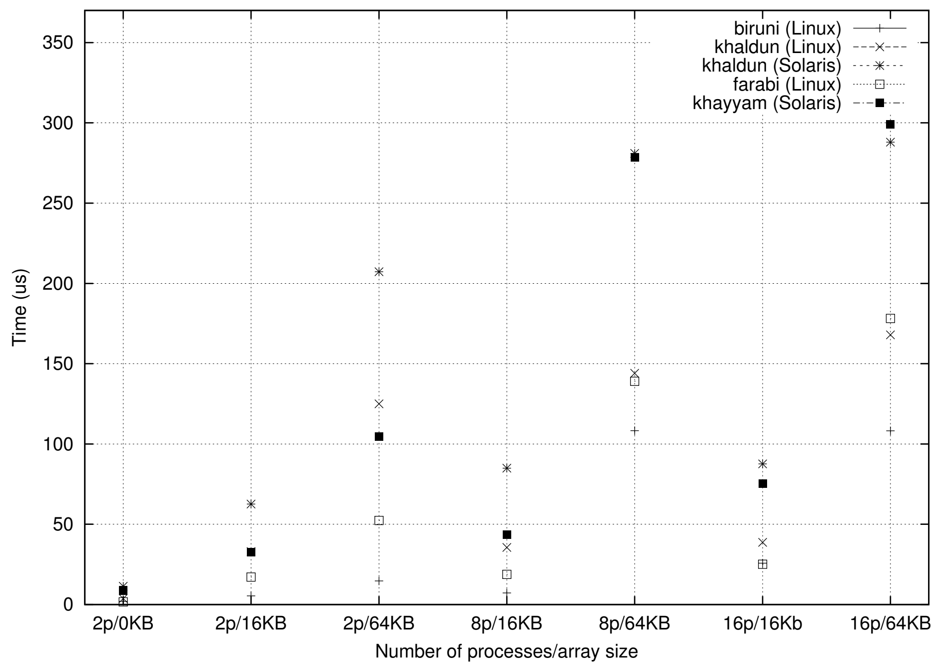


Figure 3.7: Context switching times between communicating processes

in faster context switches. Processes may vary in size, a size of zero is the base-line process that does nothing except pass the token on to the next process. A process size of greater than zero means that the process does some work before passing on the token. The work is simulated as the summing up of an array of the specified size. The summing is an unrolled loop of about a 2700 instructions [26, 27].

The effect is that both the data and the instruction cache get "polluted" by some amount before the token is passed on. The data cache gets polluted by approximately the process "size". The instruction cache gets polluted by a constant amount, approximately 2700 thousand instructions. The pollution of the caches results in larger context switching times for the larger processes. Indeed, larger processes will go through a series of cache misses every time they are woken up, this delay is counted as part of the context switching time. Figure 3.7 shows that large processes (those performing work on larger data sets) have a longer context switching time compared to those processing smaller sets.

The results displayed suggest that the token size is the determinant factor, the number of processes exchanging the same token size doesn't seem to affect performance significantly. Performance seems to drop dramatically when caches are polluted and, therefore, cache misses increase. It would be more efficient to run one single large monolithic process that does a lot of work rather than many small process that compete for the CPU.

3.4.4 Inter-process communications

This test measures various inter-process communication (IPC) facilities, namely:

Pipes — Uses two processes communicating through a Unix pipe to measure inter-process communication latencies. The benchmark passes a token back and forth between the two processes. No other work is done in the processes. The reported time is per round trip and includes the total time, that is, the context switching overhead is included,

Unix sockets — Same as the Unix pipes test but uses Unix domain sockets,

UDP — Same as the Unix pipes test but uses UDP,

RPC/UDP — Same as the Unix pipes test but uses Sun *Remote Procedure Calls* (RPC) over UDP.

TCP — Same as the Unix pipes test but uses TCP,

RPC/TCP — Same as the Unix pipes test but uses Sun RPC over TCP,

TCP connect — Measures the connection latency between two processes using TCP sockets.

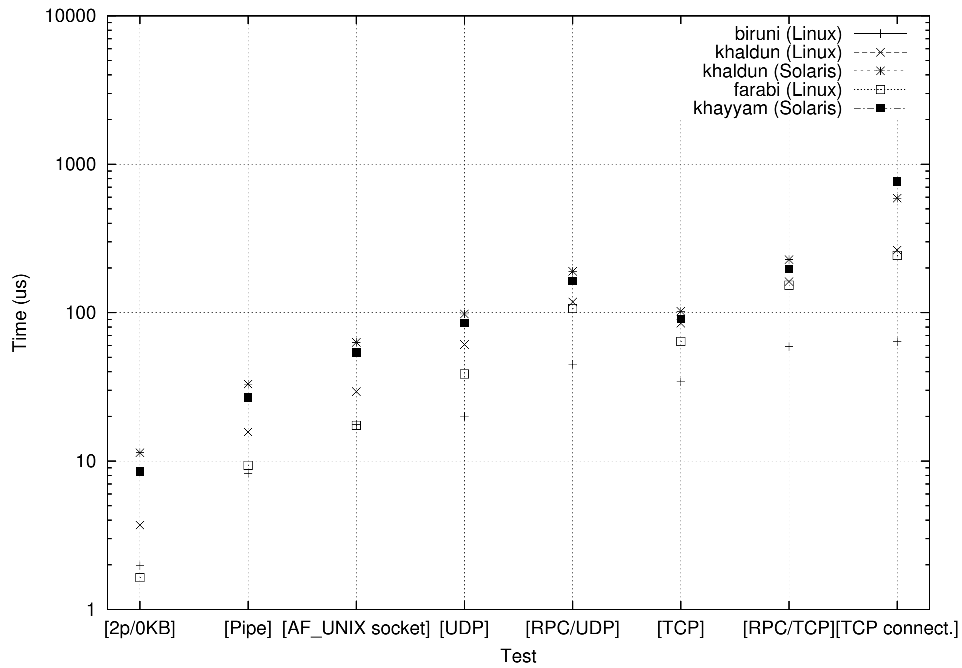


Figure 3.8: Inter-process communication latencies

Figure 3.8 shows, predictably, that the more complex the communication facility, the higher is its latency. The latencies of Unix pipes, Unix sockets, Internet sockets and RPC grow exponentially as the mechanism gets more complicated. TCP connection establishment is clearly a heavy task and, therefore connection establishment has to be avoided. The Unix pipes test is basically the same as the 2p/0K test with two processes exchanging void tokens, but the measured latency of Unix pipes here includes the context switching time.

3.4.5 File IO and virtual memory

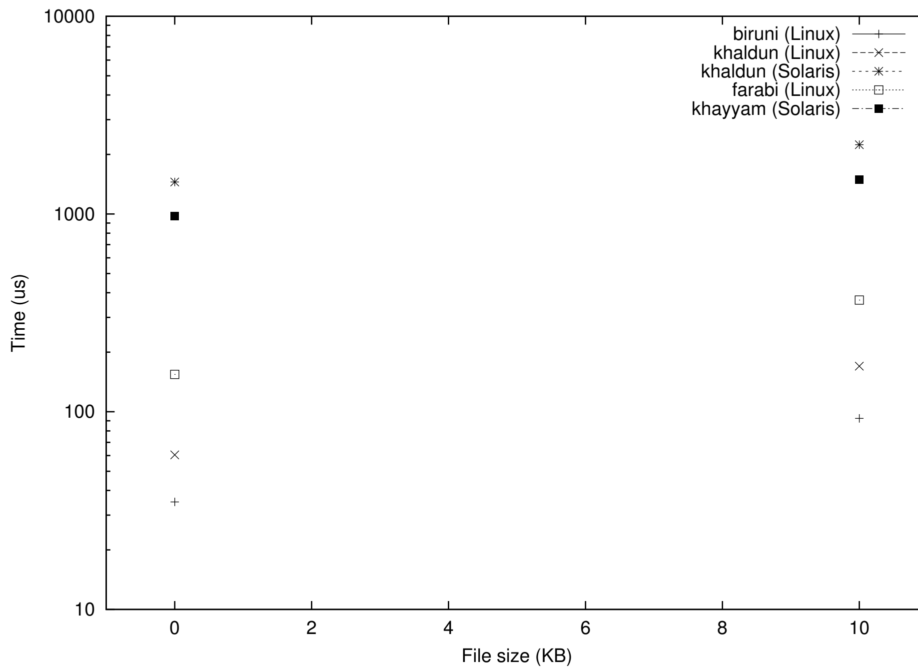


Figure 3.9: File system creation latencies

These tests measure file IO bandwidth. Figure 3.9 shows the file creation latencies for a void file and a 10KB one. Linux operations seem to go about an order of magnitude faster than Solaris on the same hardware, this may be caused by the fact the Linux using ext2fs performs directory operations in cache while Solaris may be doing them synchronously.

3.4.6 libc code

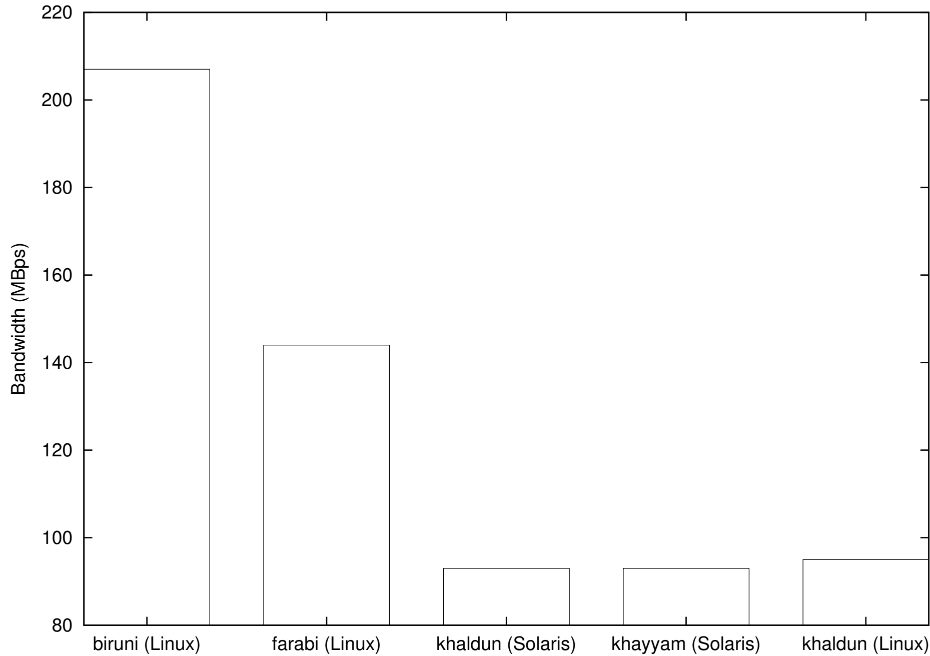


Figure 3.10: Hand-coded bcopy() bandwidth

This test measures the speed of an often-sollicited operation: memory copy. The benchmark compares the speed of the memory copy code as implemented by the libc library to a simple hand-coded memory copy. The performance figures displayed in Figures 3.11 and 3.10 are counter intuitive since the memory copy through libc is faster than the hand-coded implementation. This may be explained by the faster copy code of the glibc implementation of bcopy. Besides, since bcopy isn't a system call (a plain library function), going through glibc doesn't introduce the extra system call overhead. After investigating the code fragments in glibc and in the benchmark, we found out that the glibc code tries to do page aligned copies and attempts to copy virtual memory addresses rather than moving actual data blocks.

3.5 Conclusions

The operating system is a very complex collection of software, this complexity comes from many conflicting requirements like security, performance, portability and sim-

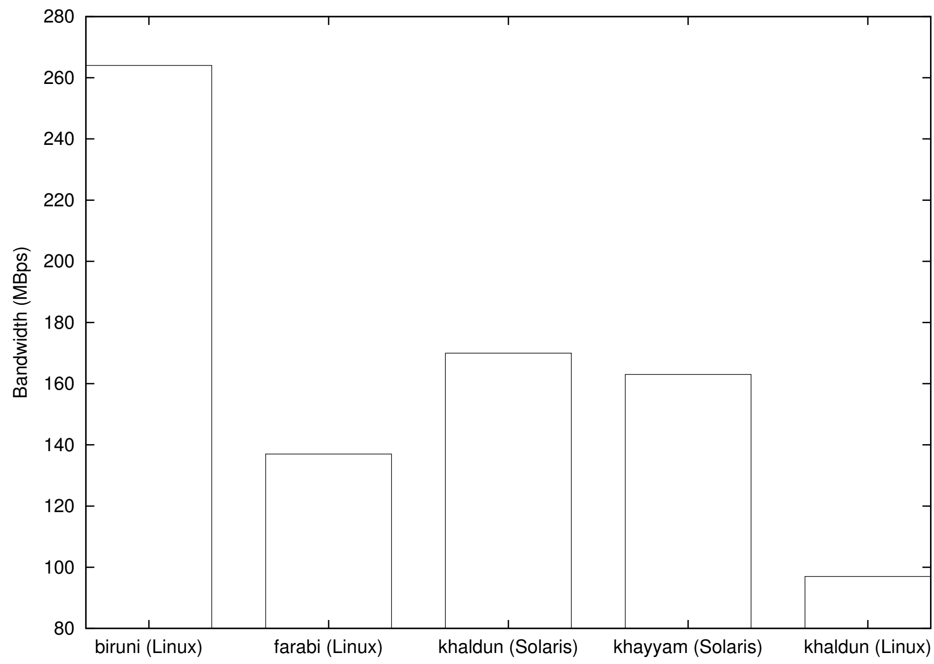


Figure 3.11: bcopy() bandwidth through libc

plicity. The lmbench tests have shown that a commodity and cheap hardware like an Intel desktop running Linux can perform better than a Sun workstation running Solaris, systems running Linux have performed consistently better than Solaris. Contrarily to the common assumptions about RISC processors being faster than the Intel offerings, a low end desktop computer ran faster than a Sun workstation when provided with a good operating system. Besides, Linux does run faster than Solaris on the same hardware.

Operations involving the operating system and, therefore, system calls are relatively expensive when compared to the number of floating point operations that can be performed within the same time frame. This suggests that interactions with the operating systems have to be reduced as much as possible and application decomposition has to remain coarse grained within reasonable limits.

We believe Linux performed better because it's a complete and modern rewrite of Unix, whereas Solaris has inherited some ancient code from AT&T Unix System V that kept dragging it behind and not permitting rewrites or redesigns. The Linux development model benefits from a much wider developer base and, therefore, seems to have a better design, besides Linux development is not restrained by corporate rules or market pressure making rewrites of sub-optimal code possible. Other high level benchmarks have shown that Solaris scales better than Linux on SMP systems with a large number of processors, we did not consider this class of hardware because they cannot be regarded as commodity.

Chapter 4

Interconnecting Nodes

This chapter focuses on networking fundamentals and the performance issues (throughput and latency) in the context of message passing. Since today's most popular networking technology is Ethernet which qualifies as the only commodity network, our measurements and examples are given relative to this technology. We'll start by discussing the network performance issues in general and then the TCP/IP overhead specifically. Finally we'll present the benefits of zero-copy, a popular communication acceleration mechanism.

4.1 Introduction

Computer system's components which are connected with an internal bus are said to be *tightly coupled*. Indeed, the bus is a network that connects computer's components that are physically very close that's why bus designers do not have to bother about error correction, high energy signaling, additional software compatibility layers, etc.

One of the key features of any high-performance custom computer is its interconnection network. Indeed, such computers use common processors, for example the Cray T3D uses 150 MHz Alpha processors that are used in Digital's workstations and Paragon uses 50 MHz Intel i860 that are common in some graphics or network boards. That is, a super computer relies mostly on a high-performance interconnection network to achieve high-performance processing rather than on a super-fast processor.

When connecting remote systems many issues raise, mainly: problems related to limitations on long-distance physical media, unreliable links, incompatibilities due to distinct hardware made by different manufacturers, etc.

4.2 Dissecting network latency

Even a low-end Network Interface Card (NIC) is capable of receiving back-to-back frames, but (for some hardware designs) the CPU might not be able to keep up with the pace. If the CPU is not fast enough to pick up the data from the NIC and free the receive buffers, networking performance could drop dramatically (by a factor of 5 in some cases) [15]. Indeed, if a frame doesn't get moved off the NIC buffer within the appropriate time frame, it will be overwritten or dropped. This frame loss results in the retransmission of the same frame by higher-level connection-oriented protocols. Retransmissions cause very serious performance drops especially when it comes to some higher level protocols like the *Transmission Control Protocol* (TCP). For example, modern implementations (implementing van Jackobson's *Slow Start*) of TCP assume network congestion when they encounter a packet loss and they do dramatically drop their transmission rate.

In a communication network, the most critical performance factors are *latency* and *throughput*. As explained above, latency is the amount of time the network spends since the send request is initiated till the first bit of the message is received on the peer receiving entity on a remote node. Figure 4.1 illustrates the major times that sum up to latency [2].

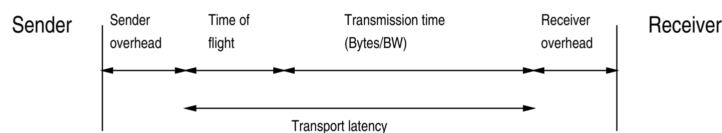


Figure 4.1: Network latency.

The sender/receiver overhead includes high-level protocol processing performed by the CPU and low-level link-layer processing performed by the on-board chips. This overhead strongly depends on the higher level protocols and the way they are implemented in the operating system, for instance Unix operating systems do not permit applications to access hardware directly. The time of flight is the time that the first bit of the message spends to reach the receiver since it leaves the sender NIC. Latency could be simply viewed as follows:

$$Total\ latency = Sender\ overhead + Receiver\ overhead + Transport\ latency$$

Network communications require some processing power from the transmitting and receiving nodes. Networking protocols are very complex pieces of software and the way this software is ran by the operating system greatly affects the overall performance. It is also noticeable that performance might vary depending on the design of the NIC.

For example, the bus technology used to connect the NIC to the CPU and the bus access method will place tight bounds on the transfer speeds. According to Figure 4.1, the time it takes the smallest packet to transmit gives a good approximation of the latency since, in a small network, the time of flight is almost nil[1] and the transmission of a few bytes at a high rate is also almost nil[2]. Using the same assumptions, the throughput can be approximated by measuring the transmission time of very large packets.

As discussed in the first chapter, the models representing the original Ethernet design showed poor results, but today's UTP and the star-shaped cabling topology made it possible to make *Switched Ethernet*, with a dedicated (not shared) medium connecting nodes to an Ethernet switch. Switched Ethernet uses separate channels for sending and receiving and a node can operate in full duplex simply by disabling collision detection. The `ttcp`[3] network benchmarks showed about 12% performance gain using full duplex with FastEthernet [15]. Besides, using switching instead of broadcasts makes the aggregate available bandwidth add up to the number of available switch ports.

4.3 Transport Overhead

Figure 4.2 shows the round-trip times of respective *Protocol Data Units* (PDUs) both over raw Ethernet and TCP/IP. The test equipment is outlined in D.3.

Since Ethernet sends no less than 64 bytes, it was not possible to measure latency by transmitting a single byte. Still, the transmission time of 64 bytes can be considered irrelevant at Gigabit Ethernet speeds. The figure suggests an average host latency for reads and writes of about 47 μs. Figure 4.2 shows, in contrast with common agreement, that TCP is not very bad when it comes to latency, it adds only about 7% overhead. We noticed that there's an agreement between kernel developers that Linux TCP/IP implementation has the lowest latency among all operating systems. It seems that the fixed latencies are much more significant compared to the per-byte latencies, that is the overhead incurred by switching to and back from kernel space is significant compared to the kernel's code processing.

It goes without saying that this is, by no means, a TCP performance measurement but a measure of the latency introduced by the TCP kernel code processing that performs the transmit and the receive operations. In this test, we made all the arrangments to avoid all TCP's complex processing like fragmentation and reassembly, congestion control window effects, delayed acknowledgements, retransmissions, etc.

The difference between the slopes of the two curves suggests that there's a per-byte overhead in TCP processing. The most likely cause is the memory copies to and from user space.

[1] On a fiber (at the speed of light), it takes 0.06 μs to do 20 m.
[2] At Gigabit Ethernet rates, it takes 0.5 μs to transmit 64 bytes.
[3] "Test TCP" is a TCP performance measurement tool.

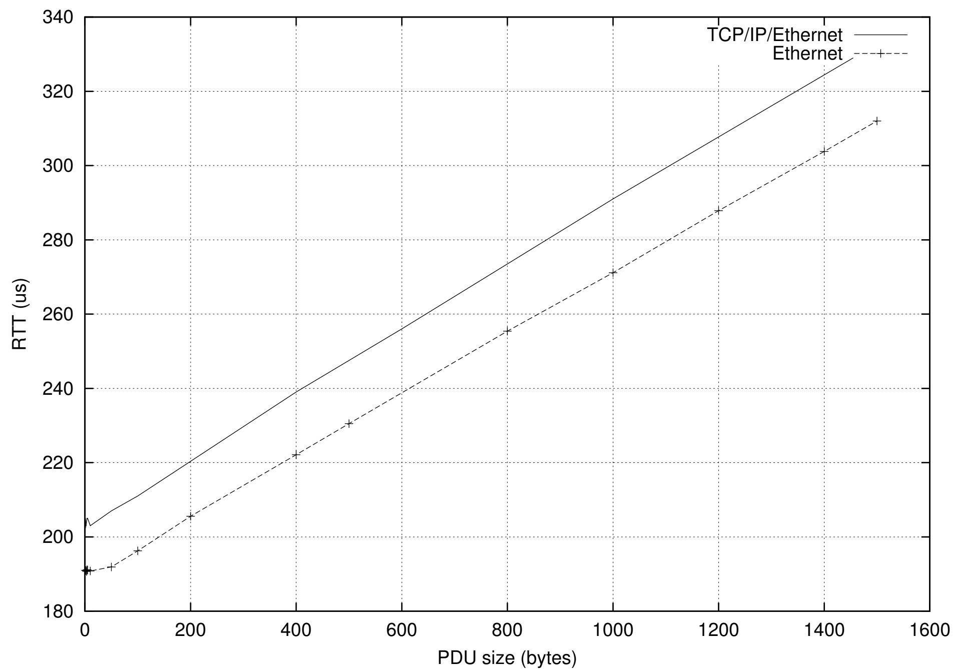


Figure 4.2: TCP and raw Ethernet round-trip times

4.4 Zero copy network IO

As seen in section 4.3 and in the previous chapter, the strict checking on access from user space to kernel space incurs a very high cost. For any data movement operation, the kernel has to copy the message from user space to kernel space, build protocol headers and copy the PDU to the NIC. Many works dealt with the problem by trying to shorten the critical path, some of them, like [22] and [23], dissected the send and receive process and showed the results outlined in Table 4.1 for the same NIC with and without DMA. The DMA seems to be a very attractive option for large transfers, almost all modern high speed Ethernet devices use DMA.

Operation for N words	**..using**	**Cost (μs)**
Copy from host memory to host memory	CPU	$0.03 \times N$
Copy from host memory to NIC	CPU	$0.25 \times N$
Copy from NIC to host memory	CPU	$0.57 \times N$
Copy from host memory to NIC	NIC DMA	$2.39 + 0.03 \times N$
Copy from NIC to host memory	NIC DMA	$2.00 + 0.03 \times N$
Interrupt handling		6.5
System call		1.0

Table 4.1: Cost of send and receive operations per word on Gigabit Ethernet

A few projects investigated ways to reduce the overhead of user space to kernel space switching, some considered implementing a reliable transport protocol on the NIC. Some modern NICs come with on-board CPUs and, sometimes, a significant amount of memory, some Ethernet NICs provide built-in circuitry to calculate TCP checksums and though Linux implements these routines in native CPU code, the NIC's *Application Specific Integrated Circuits* (APIC) are faster and may introduce a few microsecond gains. Most other projects considered *zero copy networking*, which is a way to eliminate user space to kernel space (and vice versa) memory copies.

Zero copy networking raises big challenges: it introduces deep changes to the traditional UNIX IO semantics [24] and kernel code assumes the persistence of data pages and is not designed to expect page faults while executing a system call. Implementing TCP/IP in a true zero copy fashion is very tricky, namely:

- the user space memory pages have to be "pinned down" so that they don't move while the IO operation is taking place,
- the user space memory pages have to be mapped to the NIC's IO addresses in the PCI IO address space,
- the user space memory pages may have to be set as *copy-on-write*, that is they are not changed until a write is issued to them, which would trigger the write to the NIC's IO addresses,

- some NICs are unable to perform DMA to high memory addresses and they would require *bounce buffers*, which are accessible intermediate buffers, introducing an extra memory copy,
- since much of the virtual memory operations are handled by hardware (paging and segmentation on Intel IA32 for example) which requires word-aligned[4] data blocks (and page-aligned segments), the kernel has to insure alignment. It turns out that Ethernet header (14 bytes), which is clearly not word-aligned, does require extra work,
- on reads, the kernel has to synchronize with the application, because the calling process may not be waiting on a `read()` and the buffer may not be large enough.

We found zero copy networking a very exciting and promising research area, but after spending a lot of time understanding all its intricacies we discovered that it's much harder to implement then it may sound. Zero copy test implementations exist since the 80's but none of them knew widespread.

4.5 Conclusion

The transport layer overheads measures have shown that the TCP overhead isn't significant, therefore it would make sense to use TCP-based communications in a cluster of compute nodes. The send and receive latencies are quite high which suggests that the communication using short messages would be less efficient than bulk transfers.

[4]The DMA subsystem also imposes too many restrictions: word-aligned blocks, of contiguous physical memory and, for some hardware, below the 16 MB limit.

Chapter 5

The Storage Sub-system

"The steady state of disks is full"
— Ken Thompson

It's almost inevitable to access files for most applications. Besides, most modern science and engineering applications manipulate some huge sets of data and, therefore, put a heavy stress on the file system. Another annoying fact about storage devices is that they're still using relatively very slow mechanics which makes them definitely the bottleneck in processing with intensive disk IO.

5.1 The file system — part of the kernel

UNIX operating systems rely heavily on the file system which is designed to keep track of many system parameters like file ownership, access time, type, etc. The performance problems related to handling files generally do not pertain to the file system itself but to the underlying disks subsystem. The disks are known to be very sensitive to concurrent access which occurs very often on multi-tasking systems like UNIX.

RAID 0 is an easy way to introduce more parallelism in disks operations. Indeed, striping files over multiple disks reduces the probability of having more than a single access occurring to a particular disk. Besides, the asynchronous nature of disks operation lets the operating system issue multiple requests and have multiple disks execute them simultaneously.

A node in a cluster might need to share or access files on other nodes. The *Network File System* (NFS) is the most popular file sharing protocol among UNIX systems. A few clusters are using PVFS, but it's a high-level file system that falls into distributed environments rather than file systems for data storage.

5.2 RAID 0

RAID stands for Redundant Array of Inexpensive Disks, it's a way of grouping disks together to make them look like a single contiguous block device from the UNIX operating system point of view. RAID arrays are used in business environments mostly for securing data storage by providing redundancy of devices, data and metadata. This makes hardware failures much less fatal to data and recovery from moderate to severe hardware failures can be guaranteed. Some vendors provide devices that implement the RAID algorithms in hardware, the operating system would be presented with a plain SCSI device and all RAID operations wouldn't involve the OS in any way.

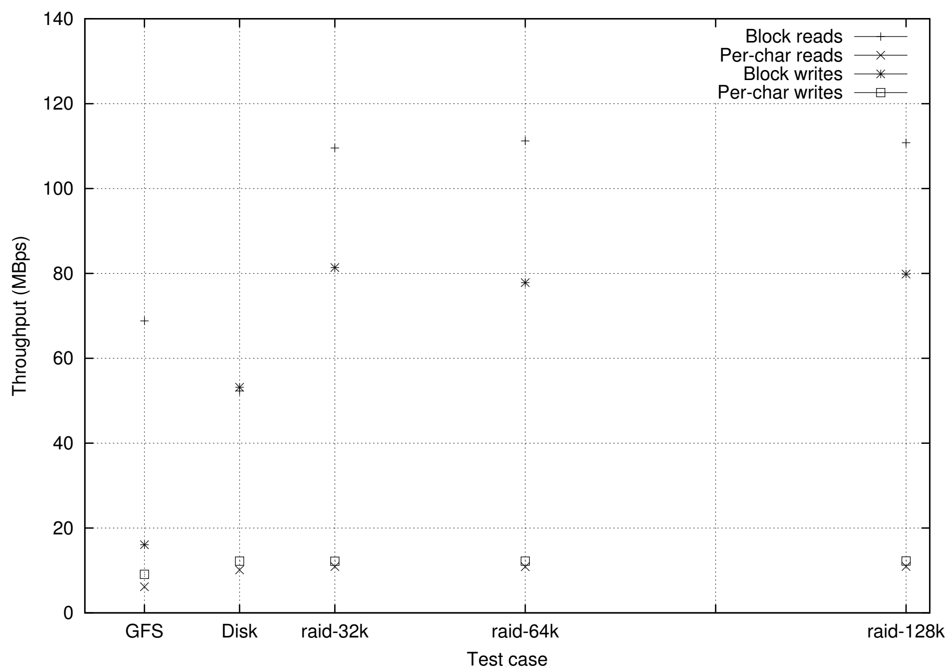


Figure 5.1: Throughput of different file systems and RAID arrays compared

Most modern UNIX systems come with a RAID support within the kernel so that it would treat a set of disks as if they were a single device but the underneath IO operations would be performed according to the RAID specifications. Data striping is an easy way of increasing the utilization of a RAID array, it consists into spreading a data block write among all the disks of the set in a round robin fashion. This data access schema pipelines SCSI commands issued to different disks, besides, modern

SCSI devices can do *command tagged queuing* which is basically, the asynchronous execution of SCSI command. The results shown in figures 5.1 and 5.2 were measured using the configuration outlined in Appendix D.2.

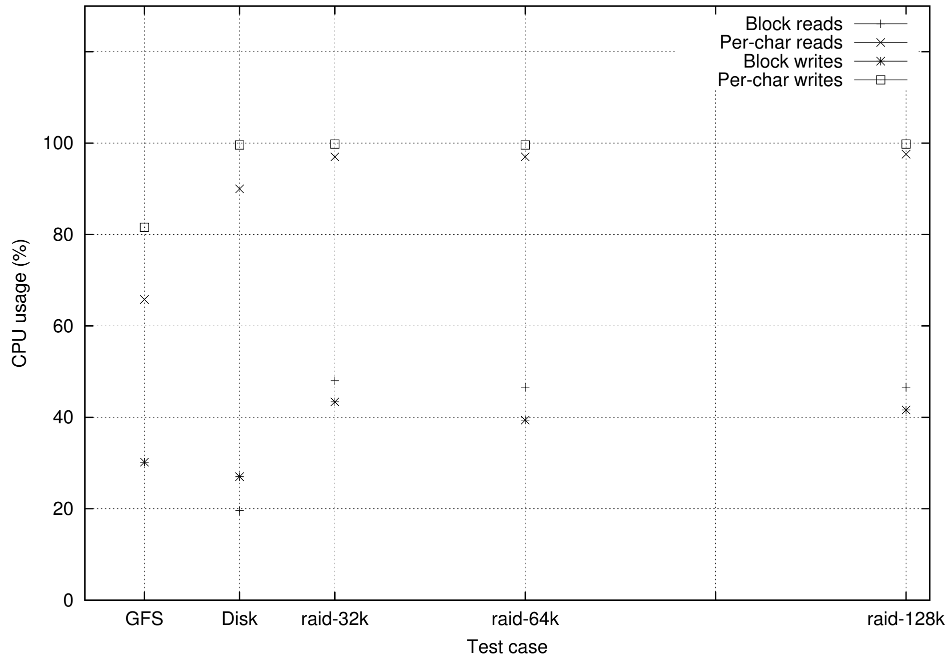


Figure 5.2: CPU usage of different file systems and RAID arrays compared

Figure 5.1 shows the throughput of different file systems over different configurations of disks. It suggests that:

- the RAID block size doesn't seem to affect the throughput,
- the RAID array performs as good as a single disk on per-char operations,
- the RAID array performs better than a single disk in block IO,
- block IO operations are performed much faster than per-char operations, block reads in particular, seem to saturate the PCI bus.

SCSI disks operate on blocks, usually 512 bytes long, and each IO operation incurs an overhead due to the preparation of buffers, the issuing of the actual SCSI command and the wait till completion. This kind of overhead is a one-time cost and is the same for

1 byte and for 4096 KB, that's why block IO goes faster then per-char IO. Figure 5.2 shows the CPU usage of different file systems over different configurations of disks. It suggests that:

- the RAID block size doesn't seem to affect the CPU usage,
- block IO operations tie the CPU much less than per-char operations,
- the in-kernel mapping of RAID 0 blocks to actual disk blocks about doubles the CPU usage, but it stays much lower with respect to single disk file systems,
- because of the very high aggregate speed of the RAID array, the CPU becomes fully utilized on per-char operations, actually, it becomes the bottleneck.

It's clear that application developers would better use block IO and per-byte IO should be avoided by all means (like ring buffers). RAID 0 introduces a reasonable cost in CPU usage, but it allows for very fast reads. The ability to have a single large file system comes very handy with large files which are very common in scientific environments.

5.3 NFS — the Network File system

Sun Microsystems published the Request for Comment (RFC) 1094 on March 1989 specifying NFS as a simple network file system. NFSv3 came as RFC 1813 and NFSv4 as RFC 3010, but the most widely used NFS clients and servers implement the original NFSv2 as specified in RFC 1094 and, to a lesser extent, NFSv3. The first design goal of NFS was to come up with a simple and fault-tolerant network file system without real focus on performance or security, but the openness of the standard and its simplicity made NFS the ubiquitous UNIX network file system. The SPEC System File Server Suite 97 (version 2.0) were ran by [20], we didn't have access to this commercial test suite, but our relatively simplistic tests experience didn't contradict with their results. The environment is comprised of one NFS server and five clients as described in D.4. The *load* is the total number of requests that reach the server and the *response time* is how fast the server could successfully service a request.

5.3.1 User space NFS

Almost all NFS server implementations run in user space, mostly because, until recently, the need for very high-speed file serving didn't show up. Though the user space NFS server running under Linux was praised for satisfactory performance in business environments, it didn't keep up with a relatively high load for a scientific computation environment [20].

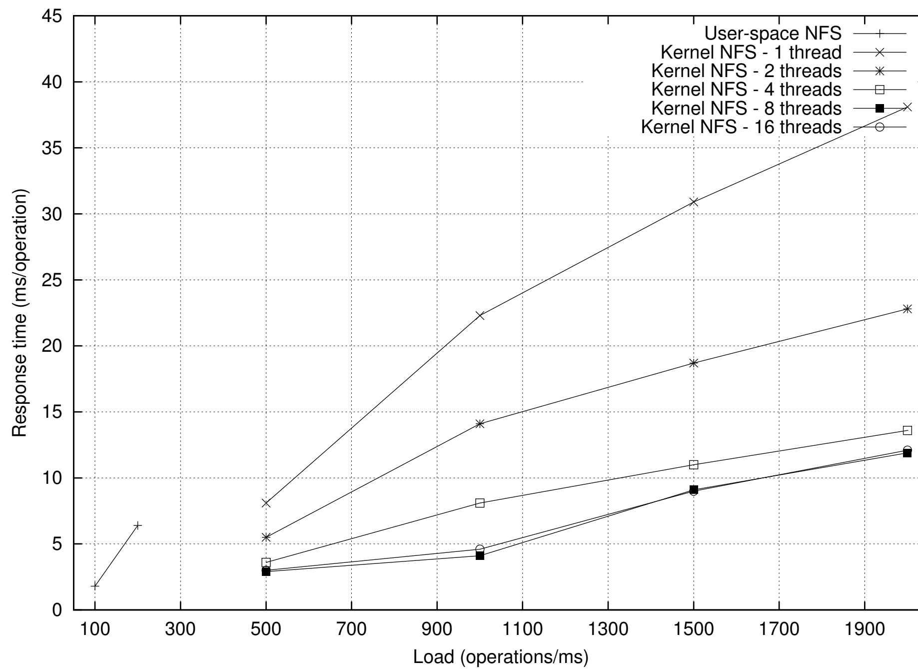


Figure 5.3: Response time of user-space NFS vs. in-kernel NFS [20]

5.3.2 Kernel space NFS — Shortening the critical path

In-kernel servers are a very tentative approach to achieve very low IO latencies, Linux has at least one implementation of an in-kernel web server (tux) and one of an in-kernel NFS server (knfsd). Figure 5.3 shows that a number of server threads large enough can reduce response times significantly, the parallel request processing resulting from multiple threads is responsible for the high performance. Figure 5.4 shows that beyond 1100 request per millisecond coming from 5 different clients, the FastEthernet network saturates and the network starts loosing packets. This shows as a drop in the number of successfully completed operations per millisecond since some requests and some responses get lost on the way.

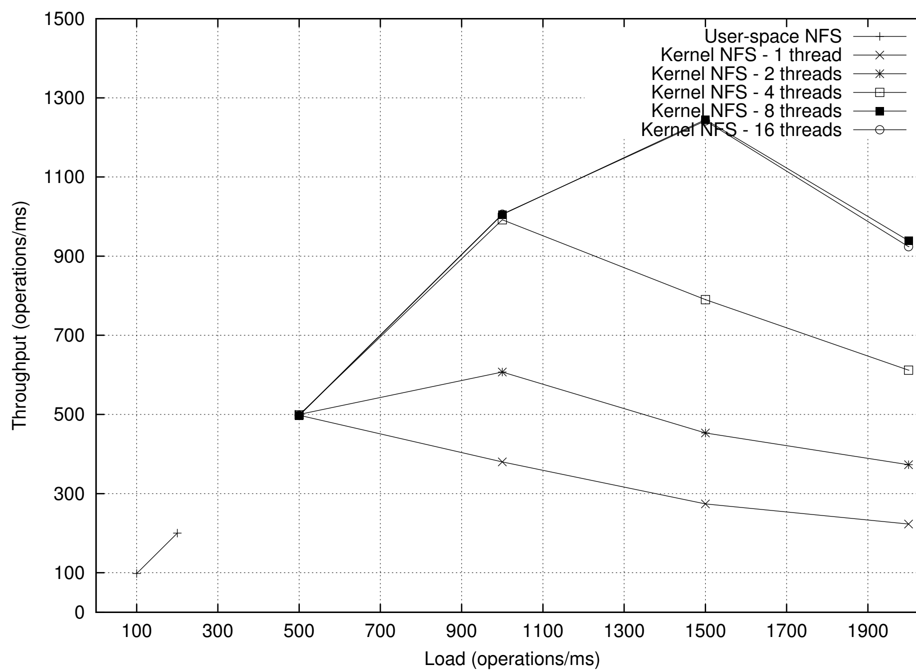


Figure 5.4: Throughput of user-space NFS vs. in-kernel NFS [20]

5.4 Zero copy.. again

Zero copy is much easier to implement with file IO than with network IO. File operations involve reading and writing data blocks with absolutely no changes to the data, whereas network IO requests involve a lot of kernel processing for the data (packets) to

be sent or received. File system reads happen only when the process requests them, so when the kernel gets a block read, it knows that the process is waiting for it and delivers it immediately, whereas network reads may come "unsolicited" and require extra work to synchronize the kernel and the user process.

```
[ichihi@tux imports]$ cat /proc/4785/maps
08048000-080c7000 r-xp 00000000 03:03 176610 /bin/bash
080c7000-080cd000 rw-p 0007e000 03:03 176610 /bin/bash
080cd000-0810e000 rwxp 00000000 00:00 0
40000000-40012000 r-xp 00000000 03:03 176378 /lib/ld-2.2.5.so
40012000-40013000 rw-p 00012000 03:03 176378 /lib/ld-2.2.5.so
40013000-40014000 r--p 00000000 03:0a 835719 /usr/lib/locale/ar_TN/LC_IDENTIFICAT
ION
40014000-40015000 r--p 00000000 03:0a 852115 /usr/lib/locale/ar_TN/LC_MEASUREMENT
40015000-40016000 r--p 00000000 03:0a 835726 /usr/lib/locale/ar_TN/LC_TELEPHONE
40016000-40017000 r--p 00000000 03:0a 852597 /usr/lib/locale/ar_TN/LC_ADDRESS
40017000-40018000 r--p 00000000 03:0a 852112 /usr/lib/locale/ar_TN/LC_NAME
40018000-40019000 r--p 00000000 03:0a 852103 /usr/lib/locale/ar_TN/LC_PAPER
40019000-4001a000 r--p 00000000 03:0a 852100 /usr/lib/locale/ar_TN/LC_MESSAGES/SY
S_LC_MESSAGES
4001a000-4001b000 r--p 00000000 03:0a 835724 /usr/lib/locale/ar_TN/LC_MONETARY
4001b000-40021000 r--p 00000000 03:0a 852596 /usr/lib/locale/ar_TN/LC_COLLATE
40021000-40022000 r--p 00000000 03:0a 852093 /usr/lib/locale/ar_TN/LC_TIME
40022000-40023000 r--p 00000000 03:0a 852110 /usr/lib/locale/ar_TN/LC_NUMERIC
40023000-4002c000 r-xp 00000000 03:03 176413 /lib/libnss_files-2.2.5.so
4002c000-4002d000 rw-p 00008000 03:03 176413 /lib/libnss_files-2.2.5.so
4002d000-40030000 r-xp 00000000 03:03 176608 /lib/libtermcap.so.2.0.8
40030000-40031000 rw-p 00002000 03:03 176608 /lib/libtermcap.so.2.0.8
40031000-40032000 rw-p 00000000 00:00 0
40032000-40034000 r-xp 00000000 03:03 176393 /lib/libdl-2.2.5.so
40034000-40035000 rw-p 00001000 03:03 176393 /lib/libdl-2.2.5.so
40035000-40153000 r-xp 00000000 03:03 176389 /lib/libc-2.2.5.so
40153000-40158000 rw-p 0011e000 03:03 176389 /lib/libc-2.2.5.so
40158000-4015c000 rw-p 00000000 00:00 0
4015c000-40187000 r--p 00000000 03:0a 852118 /usr/lib/locale/ar_TN/LC_CTYPE
401a1000-401aa000 r-xp 00000000 03:03 176421 /lib/libnss_nisplus-2.2.5.so
401aa000-401ab000 rw-p 00008000 03:03 176421 /lib/libnss_nisplus-2.2.5.so
401ab000-401bc000 r-xp 00000000 03:03 176397 /lib/libnsl-2.2.5.so
401bc000-401bd000 rw-p 00011000 03:03 176397 /lib/libnsl-2.2.5.so
401bd000-401bf000 rw-p 00000000 00:00 0
401bf000-401c8000 r-xp 00000000 03:03 176418 /lib/libnss_nis-2.2.5.so
401c8000-401c9000 rw-p 00008000 03:03 176418 /lib/libnss_nis-2.2.5.so
bfffb000-c0000000 rwxp ffffc000 00:00 0
```

Figure 5.5: Memory mapped files for a Linux shell (/bin/bash)

POSIX.1b specifies the `mmap()` system call, it's a way of mapping a file's blocks to a virtual memory region. This is possible because the file system operations use the *page cache*. All reads and writes to a memory mapped region will result into actual reads and writes to the file. Using memory mapped IO usually boosts file access speeds. Many modern Unix operating systems like Linux and Solaris use mmap() intensively, usually all libraries and system files are "mmap-ped" as shown in Figure 5.5. We made some changes to the source code of bonnie, a common disk benchmarking utility, to make it use memory-mapped files. The test environment is described in D.5 and the patch is listed in C.2.

Figure 5.6 shows the regular bonnie tests with normal results compared to the performance with memory-mapped IO on a 1GB file. The memory-mapped IO seems to introduce some acceleration that is significant in block reads. The mmap() test shown

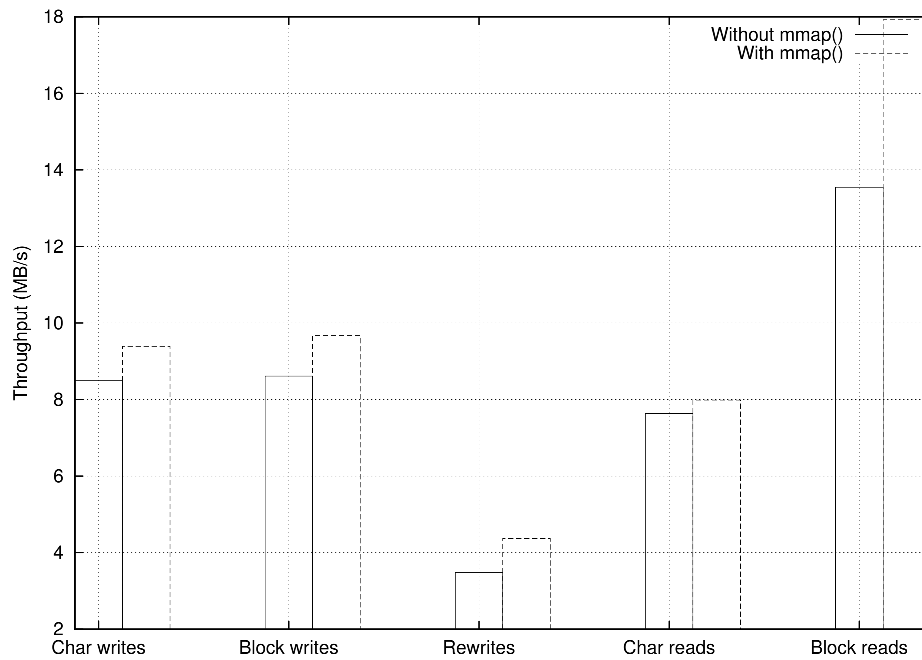


Figure 5.6: Bonnie results with and without mmap()

in Figure 5.7 looks abnormal because of a common problem with some IDE boards. Many IDE disks tend to "lie" answering a sync command, that is they would acknowledge a sync command without performing a synchronization with the buffers.

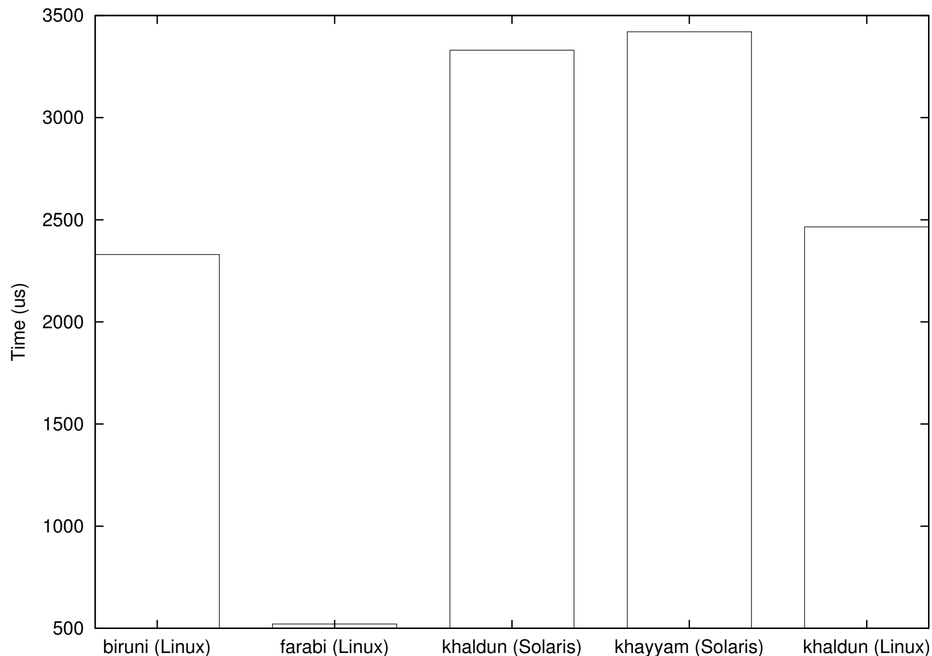


Figure 5.7: mmap() latencies

5.5 Conclusion

RAID 0 seems to be a simple and very efficient way to achieve very high disk IO throughput. Zero copy disk IO, through mmap(), is very easy to use and seems to give substantial performance enhancements over regular disk IO. Besides, it's very well implemented because it's heavily used by critical sub-systems of the Unix operating system. Per-character IO ties the CPU too much, therefore it has to be avoided as much as possible, popular programming techniques permit to perform sequential operations while actually doing block IO like ring buffering (double buffering).

NFS is the most popular file sharing facility on Unix environments but it performs very badly under heavy loads. The in-kernel NFS service and GFS may provide a viable alternative under some circumstances. kNFSd lacks many capabilities but has

the benefit of being very accessible and relatively easy to administer whereas GFS is a featureful and reliable file system but it's relatively expensive and more complex.

Conclusions

The aim of this work was far from building a clustering environment. However, the results displayed here can be valuable input to cluster designers who would size how much performance boost they can expect from a particular development. It also points the hardware factors to which programmers and systems administrators have to pay particular attention.

PC systems seem to perform better than traditionally more powerful workstations, we believe that all the marketing attention that the Personal Computer got helped develop its components at a much faster pace. Besides, the massive production helped drop prices dramatically. On the operating system side, Linux is performing pretty well because it's development has always been free of any non-technical constraints so over the last 10 years it has evolved solely with the goal of achieving outstanding feature set and performance.

This work lacks the evaluation of communication mechanisms and high-level libraries to give a sustainable comparison with supercomputers. Indeed, there are many communication facilities that were developed for cluster computing like Distributed Shared Memories and Remote DMA. Besides, we would need to study the behavior of different computing workloads and the expected speedup, scientific applications have very sparse communication and computation patterns and that would govern their speedup when ran on a computer cluster.

A nice addition to this work would be the evaluation of a high-level programming library like MPI over a high-performance communication mechanism like zero-copy. We have shown, though, that TCP/IP overhead is not significant when compared to all the programming facilities is provides.

Acknowledgment

We would like to express our gratefulness to Dr. Ahmed Elleuch and Pr. Farouk Kamoun for their valuable input and comments. Many thanks go to Dr. Ahmed Elleuch for his dedication to bring this work to its present status. Very little could have been done without the wonderful software produced by the Open Source community and used for development, document processing and performance measurement all over this project, special thanks go to Linus Torvalds et al., Matthias Ettrich and Larry McVoy.

Appendix A

CPU Optimizations

- Pipelining
  The execution of an instruction requires a number of clock cycles depending on its complexity. A typical instruction requires four phases: *fetch*, *decode*, *execute* and *write-back*. Early processors used to issue one instruction and wait till it undergoes the four phases resulting in wasting time in wait states. The pipeline, deploys a number of units to perform the phases in parallel. That is, a four-stage pipeline (with four units) can fetch instruction *i*, decode instruction $i+1$, execute instruction $i + 2$ and write-back instruction $i + 3$ simultaneously. Under perfect conditions, the pipeline can issue one instruction per cycle. But, in particular conditions, it is impossible to keep all the stages usefully busy simultaneously. Indeed, dependencies between successive instructions might disallow pipelining, branching is also a major factor that slows down the pipeline because when a branching occurs the control unit has to flush the pipeline wasting the anticipated processing.

- Out-of-order execution
  Consists into breaking the instruction's sequential order to keep the pipeline stages busy all the time. Instruction $i + 1$ may enter the pipeline before instruction i is issued [1]. A complex lookahead mechanism must be implemented to calculate dependencies between instructions.

- Branch prediction
  Used in loops, where there is always a conditional jump. The execution unit assumes that the condition will be true and the branch will occur and anticipates jump address calculations. This anticipation speeds up jumps and it slows it down when the condition is false but the overall performance is improved in most cases [1].

- High clock frequency

Increasing the clock rate reduces the base cycle of a CPU and yields to improved performances. But the overall improvements are not linear with clock rates because they depend on many more parameters especially bus rate.

- Software optimizations
  Some CPU designs rely on complier-generated optimizations to achieve high performances. For example, RISC processors assume that the compiler will make an efficient use of the instructions set [6][1].

- Cache splitting
  Consists into using separate caches for data (D-cache) and for instructions (I-cache). The principle of locality applies to data and instructions independently, that is the working set in terms of instructions is different from the one for data [1]. Hence, using separate caches will improve cache hits. Cache splitting is typical of RISC processors, but some CISC use it too.

Appendix B

An Example Supercomputer: Intel Paragon

This appendix gives a very brief description of some of the Intel Paragon sub-systems to show how different the design is from commodity systems.

The most obvious way to achieve high-performance computing is parallel processing. Custom parallel computers use very complex technologies to achieve high performance, hence they are very costly. We can distinguish two main categories of parallel computers: *Massively Parallel Processors* (MPP) and *Symmetrical Multi-Processors* (SMP). MPP are built around a large set of processors (between 64 and 65536) which are interconnected together with a high-speed network, each processor may have its own memory, SMP are smaller scale computers using fewer processors (2 to 64) which are interconnected *via* a bus and share the same memory.

As inter-processor communications are critical to parallel computation performance, MPP design relies on high-speed networking techniques. Most of MPP are made up of a switched network of CPUs, the switching components are fast single-chip switches employing cut through routing with short wire delays and wide links. The network interface is, in bus terms, "close" to the processor, typically on the processor-memory bus, rather than on a standard IO bus [17].

Since all massively parallel processors are custom computers we won't discuss all the different architectures. In the next, we'll present one example machine: the Intel Paragon. Intel made its first steps in the world of parallel computers through the iPCS. The Paragon, which is the successor of the iPCS, can use up to 2048 processors which sums up to 150 GFlops. Each node is comprised of two processors and nodes are connected in an array topology. The layout of a node is outlined in Figure B.1.

The components of one node are interconnected trough two buses operating at a peak bandwidth of 400 MBps: one address bus (32-bit width) and one data bus (64-bit

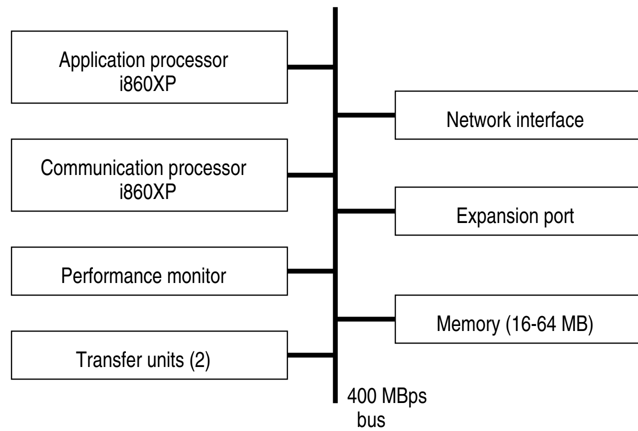


Figure B.1: Architecture of a node in Intel Paragon.

width). Since a dedicated processor performs IO operations, processing and communications could be overlapped and causes the calculation processor to spend much more time on processing then what it would do if it were taking in charge communications too.

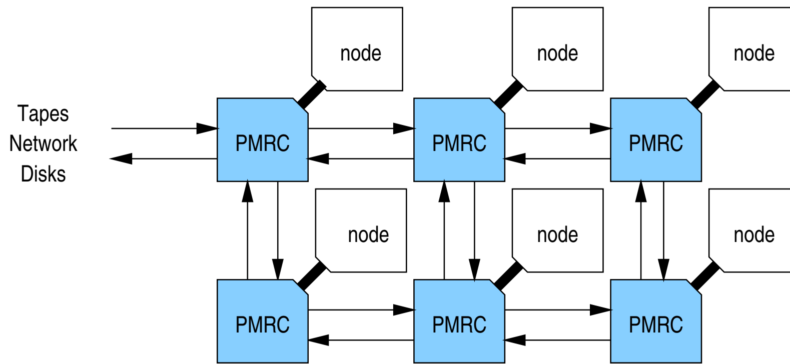


Figure B.2: Paragon interconnect.

Each node is connected to the interconnection network with a *Paragon Mesh Routing Chip* (PMRC) as shown in Figure B.2. Routing chips are connected together into a mesh through a two-way network operating at a peak speed of 200 MBps. A node in the Paragon could perform one of the following tasks: calculation, service or IO. Calculation nodes perform purely parallel processing, service nodes provide the UNIX (OSF/1 UNIX) services for program development and compilation and IO nodes perform the access to the storage devices.

Appendix C

Source code

C.1 The pf_packet benchmark

This is the source code for the raw packet round-trip time measurement, it provides a useful resource for low level network programming.

C.1.1 The sender (rp_send.c)

```
/* rp_send.c - Raw packet send
 * Imed Chihi <imed.chihi@ensi.rnu.tn>, <imed@linux.com.tn>
 * 2001-10-27
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public
 * License along with this program; if not, write to the Free
 * Software Foundation, Inc., 59 Temple Place, Suite 330, Boston,
 * MA 02111-1307, USA
 */
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <features.h>    /* for the glibc version number */
#include <sys/time.h>
#include <netinet/in.h>
#if __GLIBC__ >= 2 && __GLIBC_MINOR >= 1
#include <netpacket/packet.h>
#include <net/ethernet.h>     /* the L2 protocols */
#else
```

```
#include <asm/types.h>
#include <linux/if_packet.h>
#include <linux/if_ether.h>   /* The L2 protocols */
#endif
#define MAX_MSG_SIZE          ETH_DATA_LEN
int main(int argc, char **argv)
{
  int rp_socket, i, j, in_buf_size, out_buf_size, my_add_size, paddr_size;
  short  rp_protocol;
  struct sockaddr_ll ll_myaddr;
  struct sockaddr_ll ll_paddr;
  struct ethhdr hw_header;
  char in_buf[ETH_HLEN + MAX_MSG_SIZE], out_buf[ETH_HLEN + MAX_MSG_SIZE];
  char dummy[MAX_MSG_SIZE];
  struct timeval t_start, t_now;
  int res;
  /* Various sizes for the packet */
  int packet_sizes[13] = {1, 2, 10, 50, 100, 200, 400, 500, 800, 1000, 1200, 1400, 1500};
  /* My MAC address, replace with yours */
  char hw_myaddr[ETH_ALEN] = {0x00, 0xD0, 0xB7, 0x81, 0x2A, 0xF8};
  /* The peer's MAC address, replace with yours */
  char hw_paddr[ETH_ALEN] = {0x00, 0xD0, 0xB7, 0x81, 0x29, 0xE6};
  /* Filling the buffer with recognizable junk */
  dummy[0] = '0'; dummy[1] = '0';
  for(i=2; i<MAX_MSG_SIZE; i++) {
    dummy[i] = (char) (i%80 + 32);
  }
  /* Opening a PF_PACKET (aka AF_PACKET) socket. We picket DECnet as
   * the low level network protocol, so we don't interfere with
   * production network traffic */
  rp_protocol = ETH_P_DEC;
  rp_socket = socket(PF_PACKET, SOCK_DGRAM, htons(rp_protocol));
  if (rp_socket == -1) {
    perror("socket()");
    return 1;
  }

  ll_myaddr.sll_family = AF_PACKET;
  ll_myaddr.sll_protocol = htons(rp_protocol);
  ll_myaddr.sll_ifindex = 3; /* lo=1, eth0=2, eth1=3, etc. */
  ll_myaddr.sll_halen = ETH_ALEN;
  memcpy(ll_myaddr.sll_addr, hw_myaddr, ETH_ALEN);
  ll_paddr.sll_family = AF_PACKET;
  ll_paddr.sll_protocol = htons(rp_protocol);
  ll_paddr.sll_ifindex = 3; /* lo=1, eth0=2, eth1=3, etc. */
  ll_paddr.sll_halen = ETH_ALEN;
  memcpy(ll_paddr.sll_addr, hw_paddr, ETH_ALEN);
  if (bind(rp_socket, (struct sockaddr *)&ll_paddr, sizeof(ll_paddr)) == -1) {
    perror("bind()");
    close(rp_socket);
    return 1;
  }

  /* Build header */
  memcpy(hw_header.h_dest, hw_paddr, ETH_ALEN);
  memcpy(hw_header.h_source, hw_myaddr, ETH_ALEN);
  hw_header.h_proto = htons(rp_protocol);
  memcpy(out_buf, &hw_header, sizeof(hw_header));
  memcpy(out_buf + sizeof(hw_header), dummy, sizeof(dummy));
```

```
  for (j=0; j<13; j++) {
    gettimeofday(&t_start, NULL);
    for(i=0; i<1000000; i++) {
      sendto(rp_socket, out_buf, packet_sizes[j], 0, \
      (struct sockaddr *)&ll_paddr, sizeof(ll_paddr));
      in_buf_size = recvfrom(rp_socket, in_buf, MAX_MSG_SIZE, 0, \
      (struct sockaddr *)&ll_paddr, &paddr_size);
    }
    gettimeofday(&t_now, NULL);
    printf("RTT of 1000000 frames of %d bytes took: %ld s %ld us\n", \
    packet_sizes[j], t_now.tv_sec - t_start.tv_sec, t_now.tv_usec \
    - t_start.tv_usec);
  }
  close(rp_socket);
  return 1;
}
```

C.1.2 The receiver (rp_receive.c)

```
/* rp_receive.c - Raw packet receive
 * Imed Chihi <imed.chihi@ensi.rnu.tn>
 * 2001-10-27
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public
 * License along with this program; if not, write to the Free
 * Software Foundation, Inc., 59 Temple Place, Suite 330, Boston,
 * MA 02111-1307, USA
 *
 */
#include <sys/socket.h>
#include <unistd.h>
#include <stdio.h>
#include <netinet/in.h>
#include <string.h>
#include <features.h>    /* for the glibc version number */
#if __GLIBC__ >= 2 && __GLIBC_MINOR >= 1
#include <netpacket/packet.h>
#include <net/ethernet.h>     /* the L2 protocols */
#else
#include <asm/types.h>
#include <linux/if_packet.h>
#include <linux/if_ether.h>   /* The L2 protocols */
#endif
#define MAX_MSG_SIZE      ETH_DATA_LEN
int main(int argc, char **argv)
{
  int rp_socket, paddr_size, i, j, in_buf_size, out_buf_size, res;
  short  rp_protocol;
  struct sockaddr_ll ll_myaddr;
```

```
    struct sockaddr_ll ll_paddr;
    struct ethhdr hw_header;
    char hw_myaddr[ETH_ALEN] = {0x00, 0xD0, 0xB7, 0x81, 0x29, 0xE6};
    char hw_paddr[ETH_ALEN] = {0x00, 0xD0, 0xB7, 0x81, 0x2A, 0xF8};
    char in_buf[MAX_MSG_SIZE], out_buf[MAX_MSG_SIZE];
    int packet_sizes[13] = {1, 2, 10, 50, 100, 200, 400, 500, 800, 1000, 1200, 1400, 1500};
    rp_protocol = ETH_P_DEC;
    rp_socket = socket(PF_PACKET, SOCK_DGRAM, htons(rp_protocol));
    if (rp_socket == -1) {
      perror("socket()");
      return 1;
    }

    ll_myaddr.sll_family = AF_PACKET;
    ll_myaddr.sll_protocol = htons(rp_protocol);
    ll_myaddr.sll_ifindex = 4; /* lo=1, eth0=2, eth1=3, eth2=4, etc. */
    ll_myaddr.sll_halen = ETH_ALEN;
    memcpy(ll_myaddr.sll_addr, hw_myaddr, ETH_ALEN);
    if (bind(rp_socket, (struct sockaddr *)&ll_myaddr, sizeof(ll_myaddr)) == -1) {
      perror("bind()");
      close(rp_socket);
      return 1;
    }

    memset(&ll_paddr, 0, sizeof(ll_paddr));
    paddr_size = sizeof(ll_paddr);
    for(j=0; j<13; j++) {
      printf("Listening for %d bytes\n", packet_sizes[j]);
      for(i=0; i<1000000; i++) {
        in_buf_size = recvfrom(rp_socket, in_buf, packet_sizes[j], 0, \
        (struct sockaddr *)&ll_paddr, &paddr_size);

        sendto(rp_socket, in_buf, in_buf_size, 0, \
        (struct sockaddr *)&ll_paddr, sizeof(ll_paddr));
      }
    }
    close(rp_socket);
    return 0;
  }
```

C.2 The bonnie-mmap patch

To apply this patch, just uncompress the bonnie tar ball and run patch using:

```
$ tar xvzf bonnie.tar.gz
$ cd bonnie
$ patch -p1 < ../bonnie-mmap.patch
```

Here are the contents of bonnie-mmap.patch:

```
--- bonnie.vanilla/Bonnie.c     Wed Aug 28 17:23:49 1996
+++ bonnie/Bonnie.c     Tue Nov  5 08:00:00 2002
@@ -7,6 +7,7 @@
   *
   * COPYRIGHT NOTICE:
```

```
  * Copyright (c) Tim Bray, 1990-1996.
+ * mmap patch by Imed Chihi <imed@linux.com.tn>
  *
  * Everybody is hereby granted rights to use, copy, and modify this program,
  *  provided only that this copyright notice and the disclaimer below
@@ -29,6 +30,7 @@
 #include <fcntl.h>
 #include <sys/types.h>
 #include <sys/time.h>
+#include <sys/mman.h>
 #if defined(SysV)
 #include <limits.h>
 #include <sys/times.h>
@@ -50,6 +52,7 @@
 #define UpdateSeek (10)
 #define SeekProcCount (3)
 #define Chunk (16384)
+#define MmapSize (96*1024*1024) /* Memory mapped region for file IO */

 /* labels for the tests, used as an array index */
 typedef enum
@@ -443,9 +446,13 @@
     if (unlink(name) == -1 && *fd != -1)
       io_error("unlink");
     *fd = open(name, O_RDWR | O_CREAT | O_EXCL, 0777);
+    mmap(NULL, MmapSize, PROT_READ | PROT_WRITE, MAP_PRIVATE, *fd, 0);
   } /* create from scratch */
   else
+  {
     *fd = open(name, O_RDWR, 0777);
+    mmap(NULL, MmapSize, PROT_READ | PROT_WRITE, MAP_PRIVATE, *fd, 0);
+  }

   if (*fd == -1)
     io_error(name);
```

Appendix D

The test environment

D.1 lmbench tests

We used lmbench 2.0 under various combinations of operating systems and machines. lmbench is a comprehensive operating system benchmark [26, 27], it provides a portable set of test tools to measure the performance of various OS mechanisms.

Host	Architecture	CPU	RAM	Caches	OS	Disks
farabi	IA32	Pentium II at 400 MHz	128 MB	L1 (16K I, 16K D) and L2 256K	Linux 2.4 (RedHat 7.2)	EIDE
biruni	IA32	2xPentium III at 850 MHz	1 GB	L1 (16K I, 16K D) and L2 256K	Linux 2.4	SCSI 3
khayyam	Sparc64	UltraSparc IIe at 500 MHz	1 GB	L2 256K	Solaris 8	EIDE
khaldun	Sparc64	UltraSparc IIi at 400 MHz	128 MB	L2 256K	Solaris 7	SCSI 2
khaldun	Sparc64	UltraSparc IIi at 400 Mhz	128 MB	L2 256K	Linux 2.4	SCSI 2

Table D.1: Tested combinations of hardware and operating systems

D.2 GFS and RAID tests

The tests were ran on a Dell PowerEdge 4400 dual Pentium III 850 MHz 1024 KB cache with 1 GB RAM running Linux 2.4.12-ac3, with a 5-disk RAID 0 array made of Seagate ST173404LC drives and connected to an Adaptec AIC-7899 which is an Ultra160/m 64-bit, 66 MHz PCI controller. The single-disk test was ran on a single Seagate ST173404LC drive, all cases used ext2 file system except GFS.

D.3 Raw Ethernet RTT

The raw Ethernet figures were measured between two Dell PowerEdge 4400 with dual Pentium III Xeon 1024 KB cache machines running Linux 2.4.12-ac4 kernel with 1 GB RAM and one Intel EtherExpress Pro/1000 Gigabit Ethernet adapter with full duplex set, it's a 32-bit, 33 MHz PCI device. The Ethernet NIC's corresponding device driver uses DMA transfers to and from host memory. The Gigabit Ethernet switch is a PacketEngines[1] PowerRail 5200, the test program used the `AF_PACKET` sockets in `SOCK_DGRAM` mode provided by Linux, the driver was e1000.o version 3.1.23 as distributed by Intel Corporation.

D.4 kNFSd tests

This test environment is comprised of one NFS server being a dual Pentium III Xeon Dell PowerEdge 4400 with 1024 KB cache, 1 GB RAM, 1 Intel EtherExpress FastEthernet NIC, 1 Adaptec AIC7899 SCSI Ultra3/m controller and 1 Seagate ST336605LC disk. Five similar clients were running 2 processes, each running IO requests on an NFS-mounted file system from the server.

D.5 Bonnie tests

The bonnie tests, with regular IO and memory-mapped IO were ran on a HP OmniBook XE3 laptop with a Pentium IIIm and 256 MB RAM, an Intel 82801CAM IDE U100 disk controller and a IC25N030ATCS04-0 30 GB IDE hard disk. The OmniBook was running RedHat Linux kernel 2.4.18-10 and the tests were ran on a 1GB file.

[1] Now Alcatel.

Bibliography

[1] KAI HWANG, *Advanced Computer Architecture*, McGraw-Hill 1993.

[2] D. PATTERSON, J. HENNESSY, *Computer Architecture – A Quantitative Approach*, Addison Wesley.

[3] DAVID RUSLING, *The Linux Kernel*, http://www.tldp.org, 1998.

[4] MATTIAS BLUMRICH, *Network Interface for Protected, User-level Communication*, Department of Computer Science, Princeton University, June 1996.

[5] HANK DIETZ, *Linux Parallel Processing Howto*, http://www.tldp.org, January 1998.

[6] ROBERT YUNG, *Evaluation of a Commercial Microprocessor*, Sun Microsystems Laboratories Inc. and University of California, Berkeley, June 1998.

[7] ZEITNET CORP., *High Performance PCI Bus ATM Adapter Design – A White Paper*, February 1998.

[8] A. BECHTOTSHEIM, F. BASKETT, V. PRATT, *The Sun Workstation Architecture*, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, March 1982.

[9] WERNER ALMESBERGER, *High-speed ATM Networking on Low-end Computer Systems*, Laboratoire de Réseaux de Communication (LRC), Ecole Polytechnique Fédérale de Lausanne, Switzerland, 1995.

[10] D. BOGGS, J. MOGUL, CH. KENT, *Measured Capacity of an Ethernet: Myths and Reality*, Western Research Laboratory, Digital Equipment Corporation, September 1988.

[11] QUANTUM CORPORATION, *Hard Drives and Computer System Performance – A White Paper*, 1999.

[12] INTEL CORPORATION, *Intel486 Processor Family*, December 1994.

[13] ANDREW TANNENBAUM, *Modern Operating Systems*, Prentice Hall International Editions, 1992.

[14] JACK DONGARRA, TOM DUNIGAN, *Message-Passing Performance of Various Computers*, Oak Ridge National Laboratory, January 1997.

[15] PAUL GORTMAKER, *Linux Ethernet Howto*, http://www.tldp.org, July 1998.

[16] TIMOTHY GIBSON, ETHAN MILLER, *The Case for Personal Computers as Workstations*, Department of Computer Science and Electrical Engineering, University of Maryland-Baltimore Country.

[17] THOMAS ANDERSON, DAVID CULLER, DAVID PATTERSON, THE NOW TEAM, *A Case for NOW (Networks of Workstations)*, University of California at Berkeley, December 1994.

[18] M. COSNARD AND F. DESPREZ, *Quelques Architectures de Nouvelles Machines*, Calculateurs Parallèles, issue 21, March 1994.

[19] MICHAEL WILL, *Linux PCI Howto*, http://www.tldp.org, July 1996.

[20] HONGJIU LU, *NFS server in Linux: Past, Present and Future*, VA Linux Systems, August 1999.

[21] TOM SHANLEY, DON ANDERSON, *PCI System Architecture – Fourth Edition*, Addison Wesley, May 1999.

[22] SHINJI SUMIMOTO, HIROSHI TEZUKA, ATSUSHI HORI, HIROSHI HARADA, TOSHIYUKI TAKAHASHI, YUTAKA ISHIKAWA, *The Design and Evaluation of High Performance Communication using a Gigabit Ethernet*, Real World Computing Partnership, 1999.

[23] MATT WELSH, ANINDYA BASU, THORSTEN VON EICKEN, *ATM and Fast Ethernet Network Interfaces for User-level Communication*, Cornell University.

[24] MOTI N. THADANI, YOUSEF A. KHALIDI, *An Efficient Zero-Copy I/O Framework for UNIX*, Sun Microsystems Laboratories, May 1995.

[25] DANIEL P. BOVET, MARCO CESATI, *Understanding the Linux Kernel*, O'Reilly, January 2001.

[26] LARRY MCVOY, CARL STAELIN, *lmbench: Portable tools for performance analysis*, USENIX '96, January 1996.

[27] CARL STAELIN, LARRY MCVOY, *mhz: Anatomy of a micro-benchmark*, USENIX '98, January 1998.

[28] JAMAL HEDI SALIM, ROBERT OLSSON, ALEXEY KUZNETSOV, *Beyond Softnet*, Znyx Networks, Uppsala University/Swedish University of Agricultural Sciences, Swsoft/INR.

[29] ERIK HENDRIKS, *BProc: The Beowulf Distributed Process Space*, ACL, Los Alamos National Laboratory, 2001.

[30] STEVE SOLTIS, GRANT ERICKSON, KEN PRESLAN, MATTHEW O'KEEFE, TOM RUWART, *The Design and Performance of a Shared Disk File System for IRIX*, Department of Electrical and Computer Engineering and Laboratory for Computational Science and Engineering, University of Minnesota.

[31] MICHAEL LITZKOW, MIRON LIVNY, MATT MUTKA, *Condor - A Hunter of Idle Workstations*, Proceedings of the 8th International Conference of Distributed Computing Systems, pages 104-111, June, 1988.

[32] DENNIS W. DUKE, THOMAS P. GREEN, JOSEPH L. PASKO, *Research Toward a Heterogeneous Networked Computing Cluster: The Distributed Queuing System Version 3.0*, Supercomputer Computations Research Institute, Florida State University, Tallahassee, Florida.

[33] SUBRAMANIAN KANNAN, MARK ROBERTS, PETER MAYES, DAVE BRELSFORD, JOSEPH F. SKOVIRA, *Workload Management with LoadLeveler*, IBM Corporation.

[34] M. CORBATTO, *An introduction to Portable Batch System (PBS)*, Scuola Internazionale Superiore di Studi Avanzati, 2000.

[35] MICHAEL LITZKOW, TODD TANNENBAUM, JIM BASNEY, MIRON LIVNY, *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*, Computer Science Departement, University of Wisconsin-Madison.

[36] AL GEIST, ADAM BEGUELIN, JACK DONGARRA, WEICHENG JIANG, ROBERT MANCHEK, VAIDY SUNDERAM, *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994.

[37] WILLIAM GROPP, EWING LUSK, NATHAN DOSS, ANTHONY SKJELLUM, *A High-Performance, Portable Implementation of the Message Passing Interface Standard*, Argonne National Laboratory and Mississipi State University.

[38] RICHARD RASHID, ROBERT BARON, ALESSANDRO FORIN, DAVID GOLUB, MICHAEL JONES, DANIEL JULIN, DOUGLAS ORR, RICHARD SANZI, *Mach: A Foundation for Open Systems*, Proceedings of the Second Workshop on Workstation Operating Systems (WWOS2), September 1989.

[39] ANDREW S. TANENBAUM, GREGORY J. SHARP, *The Amoeba Distributed Operating System*, Vrije Universiteit.

[40] THOMAS STERLING, DONALD BECKER, MICHAEL BERRY, DANIEL SAVARESE, CHANCE RESCHKE, *Achieving a Balanced Low-Cost Architecture for Mass Storage Management through Multiple Fast Ethernet Channels in the Beowulf Parallel Workstation*, NASA, University of Maryland and the George Washington University.

[41] THOMAS STERLING, DONALD BECKER, JOHN DORBAND, DANIEL SAVARESE, UDAYA RANAWAKE, CHARLES PACKER, *Beowulf: A Parallel Workstation for Scientific Computation*, NASA, University of Maryland and Hughes STX.

[42] PHILIP BUONADONNA, ANDREW GEWEKE, DAVID CULLER, *An Implementation and Analysis of the Virtual Interface Architecture*, University of California at Berkeley.

[43] BENJAMIN LAHAISE, *An AIO Implementation and its Behaviour*, Proceedings of the Linux Ottawa Symposium, June 2002.

[44] PETER WONG, BADARY PULAVARTY, SHAILABH NAGAR, JANET MORGAN, JUNATHAN LAHR, BILL HARTNER, HUBERTUS FRANKE, SUPRANA BHATTACHARYA, *Improving Linux Block I/O for Enterprise Workloads*, Proceedings of the Linux Ottawa Symposium, June 2002.

[45] LARRY MCVOY, *lmbench: Portable Tools for Performance Analysis*, Silicon Graphics Inc.

[46] TOSHIYUKI MAEDA, *Safe Execution of User Programs in Kernel Mode Using Typed Assembly Language*, University of Tokyo, 2002.

[47] AARON BROWN, *A Decompositional Approach to Computer System Performance Evaluation*, Harvard University, 1996.