The University of New South Wales
School of Computer Science and Engineering

Virtualising Darwin on L4

Bachelor of Engineering (Computer Engineering)

February 20, 2007

Author: Joshua Root

Student ID: 3022016

E-mail: `jmr@cse.unsw.edu.au`

Supervisor: Prof. Gernot Heiser

Assessor: Charles Gray

Abstract

Virtualisation involves abstracting the interface between computer hardware and the operating systems that run on it. One real machine can be used to run several operating systems simultaneously, presenting a complete machine interface to each of them. Virtualisation is increasingly popular due to its usefulness for many tasks, such as server consolidation, debugging, improving security, and running legacy software.

I have added virtualisation features to the *Darbat* operating system, a version of Apple's Darwin OS that has been modified to run on the L4 microkernel. Multiple instances of the Darbat kernel can be executed simultaneously on a single computer system. Each instance is isolated from the others by hardware-enforced memory protection. System resources, including memory, CPU time, and peripheral devices, are shared between the Darbat instances in a controlled manner.

Acknowledgements

I would like to thank Charles Gray, for his invaluable guidance; Tom Birch and Geoffrey Lee, for their contributions to Darbat and their excellent debugging advice; and Gernot Heiser, for his high standards and for giving me the opportunity to work with his group on such a challenging and rewarding project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Darwin is an operating system maintained by Apple, Inc. It is most often encountered as the basis of Mac OS X. Darwin has been modified to run on top of the *L4* microkernel, and the resulting operating system has been named *Darbat*.

Virtualisation is the abstraction of the interface between operating system kernels and computer hardware. It enables multiple kernels, along with their user environments, to be executed on a single physical computer. Just as importantly, the kernels and their applications are isolated from each other, so that a fault in one need not affect the others.

In this thesis, my goal has been to implement virtualisation features for the Darbat operating system. I have made it possible to run several Darbat kernels simultaneously, and have implemented a system whereby they can each have access to a disk device, keyboard input, and text console output.

1.1 Motivation

Virtualisation is useful for solving a wide variety of problems, and more uses seem to be constantly discovered. A virtualisation environment is often able to overcome many of the shortcomings of the operating systems being run in it. Applications for virtualisation include:

- reducing hardware under-utilisation by running many heterogeneous services on the same system, without the usual risk that they will interfere with each other's operation (server consolidation);
- safely executing code of unknown trustworthiness (sandboxing);
- providing multiple operating systems and the application software available for them on a single system;
- allowing tasks which require administrator access (e.g. testing kernel code), without allowing such access to the real machine;

- debugging and profiling — the virtual machine can be instrumented to provide information which is harder to extract from bare hardware.

Darwin is an interesting operating system for three main reasons:

1. It is one of the more popular desktop operating systems (mostly as part of Mac OS X), and can run a wide range of software, both commercial off-the-shelf and free and open source.

2. The Darwin source code is available and may be freely modified and distributed, which makes it convenient for research.

3. Darbat, the port of Darwin to L4, offers many research opportunities relating to the implementation of microkernel-based systems. What was a monolithic kernel has begun to be broken up into separate protected modules.

Thus, a virtualised environment supporting Darbat (and other operating systems) is of interest for several reasons both academic and practical.

1.2 Outline

Chapter 2 will provide background information on the concept of virtualisation, the Darbat operating system, and the components from which it is built.

Chapter 3 will present previous work that is related to the problem of running Darwin in a virtualised environment.

Chapter 4 will describe the design and implementation of the virtualised Darbat system that I have implemented.

Chapter 5 will present an evaluation of the work that has been done, in both qualitative and quantitative terms.

Chapter 6 will describe future developments that will be enabled by the work done in this thesis.

Finally, Chapter 7 will sum up what has been achieved in this thesis, and examine what conclusions can be drawn from the work that has been done.

Chapter 2

Background

In this chapter, I will first describe the idea of virtualisation and the techniques that are commonly used to achieve it. I will then describe the components from which the Darbat operating system is derived, including Darwin's kernel, the Mach kernel on which it is based, and the L4 microkernel. Lastly, I will describe the architecture of Darbat itself, as it was when I began this thesis.

2.1 Virtualisation

The idea of virtualisation is by no means new. IBM's CP 67, which is widely regarded as being the first true *virtual machine monitor* (VMM), became available in 1967 [Cre81]. Virtualisation was long considered to be a high-end feature, but in the late 1990s there was a resurgence of interest in the area, largely focused on virtualising commodity systems. This may have happened because commodity systems became powerful enough to replace mainframes in many instances.

A VMM provides an execution environment called a *virtual machine* (VM). The terms VMM and *hypervisor* are often used interchangeably. A VM allows software to be executed just as on real hardware. The advantage of virtualisation is that many separate VMs can be run on a single real computer, or even on a cluster of computers, with data transparently moved between physical machines by the VMM as needed [CH05]. An operating system running in a VM is called a *guest OS*, and correspondingly, the real machine on which the VM runs is called the *host*.

Virtualisation is related to simulation and emulation. Simulators aim to provide an exact software representation of a hardware platform. They are useful for development and debugging, as the state of the simulated machine can easily be accessed, and the software running on it cannot cause problems outside the simulated environment. Emulators are similar, but primarily aim to successfully execute software written for the machine being emulated, rather than recreating the precise behaviour of the hardware. Emulators tend to be used to run software written for platforms which are not readily available or are inconvenient to use.

To distinguish virtualisation from these related techniques, the definition given in 1974 by Popek and Goldberg is often used: “As a piece of software a VMM has three essential characteristics. First, the VMM provides an environment for programs which is essentially identical with the original machine; second, programs run in this environment show at worst only minor decreases in speed; and last, the VMM is in complete control of system resources.” [PG74]. They explain that their second requirement “demands that a statistically dominant subset of the virtual processor’s instructions be executed directly by the real processor, with no software intervention by the VMM.” Virtualisation thus provides many of the benefits of a simulator or emulator, but with a much smaller performance penalty.

In the literature, VMMs are often classified according to their characteristics. A *type I* (or *native*) VMM runs directly on the hardware, and performs resource management itself. A *type II* (or *hosted*) VMM runs on top of another operating system, and uses that operating system’s APIs to manage resources. The operating system on which a type II VMM runs is called the *host OS*. VMMs are also classified according to the way they handle guests performing privileged operations, as described in the following sections.

2.1.1 Classical Virtualisation

Code executing in a VM cannot be allowed unrestricted access to the real CPU state, as such access would allow it to gain full access to all the resources of the machine rather than the subset allocated to it by the VMM. Therefore, a subset of the instructions available to software running in the VM must not be executed on the real CPU, but must be handled by the VMM. These are called *sensitive instructions*.

A VMM must maintain the state of the real CPU such that it meets the expectations of the currently executing guest. It must have the expected access to memory regions, for example. To achieve this, the VMM loads a page table that gives the guest memory access at the appropriate locations. Writing to the page table is a sensitive operation, as the guest could expand its access. Thus, the VMM must be able to intercept page table writes and only allow them to occur if they abide by the memory allocation policy. Other CPU features must be similarly virtualised, including segmentation, interrupts, and privilege level changes.

Classical virtualisation, also known as trap-and-emulate virtualisation, relies on all sensitive CPU operations causing an exception when executed in user mode. That is, it requires that all sensitive instructions are also privileged. Guest operating systems are run in user mode, and the VMM catches exceptions caused by the guest attempting to perform privileged operations. It then performs an operation that has the same effect on the virtual machine as the attempted operation would have had on a real machine. The guest's execution can then be resumed at the instruction after the one causing the exception.

The popular x86 architecture has historically not been classically virtualisable [RI00]. Some instructions are legal in both supervisor mode and user mode, and thus do not normally raise exceptions, but behave differently depending on the mode they are executed in. This meant that to achieve virtualisation on x86, guest code had to be modified to avoid such unprivileged sensitive instructions. This could be achieved by manual source modification (see section 2.1.2), or by automatic code transformation at compile time (section 3.5) or even at runtime (section 3.2).

Both Intel and AMD, the major manufacturers of x86 processors, have recently introduced extensions to the x86 architecture which enable classical virtualisation [UNR+05,MS05]. However, using them exacts a significant performance penalty. This, along with the large number of CPUs that lack these extensions, means that other techniques are still attractive (see section 3.2). Further additions to the architecture in the future are planned to add hardware assistance for MMU and I/O virtualisation, which may increase the appeal of these features.

2.1.2 Para-virtualisation

Para-virtualisation [WSG02] involves presenting a virtual machine interface which is different to the real machine. Such systems cannot be considered true virtualisation, as they do not satisfy Popek and Goldberg's first requirement — hence the name. However, para-virtualisation offers the same isolation and resource utilisation advantages as true virtualisation, and can provide better performance.

The disadvantage of para-virtualisation is that guest operating systems must be ported to run on the virtual machine interface. Porting a non-trivial operating system to a new platform is complex, time-consuming and expensive. However, it has recently been shown that the porting process can be automated to a great extent (see section 3.5).

2.1.3 VMM Responsibilities

Figure 2.1 shows the relationship between the VMM and a guest operating system. The VMM runs in privileged mode, while the guest OS and its user-level programs run in unprivileged mode. Privileged operations are intercepted by the VMM. It then checks that the operation is safe, that is, it maintains the integrity and isolation of all VMs. If so, it is executed. If not, it may be modified to make it safe, or the VMM may indicate to the

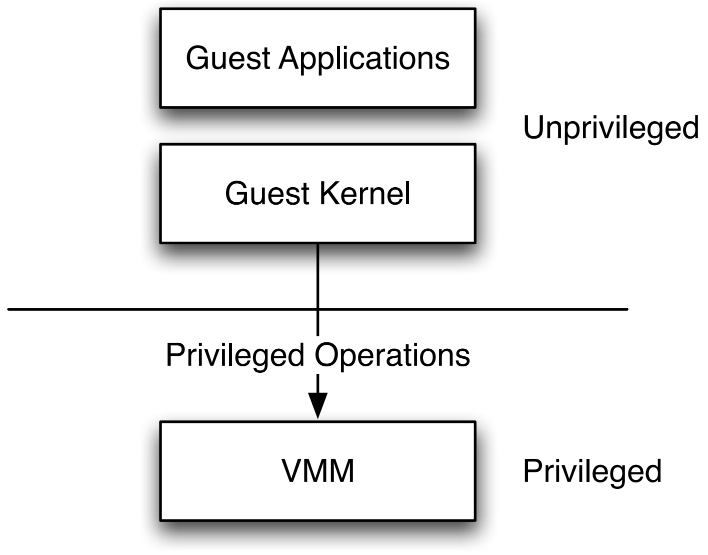

Figure 2.1: Relationship between a VMM and a guest operating system. The VMM runs in privileged mode, while the guest operating system and its applications run de-privileged.

guest that the operation failed.

The exact tasks which must be performed by a VMM vary depending on the architecture being virtualised, but on most contemporary processors they would include the following:

- maintaining guest virtual memory mappings, including page-fault handling;
- enabling controlled protection domain switching from guest user mode to guest kernel mode and back, including exception handling;
- providing guests with access to virtual I/O devices, which may be implemented in software or backed by real devices;
- delivering interrupts from devices (virtual or real) to guests;
- scheduling guest kernels and processes;
- managing, allocating and re-allocating machine resources between VMs.

2.2 Mach

Mach is a first-generation microkernel which was developed at Carnegie Mellon University in the 1980s [RJO+89]. It expanded on features found in the earlier Accent kernel, and aimed for clean abstractions and portability.

Microkernels are meant to enable improved reliability and flexibility by running many core operating system services in user mode that would traditionally have been part of the kernel. Separate services can be run in separate hardware-enforced protection domains, thus greatly reducing the likelihood that a malfunction in one service will affect the correct operation of other services. If a service crashes, it is possible to restart it, and with careful design such events can cause minimal disruption to the system's operation.

Mach provides a comprehensive API for *inter-process communication* (IPC). *Ports* are roughly analogous to UNIX's pipes, in that they are endpoints for communication. Like pipes, ports have associated access permissions, and a process can have many open ports at a time. Unlike pipes, ports are used to send structured messages rather than plain byte streams. Messages can be sent and received asynchronously, and the kernel checks all messages for validity.

Pagers are used in Mach to manage virtual memory. The default pager allocates anonymous memory regions. These regions are backed by RAM, or moved to disk when free RAM is low. Other pagers can provide memory-mapped files, or map devices for drivers.

The Mach project produced some good results, notably the virtual memory subsystem and the separation of the concepts of threads and processes. However, systems running on the Mach kernel tend to have significant performance problems. A large portion of IPC execution time is spent checking port access rights and verifying message validity. Even more serious on modern hardware is the fact that memory system performance is significantly degraded on Mach-based systems compared to monolithic ones.

Chen and Bershad [CB93] found that programs running on Mach with a user-level UNIX server exhibited a significantly higher memory cycle overhead per instruction than when running on a monolithic UNIX. Liedtke [Lie95] performed further analysis of Chen and Bershad's results and showed that the degradation was due almost entirely to the exhaustion of cache capacity, and that the component responsible for this was Mach itself. Mach's large working set limits the performance of system's built on it.

Some system implementers reacted to Mach's poor performance by moving large parts of the operating system back inside the kernel. Unfortunately, this approach simply resulted in a monolithic kernel with Mach's APIs added, and lost the advantages of the microkernel design. Ironically, it appears that this approach only closed a fraction of the performance gap [HHL+97].

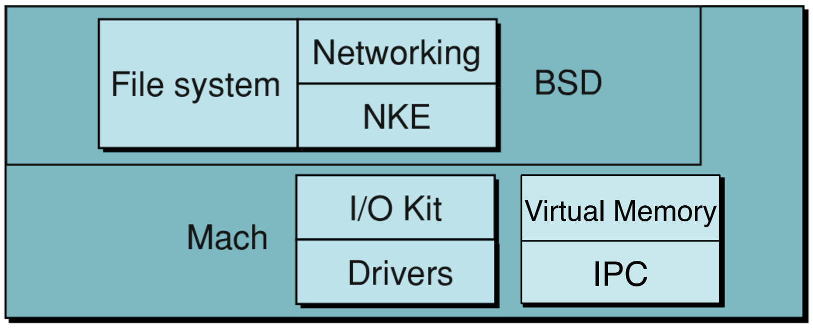

Figure 2.2: Components of the XNU kernel. Reproduced from [App06c].

2.3 Darwin

Darwin [App] is an open-source operating system which is probably best known for its role as the basis of Apple's Mac OS X [App06d]. It consists of mainly BSD user-level programs and a kernel called *XNU*, which was created by combining the Open Software Foundation's version of Mach 3 with most of a 4.3 BSD kernel. Darwin was derived from the earlier OPENSTEP operating system developed by NeXT Software [App06a]. Later development added code from more recent BSD versions, e.g. FreeBSD 5.

Darwin runs on PowerPC and x86 hardware. It should theoretically work on a wide range of systems, but it is rarely used on hardware not made by Apple. All Apple computers currently in production are x86-based.

Figure 2.2 shows the structure of the XNU kernel. All of the kernel components run in a single protection domain with full privilege. NKE stands for network kernel extension, the mechanism by which network stack components can be added to the kernel.

2.3.1 Mach

The Mach portion of the kernel provides threads, virtual memory and paging, timers, locks, scheduling, and of course IPC. The default pager uses swap files accessed through the BSD VFS interface to store less-used pages when physical memory is scarce. There is also a vnode pager for memory-mapped files and a device pager for memory-mapped devices.

2.3.2 BSD

The BSD component of XNU provides standard UNIX services such as the file system, networking, process management, signals, pipes, TTYs, pseudo-devices, and POSIX and System V shared memory and IPC. It implements these using the services provided by Mach and the I/O Kit (XNU's device driver component, described in section 2.3.3).

A *unified buffer cache* is used for both file data and virtual memory pages. This allows efficient page replacement through the file system, since the virtual memory object descriptors are also used for in-memory file data. It also allows cached data to easily be shared between the read/write and mmap interfaces.

The back end of the BSD device abstraction layers use the interfaces exported by the I/O Kit. For example, the block device code would access a hard disk drive through an IOMedia object. The I/O Kit interface is defined in terms of Mach IPC, but is converted to function calls at build time since the components are in the same protection domain.

2.3.3 I/O Kit

Darwin uses an object-oriented driver framework called the I/O Kit [App06b]. It is implemented in a subset of C++: Exceptions, multiple inheritance and templates are not supported. The standard C++ runtime type information system is also unavailable, but libkern supplies its own version.

Families are sets of classes that implement functionality that is common to categories of similar devices. Driver implementations inherit from a parent class from an appropriate family. For example, a USB keyboard driver inherits from the IOHIDevice class in the human interface device family.

Drivers are linked together using *nubs*, objects which represent communication channels or logical services. Drivers usually need the services of other drivers in order to do their job. For example, a PCI ethernet card driver needs the services of the PCI controller driver. The latter publishes a nub which allows the former to use its services. Figure 2.3 illustrates the relationship between drivers, families and nubs using the example of a SCSI disk connected to a SCSI controller card.

The controller card is connected to a PCI bus. The PCI bus driver inherits from the IOPCIBridge family, and publishes an IOPCIDevice nub for each device it finds connected to the bus. The SCSI card driver is given an IOPCIDevice reference for the card it drives. It publishes an IOSCSIParallelDevice nub for each device it finds on the SCSI bus. Finally, the SCSI disk driver, which inherits from the IOBlockStorageDriver family, is given an IOSCSIParallelDevice reference for the disk it drives, and publishes an IOMedia nub which presents a generic disk device interface.

2.3.4 Libkern

The libkern component provides simplified C++ runtime services for the I/O Kit and drivers, including an implementation of runtime type information. It also provides utility functions that are useful in many parts of the

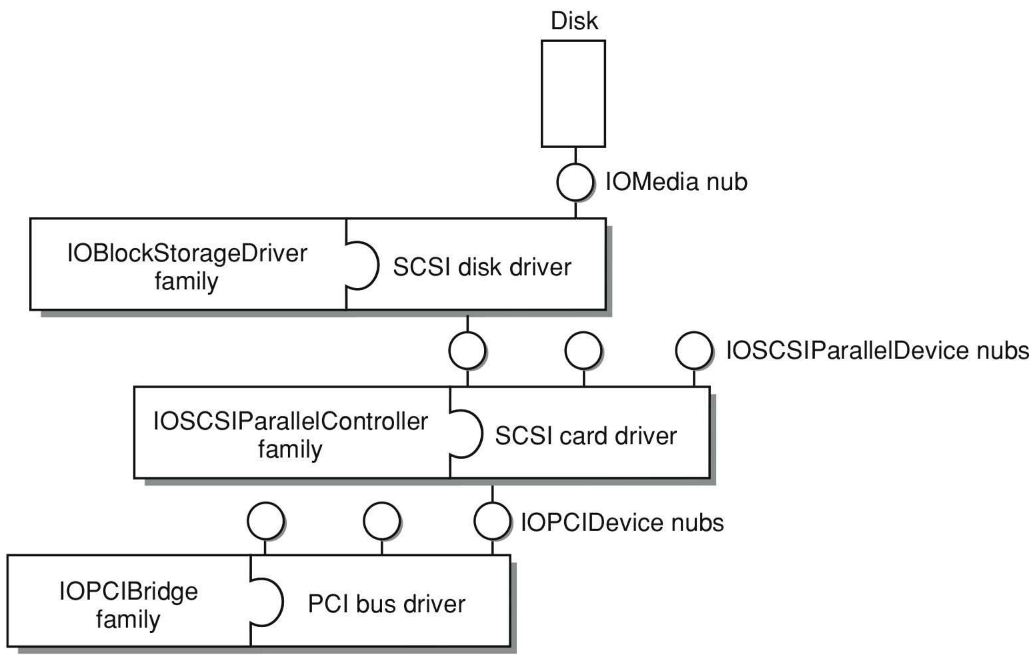

Figure 2.3: I/O Kit drivers inheriting from families and linked by nubs. Reproduced from [App06c].

kernel, e.g. byte-swapping.

2.3.5 Platform Expert

The Platform Expert is the module of XNU that is responsible for taking care of platform-specific details in the boot sequence. This includes handling the kernel boot arguments, registering interrupts, scanning the system for available devices, and driving devices such as the video console that are required from an early stage.

2.4 L4

L4 [Lie95] is a family of second-generation microkernels. It avoids the performance problems seen in earlier systems such as Mach by an approach of true minimality. The kernel contains only those features which cannot be implemented at user level while ensuring the integrity of the system. Darbat currently runs on version N1 of the NICTA::Pistachio [NIC05] implementation of L4.

The L4 API deals with two main abstractions, threads and address spaces. Threads are an abstraction of sequential program execution. Each thread has a unique identifier by which it may be named. Address spaces are an abstraction of virtual memory and protection. An address space can be named by specifying the ID of a thread that executes in it. The combination of an address space and the threads in it is called a *task*.

2.4.1 IPC

Threads can communicate via the IPC system call. In some L4 implementations, this may include a special form of IPC which maps part of the sender's address space into the receiver's. Threads can use shared memory regions for large data transfers, or indeed in whatever way the application programmer wishes; while ordinary IPC is useful for small messages and for synchronisation.

L4 IPC is synchronous, i.e. the sending thread blocks until the recipient explicitly performs a receive operation, and vice versa (with optional timeouts in some implementations). This avoids the need to manage message queues in the kernel[1]. L4 also associates minimal semantics with message data and uses very simple security mechanisms. L4's IPC is consequently at least an order of magnitude faster than Mach's [Lie96].

IPC messages are sent using up to 64 *message registers* (MRs). The exact number of available MRs is implementation-dependent. Each register is the size of a machine word. MRs have read-once semantics: reading from an MR causes the value contained in it to become undefined. Writing to the MR makes its value well-defined again.

The zeroth message register must always be used, and contains the *message tag*. The tag contains metadata such as the number of words being transferred, and also specifies parameters such as whether the operation should be a send, a receive, or both.

The sender places the values they wish to send into their MRs and invokes the IPC operation. The values are then copied by the kernel into the recipients MRs. Some MRs may be implemented using real machine registers, and in this case no actual copying needs to be done. The other MRs are implemented as memory areas within a larger structure called the *user-accessible thread control block* (UTCB).

2.4.2 Privilege management

In the N1 kernel, privileges allowing threads to alter other threads' scheduling parameters, give access to memory, and restrict communication are delegated hierarchically. (In some other implementations, the first task to be started, which is known as the *root task*, has exclusive control over these operations.) The root task has access to all memory and may create new

[1]The one exception is the asynchronous notification mechanism, which nevertheless requires no queuing or semantic checks by the kernel.

threads and address spaces. It gives access to memory to other tasks by mapping. New threads are assigned threads to act as their pager, scheduler, exception handler, and *redirectors* when they are created.

Threads may set the priority of any thread that they are assigned to as scheduler. However, they may only set a maximum priority that is equal to their own. Similarly, a pager handles page faults for a thread by mapping some of its own memory into the faulting thread's address space, but can only map memory that it has access to itself.

In the N1 kernel, threads' communication may be restricted by the use of redirectors. Different redirectors can be used for sending and receiving. If a redirector has been specified for an IPC's direction, then the message is sent to the redirector rather than its destination. The redirector can then check that the message is allowable under the system's policies, and if so, forward it to its destination. Redirectors are allowed to forge the sender field of messages that they forward so that they appear to come from the original sender rather than the redirector.

2.5 Iguana

Iguana [HL04,NIC06b] is a set of services which run on top of L4. It provides a set of basic policies which are implemented using the mechanisms provided by L4. It was designed as a basic framework on which embedded systems can be built, and provides services for memory allocation, protection domain management, and thread management, and a capability-based framework for access rights management. Protection domains are implemented as (mostly) non-overlapping address spaces, which allows for a single-address-space view of the system while retaining hardware-enforced memory protection.

One server thread provides most of Iguana's functionality and runs as the root task that is started by L4 at system boot time. Other functions of a more stand-alone nature run in their own threads. The latter category includes the `init` thread that launches most of the system's tasks, a timer, and a naming service that allows threads to create and look up mappings between strings and numbers (which could be pointers, thread IDs, etc.).

Iguana keeps track of allocated memory using objects called *memsections*. References to memsections are opaque, but threads can look up the base address of a memsection's allocation with a call to the Iguana server. A capability that allows access to a memsection contains a reference to it.

2.6 Magpie

Magpie [NIC06a] is an interface compiler. Given an interface description in a subset of the CORBA [Obj04] *interface description language* (IDL), it produces stub code that implements *remote procedure calls* (RPC) using

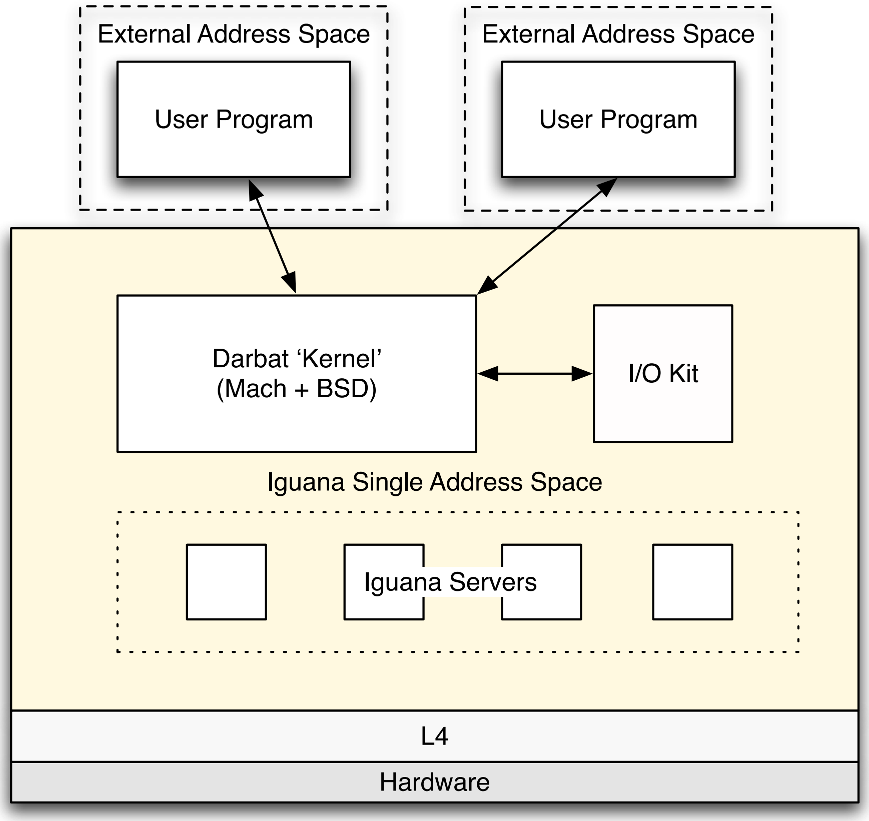

Figure 2.4: The structure of the Darbat system prior to this thesis.

L4's IPC mechanism. This means that a service implemented in a different thread can be called as though it were an ordinary function. Magpie is used extensively in Iguana to provide the interfaces to the server threads.

Using an IDL compiler greatly reduces the scope for programming errors involving the way data is put into and taken out of L4's message registers. The compiler can also take advantage of architecture-specific optimisations, though the version of Magpie being used in the Darbat project only does this on the ARM architecture.

2.7 Darbat

Darbat is a modified version of Darwin which runs entirely in user mode on top of L4. Initial work was done by Wong [Won03], but the approach of removing the Mach code entirely proved problematic. Mach's IPC and virtual memory APIs are part of Darwin's official user API, and practically every feature offered by them is used by some piece of existing software or another. Achieving full compatibility with existing Darwin software would therefore have amounted to re-implementing those Mach subsystems. While this was partially achieved by Hohmuth and Rudolph when porting Lites to L3 [HR96], they only had to implement the small subset of Mach IPC that was needed by their system.

Although the I/O Kit is co-located with the Mach and BSD components in the XNU kernel, it has been successfully separated from the rest of the kernel and ported to run on L4 [Lee05]. Darbat now makes use of this separate I/O Kit server rather than keeping XNU intact. This is done because running separate services in separate protection domains improves reliability and security, as per the principle of least privilege. It was also anticipated that this design would make virtualisation easier by not accessing real hardware from the parts of the system that would be running in a virtual machine.

The current, redesigned version of Darbat, which leaves Mach mostly intact, was developed by Gray *et al.* [GLB06]. While user programs can still use Mach IPC, Darbat takes advantage of the performance of L4's IPC by modifying the standard user libraries (e.g. libSystem) to use it. System calls are implemented as L4 IPC calls from the user program to the Darbat server.

Darbat currently uses Iguana for basic memory and address space management, as well as its resource naming service. Figure 2.4 shows the structure of the Darbat system, as it was when work on this thesis was started.

Darbat currently runs only on x86 hardware, but it may be ported to PowerPC at some point. It can be booted all the way up to a single-user shell, and while full binary compatibility with existing x86 Darwin applications has not quite been achieved due to a pending L4 ABI change and some unimplemented system calls, many nontrivial programs run very well.

Chapter 3

Related Work

3.1 IBM's VM

IBM developed the VM system in order to solve the problem of underutilised hardware. At various points in its development history it has been known as CP-67, CP/CMS, VM/CMS, VM/370, VM/ESA, and z/VM. It is a type I VMM, and unsurprisingly, it is a classical virtualisation system. It runs on various IBM mainframe models. It was first made available outside IBM in 1967, and development and use has continued to the present day.

Later IBM hardware added an execution mode called *interpretive execution* [OJG91], which allows many privileged instructions executed by guests to be handled by the hardware rather than trapping into the VMM. The VMM supplies information about the guest's privileged state to the hardware before starting the guest in interpretive mode. This greatly reduces the overhead of running the guest OS in a VM. For example, a guest's attempt to disable interrupts will be translated by the hardware into the setting of an appropriate flag in the data structure representing the virtual CPU's privileged state.

3.2 VMware

VMware [VMw07] is a series of commercial virtualisation packages produced by VMware, Inc. There are both type I VMMs (e.g. VMware ESX Server), and type II VMMs (e.g. VMware Workstation) that run on Linux and Microsoft Windows on x86 hardware. Since the x86 architecture has historically not been classically virtualisable, VMware uses binary translation of the guest kernel to enable full virtualisation. It changes sensitive instruction sequences to code that modifies the state of the virtual CPU in the appropriate manner. The translation is performed on demand at runtime.

Memory is virtualised by maintaining a *shadow page table* for each guest. The shadow page table is what is actually used by the MMU to perform address translation. Sensitive MMU operations in the guest are changed to operations that modify the shadow page table (with appropriate validity checking) as well as modifying the guest's own page table.

Each guest is given a fixed maximum memory allocation when it is started. If memory is underutilised, each guest will have access to its full allocation. When memory becomes tight, a *balloon driver* is used to force guests to conserve RAM [Wal02]. This driver uses the guest kernel's memory allocation routines to claim memory for itself. It tells the VMM which pages it has obtained, and the VMM then uses those pages for other purposes. The guest kernel should respond to the low-memory situation in its customary way, e.g. by paging out to disk.

VMware could be viewed as both a para-virtualising VMM and a tool for porting operating systems at runtime. Alternatively, a black-box view of the system might lead to the conclusion that VMware is a VMM that uses clever tricks to fully virtualise x86. It runs unmodified x86 code at nearly full speed, and the fact that sensitive instructions are replaced rather than causing traps is merely an implementation detail. The distinction may be academic, but it is interesting to note that Popek and Goldberg's first requirement may be interpreted in different ways.

VMware engineers recently evaluated the hardware support for virtualisation offered by recent Intel x86 CPUs [AA06]. However, they found that an implementation using the hardware features performed considerably worse than their current software approach. They attribute this to the fact that MMU virtualisation is not performed in hardware, and many relatively expensive traps into the VMM cannot be avoided as they occur on every potentially sensitive instruction. The binary translation approach can often avoid entering the hypervisor altogether by appropriate instruction rewriting.

3.3 Xen

Xen [BDF$^+$03] is a type I VMM which presents a para-virtualised interface. Certain privileged operations must be converted to *hypercalls*, that is, hypervisor API calls, in order for guest kernels to operate correctly. The hypervisor is mapped into the top of every guest address space to reduce the number of context switches required. Guests are responsible for managing their own page tables, and Xen only verifies that updates to page tables are allowable.

One guest is given more privileges than the others, and is known as *Domain0*. It is given direct access to physical devices, and is responsible for loading new guest instances, maintaining translations for virtual block devices, managing the firewall rules in the virtual network device, and hosting the software that presents the hypervisor control interface to the system administrator.

Disk and network I/O is performed asynchronously. Guests provide buffers, and place read and write requests in a circular queue whose location is known to the hypervisor. Guests are notified of events, such as the arrival of a network packet, by an *asynchronous event mechanism* that uses callback functions. Virtual block devices are backed in Domain0 either by real block devices or by disk images.

Guests other than Domain0 can be given memory-mapped access to particular physical PCI devices. A virtual PCI configuration space is provided, and interrupts are delivered using the asynchronous event mechanism.

3.4 QEMU

QEMU [Bel05] began as a CPU emulator, but later became capable of type II virtualisation on x86 hardware through the addition of a host kernel module called `kqemu`. User mode guest code is executed as-is, with syscalls intercepted by the kernel module. Guest kernel code is normally emulated, but experimental support for virtualising it is available.

The source code for `kqemu` was not available at first, but it was eventually released in early February 2007. Unfortunately, there was insufficient time to examine its implementation before the completion of this thesis.

3.5 Pre-Virtualisation

Pre-virtualisation [LUC$^+$05] is a para-virtualisation technique whereby sensitive instructions are handled in a kernel at compile time. The sensitive instructions can either be replaced with appropriate emulation code immediately, or padded with nops to allow the modification to be made at runtime. A library known as the *in-place VMM* is loaded along with the guest kernel. It translates sensitive operations to the API of whatever VMM the guest OS is running under, and is called by the previously inserted code when appropriate. The same kernel binary can also be run on bare hardware without an in-place VMM.

Locating the sensitive instructions is done in two main ways. Instructions which are illegal in user mode can be identified automatically by a modified assembler. The remaining instances are found by running the guest in a virtual machine, protecting all sensitive memory objects, executing a typical workload, and recording where protection faults occur.

The advantages of pre-virtualisation are greatly reduced development effort compared to usual para-virtualisation methods, and the ability to produce a guest binary which can run on bare hardware and on a variety of VMMs.

3.6 User-mode Linux

User-mode Linux (UML) [Dik00] is a para-virtualised version of the Linux kernel which is run on Linux as an ordinary user-mode program. It uses the `ptrace` mechanism to intercept the system calls made by programs running under the guest kernel, and installs its own handlers for all signals that they may receive. In this way it is able to redirect all data going to and from the kernel, allowing it to maintain the virtual machine state and isolate guest processes from the host.

3.7 Linux-on-Linux

Linux-on-Linux [Lin06] is, like User-mode Linux, a system for running Linux kernels in user mode on a host Linux. Unlike UML, it requires fairly minimal changes to the Linux source code. Some extra `ptrace` functionality is added to the host, as well as a modified scheduler which "allows each virtual machine to be assigned a fixed proportion of the processor it runs on." The guest kernel is patched to use a memory layout suitable for a Linux user process, and scatter-gather I/O is enabled for the `simscsi` virtual disk driver.

The guest kernel is launched by a user-mode VMM. Pre-virtualisation is used to replace sensitive instructions in the guest kernel with calls to the VMM, or optionally non-privileged sensitive instructions can be replaced with instructions that will trap (and be caught by the VMM). Linux-on-Linux currently runs on Itanium processors.

3.8 Mac-on-Linux

Mac-on-Linux [Mac07] is a virtualisation system which allows Mac OS or Mac OS X to be run as a guest on top of Linux. It only runs on PowerPC hardware, and is thus able to operate as a classical trap-and-emulate VMM. A host kernel module handles the execution of guest privileged operations.

The guest OS is launched by a user-level wrapper component, which arranges a virtual machine context to be set up by the kernel. Virtual hardware devices for the guest are implemented mostly in the user-mode component, and are backed using the host OS's facilities.

3.9 Dedicated Device Driver VMs

LeVasseur *et al.* developed a system for reusing unmodified device drivers in new operating systems by running them in virtual machines [LUSG04]. Each driver is run in its original OS, providing exact compatibility. This *device driver OS* (DD/OS) is para-virtualised so that it only attempts to use the machine resources allocated to it for the purpose of running the driver. A *translation module* is added to the DD/OS, which enables communication with the host OS and other VMs, allowing them to access the functionality of the driver. A driver in a DD/OS can use functionality provided by a driver in another DD/OS using this mechanism.

The translation module can access the DD/OS at whatever layer is most convenient. This allows one translation module to be used for a variety of similar devices. The interface exported by the translation module can also be whatever is chosen by its implementer.

The advantage of this approach is that existing drivers can be used without modification, and without having to emulate the interface that they expect. Devices are accessed by the rest of the system at a higher level of abstraction through the translation modules. Reliability is also improved by keeping drivers in separate protection domains.

3.10 Wombat

Wombat [LvSH05] is a port of Linux to the L4 microkernel. It was developed at National ICT Australia (NICTA). It was partly inspired by the L^4Linux work done at TU Dresden [HHL^+97]. L^4Linux is relatively difficult to port to a new CPU architecture, and ran only on x86 for many years until an ARM port was added in 2005. Wombat is designed to be portable, and currently runs on x86, ARM and MIPS. A new set of architecture-specific files for running on L4 was added to Linux, essentially treating L4 like a new hardware platform.

Wombat is designed for use in embedded systems. Embedded devices that run Linux often also need to run real-time and/or security-critical software. Running Linux on L4 allows these parts of the system to be protected, in terms of both performance and data access, from faulty or malicious code running in the Linux environment.

Wombat is built on Iguana, with the Wombat server running in the shared address space along with the Iguana servers and any native Iguana applications. Native Linux device drivers can be used provided they do not perform DMA, but it is preferred that Iguana drivers be used, as the devices can then be shared with other Iguana tasks. User-level device driver support has been added to ordinary Linux by members of the Gelato Federation, and this feature can in fact use the same drivers as Iguana [Chu04].

Unmodified Linux user programs can be run in their own address space, and communicate with Wombat via the *trampoline* mechanism. L4 uses different syscall numbers to Linux, so when a user process performs a Linux syscall, L4 treats it as an exception. The Wombat server is set up to be the exception handler for its user processes, and L4 delivers the exception to it via IPC, at which point Wombat can handle the syscall. Linux applications can also be ported to run in the Iguana environment and call the Wombat server directly using L4 IPC.

Page faults are propagated to the Wombat server by the same mechanism as other exceptions, and are handled by invoking the appropriate L4 mapping operation.

Scheduling is accomplished by ensuring that only one Linux thread per processor is runnable at any time. This means that the L4 scheduler will abide by the scheduling decisions made by Linux, as far as Linux processes are concerned. Iguana tasks that have a higher priority than Wombat will always be scheduled in favour of any Linux thread. This allows real-time tasks to coexist with Linux without compromising their correctness.

Chapter 4

The Virtualised Darbat System

This chapter describes the changes that I have made to the Darbat system. I have continued using L4 as a hypervisor, and have made it possible to run several Darbat kernels simultaneously. I have also implemented a system for enabling these kernels to share the hardware devices that are available.

4.1 Design Rationale

Much of the work of para-virtualising Darbat had already been done by virtue of it being ported to L4. Pre-virtualisation was not applicable as the guest kernel had already been purged of sensitive instructions.

Xen's structure shows remarkable similarities to a microkernel-based system, but with complexities and difficulties that can be avoided with L4 [HUL06]. Domain0 is a single point of failure for most other software in the system, and having most device functionality implemented in it also leads to large communication overheads. Having dedicated device servers with drivers in separate protection domains mitigates the reliability issues, and L4's IPC has been highly tuned for performance.

A type II VMM such as VMware Workstation would of course have met many of the goals, but it would be very difficult to achieve the ones related to device sharing. Performance is also limited by the need for guests to go through the host OS to get to the hardware. Virtual devices would either need to emulate the interface of real devices, with associated performance problems due to the multiple translations between interfaces, or they would require drivers to be written for each guest OS.

It would also be very difficult to allow control of a device to be passed from one VM to another in cases where only one can reasonably use it at a time. An example would be the video card. The host OS would generally not have provisions for dynamically giving up and reasserting control of such devices.

In light of these factors, I decided to continue and extend the para-virtualisation approach that had already begun to be applied to Darbat. The work to be done consisted of finding and removing assumptions that system resources could be exclusively controlled by a single guest kernel, and designing and implementing ways to share those resources between multiple guests. It would also be necessary to modify the system bootstrap process to allow multiple guest kernels to be started.

4.2 System Overview

I extended the para-virtualised Darbat system using multiple user-level server tasks running in the Iguana environment. The environment now consists of a supervisor task and a set of virtual device servers, as well as the guest Darbat servers themselves and the previously existing components such as the I/O Kit. The number of Darbat guest kernels that can be run simultaneously is limited only by available address space. The virtual devices are backed by real devices via I/O Kit, or by programs running in guests. The latter allows guests to use virtual disks backed by disk images.

One guest is roughly analogous to Xen's Domain0 in that it is trusted to run a supervisor UI program and will, in practice, tend to back most of the virtual disk devices. However, it is not required to back any of the virtual devices, and does not contain drivers. I will refer to this guest as *Darbat0*.

It is not actually necessary for the supervisor UI to be presented through a guest OS, nor that that guest OS should necessarily be Darbat. Another OS or a purpose-built program could be used instead. It was simply convenient to use the first Darbat to be started for this purpose.

Figure 4.1 shows the structure of the system. Arrows indicate communication. Communication with the Iguana servers and L4 is omitted for clarity. The first major difference is that there can now be multiple Darbat kernels running as servers in the Iguana environment. Each one has its own set of user processes, which it manages as before. The second major difference is that the Darbat servers do not communicate directly with the I/O Kit. Instead they must communicate with the virtual device servers, which handle the multiplexing of real devices in various ways. The *vblock daemon*, shown in the upper right, is one of these ways. It is a user application that runs under one of the Darbat kernels, but also communicates directly with one of the virtual device servers. It uses the resources of its parent Darbat kernel to provide virtual block devices to other Darbat kernels.

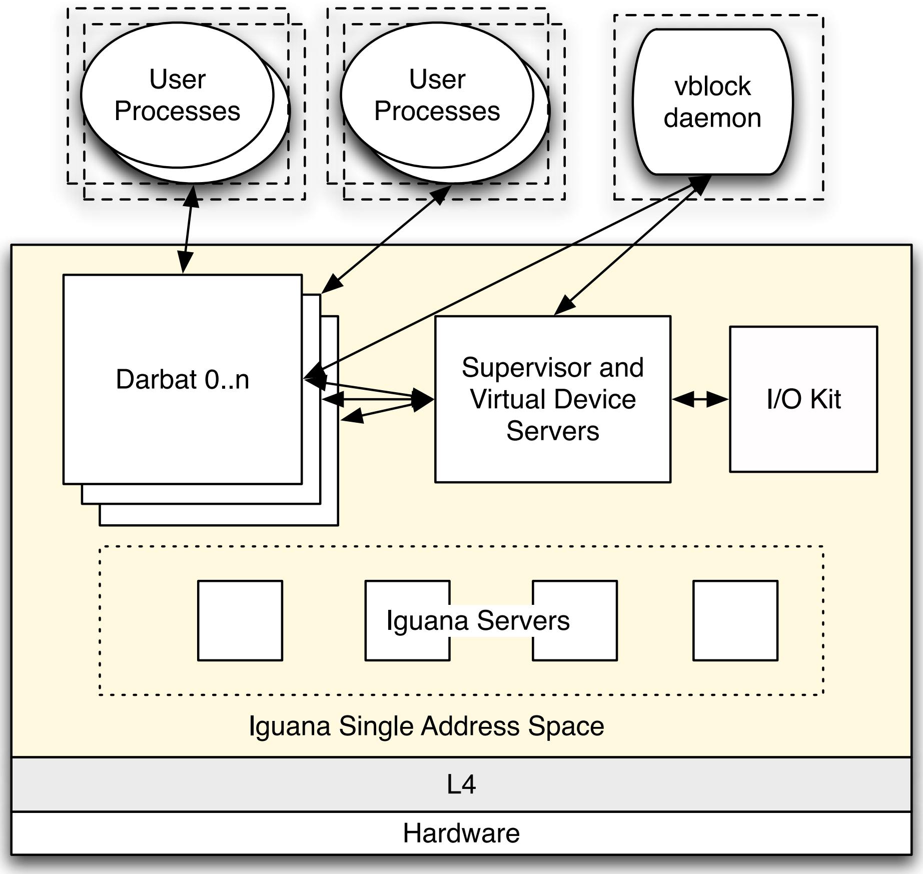

Figure 4.1: Structure of the virtualised Darbat system. Dashed lines indicate external address spaces.

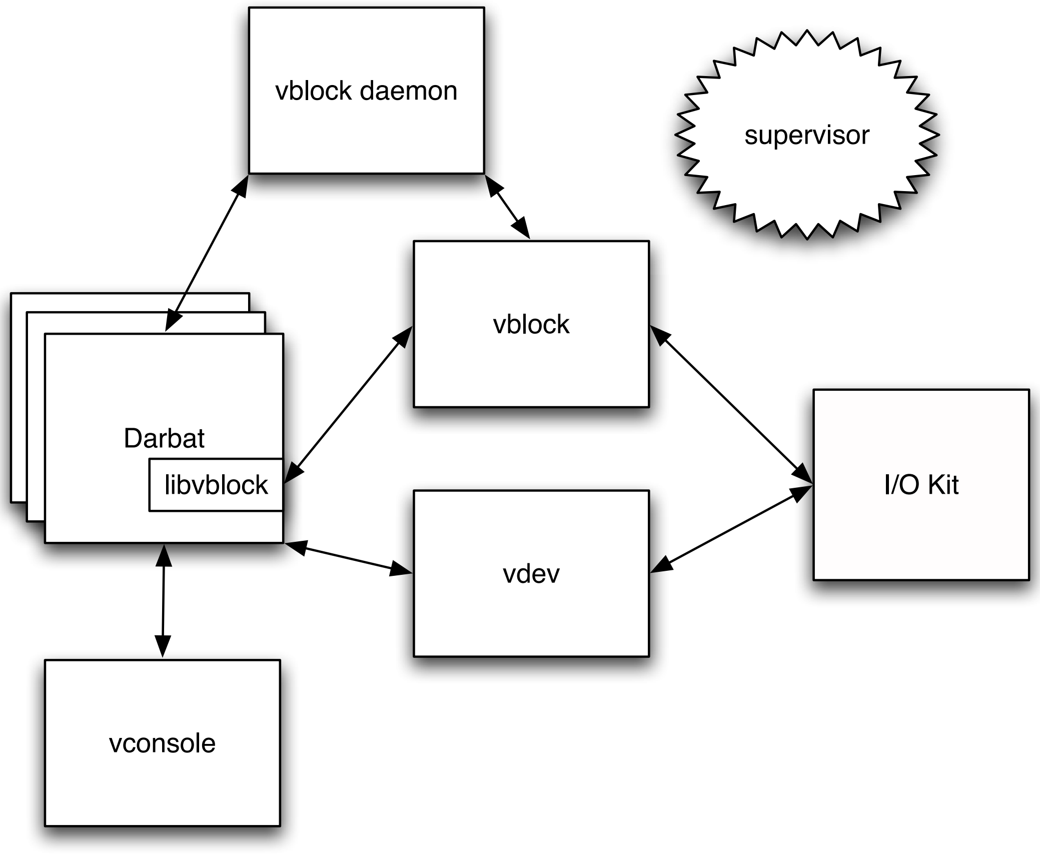

Figure 4.2: The virtualisation servers and their relationship to Darbat, I/O Kit and each other.

4.3 Components

Figure 4.2 shows the communication that takes place between the virtualisation servers, the Darbat kernels, and the I/O Kit. *Vconsole* takes text output from all of the Darbat guests and displays it on the video framebuffer. *Libvblock* is a library that is linked in to the Darbat kernels, and handles communication with the *vblock* server. The vblock server handles the routing of messages between users and providers of block devices. The provider is currently either I/O Kit, for real disks, or the vblock daemon, for virtual disks backed by a file on some other block device. The *vdev* server proxies I/O Kit API calls, and handles the multiplexing of keyboard input. The *supervisor* is responsible for the overall management of all the components in the system.

4.3.1 Supervisor

The supervisor is responsible for launching guests. It also co-ordinates changes to device configurations. It launches Darbat0 when it is started, and tells the vconsole, vblock and vdev servers about the new Darbat instance. This causes them to display its console output on the screen, note that it has access to I/O Kit's disk device, and start sending it characters typed on the keyboard, respectively. The supervisor then waits for I/O Kit to register its disk devices with vblock, and only then allows Darbat0 to continue and begin using the disk.

The supervisor calls on Iguana's `init` task to do the actual launching of new Darbat instances. I decided to implement it this way because `init` already had the code to launch servers in the Iguana environment, and in fact had ready access to necessary information that would have been more difficult to obtain in a different task, such as the reference to the naming server that needs to be placed on the stack of each new task.

After the initial system setup is done, the supervisor will currently act upon three types of message. First, it receives notification from I/O Kit of certain characters being entered on the keyboard, which will cause it to send messages to vdev and vconsole telling them to switch the ownership of the keyboard and console. This means that keystrokes will go to a different guest, and that same guest's console output will be displayed on the screen.

Second, a message can cause the supervisor to launch a new guest. This is called by the `guest_launcher` program running under Darbat0. A reply to such messages is sent with the thread ID of the new guest. The third type of message is generally received directly after replying to a launch guest message. It is a request to send a message to a guest that lets it know it can begin using its disk devices. This cannot be sent by the supervisor automatically, because the `guest_launcher` needs to call vblock to give the guest access to its disk first, for which it needs the guest's thread ID.

4.3.2 The vconsole server

The vconsole server accepts character output from guests. A shared, page-sized memsection is used as a ring buffer that holds characters to be transferred to vconsole. If the video console is active, guest output is displayed on it. A message from the supervisor can activate or deactivate the video console.

One guest is said to 'own' the screen at any given time. This guest's output is displayed on screen as it is received, while that of other guests is only stored in FIFO queues. When a queue reaches a maximum length, the oldest items are deleted from it until it is under the maximum length again. Upon receiving an appropriate message from the supervisor, the ownership of the screen is changed. The screen is cleared, and the contents of the new owner's queue are then displayed.

4.3.3 The vdev server

This server acts as a proxy between guests and the I/O Kit. It currently simply passes I/O Kit RPC calls straight through without modification, but it is expected that vdev will become much more useful when multiple I/O Kits are present in the system (see section 6.5). At present, it is really only adding latency for I/O Kit API calls.

The vdev server does have one important function. It keeps track of the current 'owner' of the keyboard and forwards keystrokes to the appropriate Darbat kernel. A message from the supervisor causes the owner to be changed. By switching the video console device to a different guest along with the keyboard, the entire user interface can be moved among guests in the same way that a KVM switch does so with real machines.

4.3.4 The vblock server

This server provides an interface for block devices such as hard disks. Virtual block devices may be backed by a real device or by a backing store managed by a daemon running on a guest OS. Such a daemon would typically use a partition or disk image on one of its host kernel's available virtual block devices to back the devices it exports.

Guests may query the vblock server to see what devices are available to them, and may read from and write to their devices. Reading or writing is accomplished by calling the server with a buffer reference, a block number, and a device identifier. Buffers must be supplied by guests. All operations are asynchronous, so read/write calls to providers are expected to return as soon as the request has been enqueued. A separate thread in the server handles upcalls [Cla85], which signal the completion of I/O operations to guests.

Guests may also query the geometry of their devices, request that all outstanding data for a device be flushed to disk, and send a capability to a device's provider. A capability must be sent for all buffers that are to be used with a read or write call, so that the provider can access the buffer to read or write the data from it.

Providers may add and remove devices, and give a guest access to a device. The supervisor may add and remove clients. The vblock server keeps track of the available devices, and the access that the registered clients have to them, in a set of hash tables.

Currently, due to complications in the implementation of the vblock daemon (see section 4.3.6), devices hosted by guest processes are limited to having only one disk operation outstanding at a time. If the client performs another call while a disk operation is outstanding, vblock will return an error code of EAGAIN, and the client is expected to retry the call after some period of time. Whether each given provider has an outstanding operation is kept track of in a hash table. A value is inserted upon successfully enqueuing an operation, and is removed by the upcall thread upon delivering the corresponding completion message.

The vblock server's IDL4 interface definition is reproduced in appendix A.2. It is compiled into RPC stub code by the Magpie interface generator.

4.3.5 Libvblock

Libvblock is a library that is linked into guest kernels. It manages a pool of I/O buffers and handles communication with the vblock server.

Zero-copy is used as much as possible when doing I/O with vblock devices. A pointer to the buffer being used is passed with each request. This requires that a capability for the buffer memory has been given to the provider beforehand. Libvblock takes care of these chores.

The client simply needs to call a function to get a suitable buffer, then pass the buffer reference to the read or write call, and call another function to return the buffer when the corresponding completion message is received. Libvblock ensures that the capability is first sent if it has not already been.

The kernel using libvblock is expected to have a function that it calls when it receives a message that signals the completion of an asynchronous disk I/O operation. A pointer to this completion function is passed in to libvblock's initialisation function, since libvblock may receive completion messages during its normal operation. In particular, it must receive them to implement synchronous I/O, since the device providers always behave asynchronously.

Libvblock's interface as defined in its public header file is reproduced in appendix A.1.

4.3.6 The userland vblock daemon

The vblock daemon is a user program that runs under the control of a Darbat guest OS. It backs virtual block devices using files in the filesystem provided by its OS. It follows the same protocol as I/O Kit when communicating with the vblock server.

Darbat does not currently preempt its user threads. It simply assumes that they will eventually make a system call, at which point they will be blocked waiting for a reply, so any thread can be scheduled by replying to it. Unfortunately, this means that a user process that blocks in an L4 IPC call that does not involve the Darbat server effectively locks up its parent Darbat instance. For this reason, the vblock daemon has to use a polling loop that tries a non-blocking L4 receive call, and does a short usleep() if no message is received.

The code is structured such that it will be very easy to convert it to be dual-threaded when Darbat gains preemption abilities. But until then, it is single-threaded. This means that the same thread that receives requests from the vblock server can be blocked waiting for the Darbat server, which can itself be blocked waiting for the vblock server. This circular wait condition must be avoided by having the vblock server refuse to send a message to the daemon if there is an I/O already outstanding.

4.4 Memory Management

Each guest is run as an Iguana task, which means that memory allocated to them is part of the shared address space. This will likely change in the future; see chapter 6. One large Iguana memsection is used as the guest's 'physical memory', and allocations are generally done using the guest's own mechanism, e.g. `kalloc()`. Notable exceptions to this are when the memory is shared with another task — disk I/O buffers are allocated as 1:1 mapped memsections so that they can be used in DMA operations by I/O Kit, for example.

An important memory management change that had to be made in the Darbat kernel itself was a new system call that I called `mapcap`. This call takes a capability that gives access to an Iguana memsection, and maps the corresponding memory into the address space of the calling process. It is necessary to do this so that the vblock daemon can access I/O buffers given to it by the Darbat kernels that use the virtual devices it provides (see Section 4.3.6).

The `mapcap` call works by looking up the address and size of the memsection, and making a note of the fact that the calling process has a mapping for each of the pages involved. An equal-sized region of the process's address space is reserved in the Mach virtual memory system, so that there is a region where it is safe to map the memory because Mach's VM will not put anything else there.

The page fault handler then checks for mappings created by `mapcap`, and backs the pages appropriately. The Mach VM system does not know anything about these mappings, of course, but the vblock daemon can access the data, and copy it into another buffer that can be used with ordinary system calls.

4.5 Build System Changes

The Darbat build system had to be modified not only to build the new virtualisation servers, but to allow an arbitrary number of Darbat kernels to be built. The former was not difficult, but the latter took a bit of thought. The build scripts being used did not have any easy way to create a new image by simply relocating an existing one. The solution that I eventually employed was to add a slightly modified version of the function used to build an application. The regular version named the application based on the name of the directory containing its source. My modified version allows a name to be specified separately, so multiple applications can be built from the same source. I then made the script that builds the Darbat server take a numeric parameter, from which it then derives its name.

4.6 Asynchronous Disk I/O

I implemented support for asynchronous disk I/O operations in Darbat. This involved a few minor changes in the disk device code in the Darbat kernel itself, but most of the work was done in libvblock (see Section 4.3.5) and I/O Kit.

In I/O Kit, an extra thread was added. The main thread still receives disk messages, but it now simply places them in a queue (protected by a mutex) and wakes up the new thread using a semaphore. The thread doing the disk operations simply takes descriptors off the queue, calls the driver, and sends a message to the request's originator to signal its completion.

4.7 Difficulties encountered

The sheer size of the Darbat code base proved to be a source of much difficulty during development. Besides the time needed to become familiar with the overall layout and the locations of specific files of interest, it was often extremely difficult to determine the semantics of many sparsely-documented functions because of the many layers of indirection that were typically traversed.

In particular, it took a large amount of code reading and several failed attempts to discover that the Mach virtual memory system did not appear to be capable of mapping arbitrary physical memory at an arbitrary virtual location, as required by the vblock daemon.

Chapter 5

Evaluation

In this chapter, I will present an evaluation of my work in terms of both the qualitative features that have been added, and quantitative performance impact of those features.

5.1 Qualitative Evaluation

I have demonstrated that the system can run three Darbat kernels simultaneously. Adding more is limited only by lack of available memory in Iguana's shared address space. Each kernel is built from the same source, though multiple copies of the binary must currently be relocated so that they can be loaded in the shared address space.

All guests run in separate hardware address spaces, so a fault in one cannot affect the others unless it was providing a service to them.

The system starts with one guest instance (Darbat0) running, and others can be launched when desired. All of the instances can send their console output to the vconsole server, and it is displayed on screen if the user has selected that particular instance to be visible. The user can change which guest's output is visible on the screen with a keystroke, which also sets which guest receives characters from the keyboard.

All of the guests can access vblock devices that have been allocated to them through the vblock server. Darbat0 is allocated the physical disk, and any running guest can run the vblock_daemon program to host a new vblock device, which can then be allocated to a new guest.

All guests can communicate with I/O Kit using its native API via the vdev server, though I/O Kit's behaviour with multiple guests using this feature has not been extensively tested. Arguments and return values are certainly communicated correctly, at least.

In summary, the main goals of this thesis, to run multiple Darbat kernels on a single machine simultaneously, and to give them all access to a selection of essential devices, have been achieved.

5.2 Quantitative Evaluation

In this section, I will present my measurements of the performance impact of the virtualisation features I have added. I had planned to run a whole-system benchmark such as AIM7 [AIM], but all such programs that I tried either proved very difficult to build, or failed to run on Darbat due to its incomplete state.

The structure of the virtualised system suggests that the main performance impact would be added latency when doing disk operations, due to all disk messages being routed through the vblock server. I therefore decide to use the IOzone filesystem benchmark [Cap06] to measure the latency of disk I/O operations.

I have taken measurements on the following configurations:

- Native Darwin/Mac OS X,
- The unvirtualised Darbat system,
- The virtualised Darbat system running a single guest,
- The virtualised Darbat system running a single guest, but using only synchronous I/O, and
- The virtualised Darbat system running two guests.

5.2.1 Hardware Used

All tests were run on an Apple Mac mini with a 1.5 GHz Intel Core Solo processor. It had a Serial-ATA hard disk drive with a capacity of 55.89 GiB. Tests were run using files on a 34.88 GiB partition, which was formatted with a Journaled HFS+ filesystem. The partition had 16.22 GiB of free space. The system also had 512 MiB of RAM, though that is somewhat irrelevant, as Darbat currently only makes use of a fixed amount of memory.

5.2.2 Experimental Method

Measurements were taken with IOzone version 3.281. A few small changes were made to its source code, to enable cross-compilation for an x86 target on the PowerPC development machine being used, and to reduce loss of precision introduced by unnecessary casting of a value between integer and floating point.

Four different configurations were tested, with three runs on each. The machine was rebooted between runs. Here are the descriptions of the configurations, along with the abbreviations that will be used to refer to them in the following sections:

- osx: Mac OS X 10.4.8, started in single user mode to match Darbat.

- darbat-novirt: Darbat configured to communicate directly with I/O Kit. Vblock, vconsole, vdev and libvblock are not present.
- darbat0: Virtualised Darbat with one guest running. The guest uses the real disk partition through vblock.
- darbat0-sync: Same as darbat0, but with asynchronous disk I/O disabled.
- darbat1: Virtualised Darbat with two guests running. The first guest is the same as darbat0, but it now hosts a vblock device backed by a disk image in its filesystem, with a total size of 6 GiB and 1.51 GiB of free space. IOzone is run under the second guest, which uses the device hosted by the first.

IOzone was invoked with the command line, "`./iozone -a -g 32768 -L 64 -S 2048 -b latest-latency.xls -N -o`". The flags given have the following effects:

- `-a` enables automatic mode. IOzone chooses a range of file and record sizes to use.
- `-g 32768` sets the maximum file size to 32768 KiB. This is used because larger sizes currently cause Darbat to crash, for reasons that are not yet fully understood.
- `-L 64 -S 2048` tells IOzone the processor's cache line size in bytes and total cache size in KiB, respectively.
- `-b latest-latency.xls` causes output to be saved in a binary spreadsheet file with the given name.
- `-N` causes results to be reported as microseconds per operation, rather than as KiB per second.
- `-o` causes the test files to be opened with the `O_SYNC` flag. This means that write calls will not return until the data has actually been sent to the disk. Thus, the true cost of doing I/O is measured, rather than the cost as modified by the buffer cache. This is important because of the file size limitation noted above.

5.2.3 Results

Figure 5.1 shows the time taken to complete a write call for the tested range of record sizes, with a representative file size of 16 MiB. Figure 5.2 shows the time taken to complete a read call for the tested range of record sizes, once again with a file size of 16 MiB.

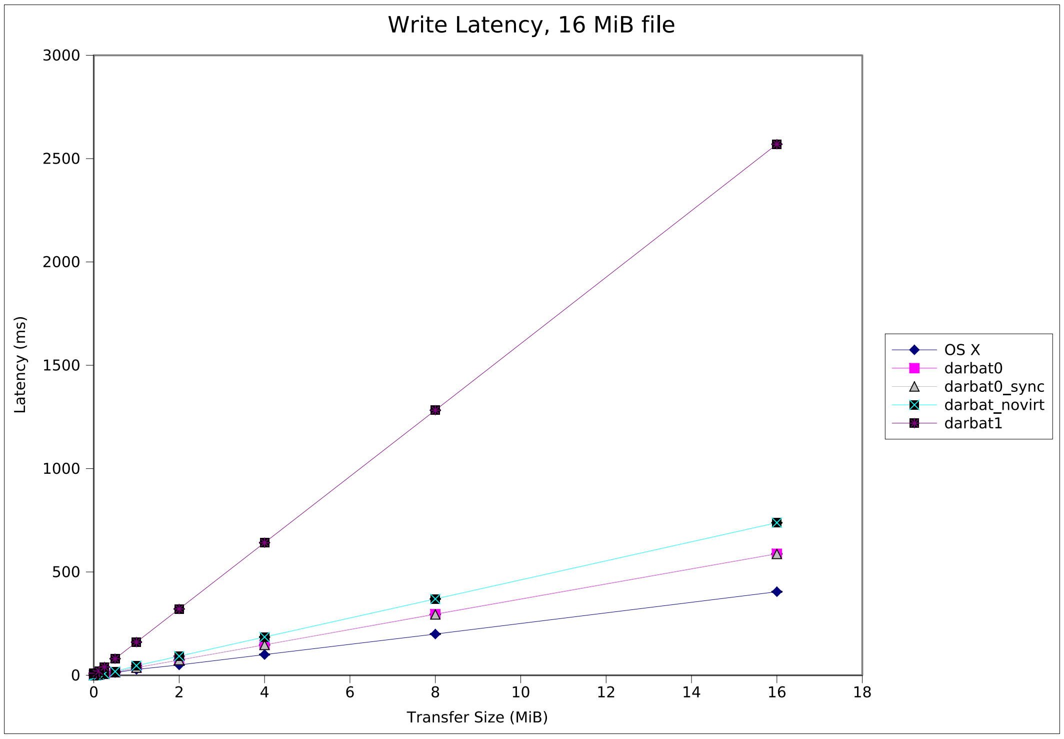

Figure 5.1: Comparison of write latencies.

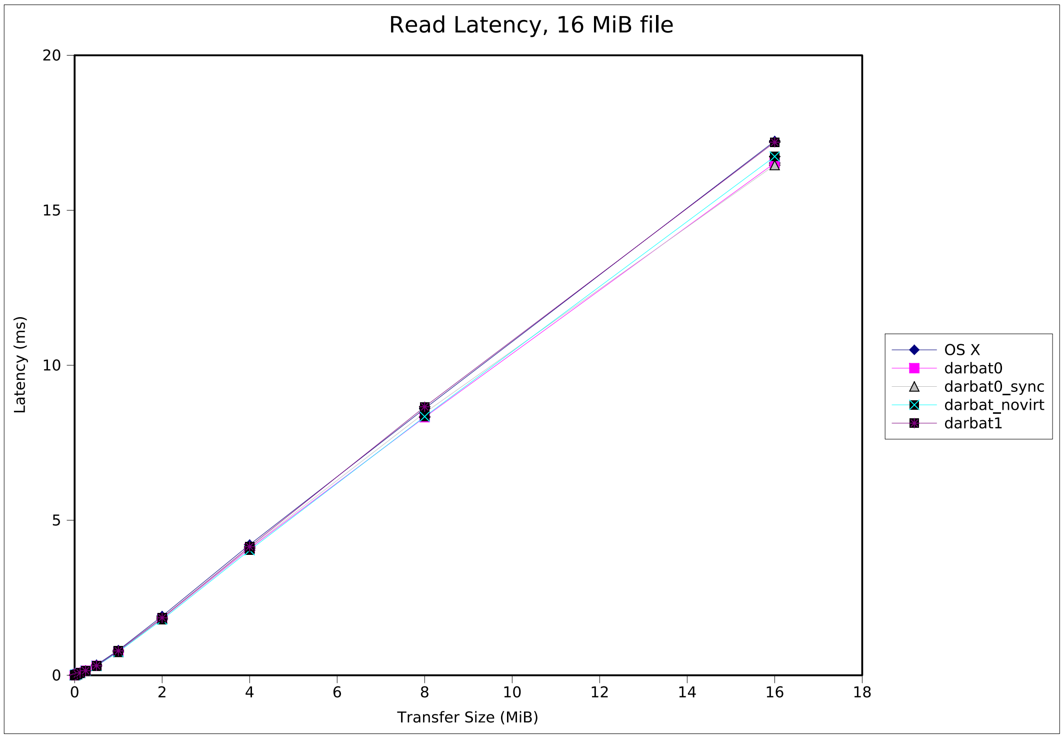

Figure 5.2: Comparison of read latencies.

Configuration	Section	μ (μs)	σ_{N-1}
darbat-novirt	disk_[read/write]	97.7	1.247
	disk_rdwr	28.3	2.055
	I/O Kit	26.7	3.091
	Driver	832	3.559
darbat0-sync	disk_[read/write]	93.7	1.700
	vblock_rw	36.3	3.300
	I/O Kit	24.3	3.399
	Driver	813	6.976

Table 5.1: Time spent in various parts of the call sequence for performing a disk I/O operation.

The write latency results are counterintuitive in that darbat-novirt took longer than darbat0. To investigate this phenomenon more closely, the darbat0-sync and darbat-novirt configurations were compiled with code that measured the average time taken to complete four different parts of the call sequence. The same IOzone command as was used for the previous tests was run on these instrumented kernels, producing average times computed from around 118,000 reads and writes. The difference in cumulative times was calculated to determine the time spent in each layer, excluding the time spent in lower layers. The results, averaged over three runs, are shown in Table 5.1. The sections are named as follows:

disk_[read/write] The outermost function where the code differs at all between the configurations. It selects a buffer to use in the virtualised case. In both cases, it copies data in or out of the buffer, and calls the next function in the sequence.

disk_rdwr / vblock_rw The function that performs the IPC, to I/O Kit in the unvirtualised case, or to vblock in the virtualised case. This code is in libvblock in the virtualised case.

I/O Kit Processing done by the I/O Kit server that is not in the disk driver itself.

Driver Time spent in the actual disk driver code.

Figure 5.3 shows nonlinear fluctuations in write latency when using small record sizes, and Figure 5.4 illustrates that darbat1 always takes at least 10 ms to complete a write operation.

Summary of Results

Read latencies are extremely close between all tested configurations. Write latencies for darbat0 and darbat-novirt are considerably higher than for na-

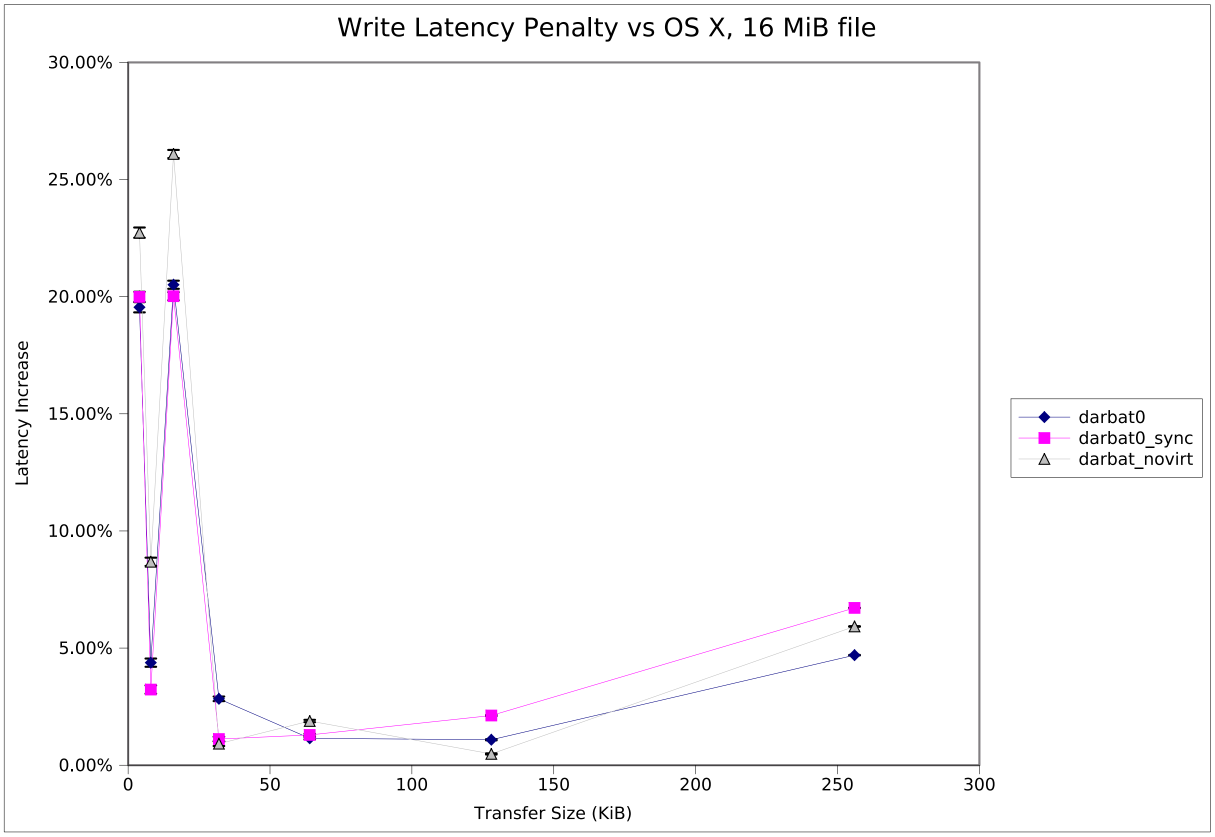

Figure 5.3: Nonlinear behaviour with small transfer sizes.

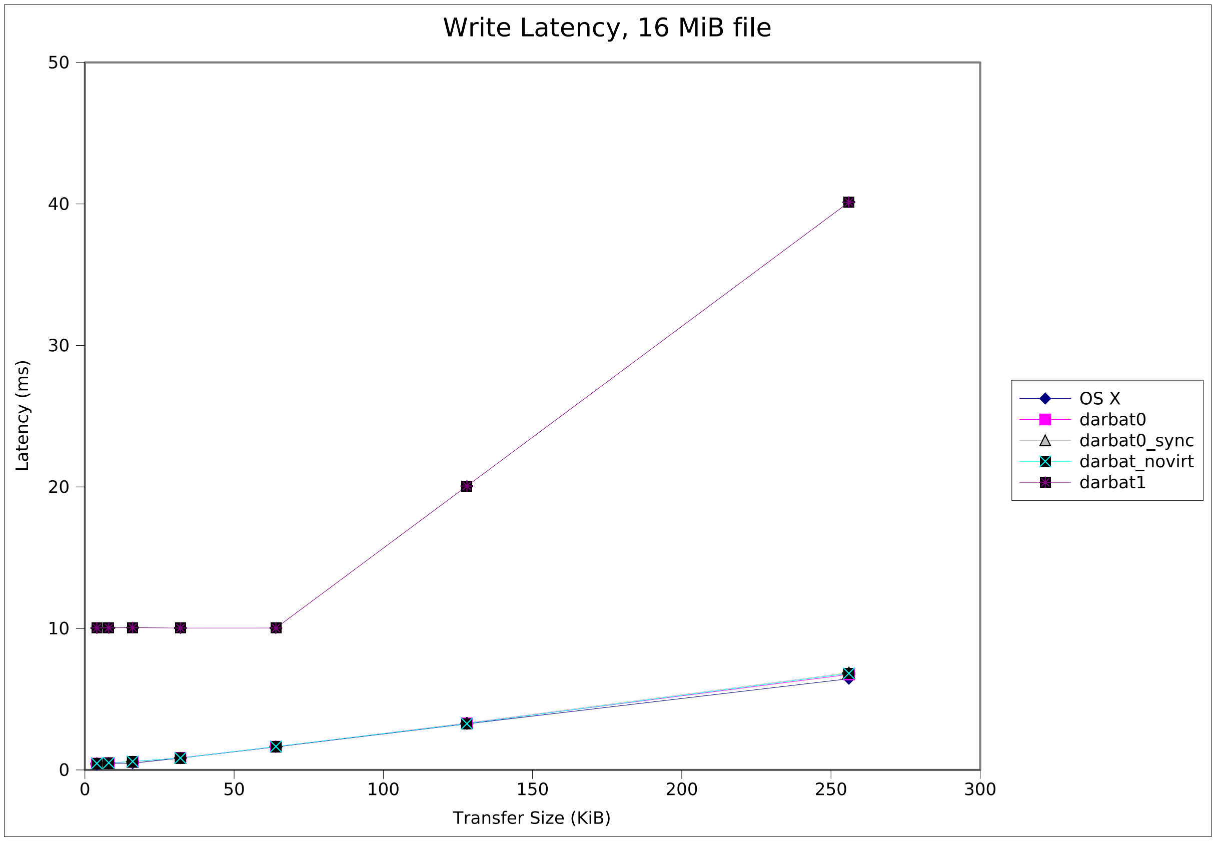

Figure 5.4: Darbat1 takes a minimum of 10 ms to complete a transfer.

Configuration	μ (increase over OS X)	σ_{N-1}	SEM
darbat-novirt	0.612	0.754	0.079
darbat0-sync	0.431	0.556	0.059
darbat0	0.440	0.557	0.059
darbat1	12.775	6.990	0.737

Table 5.2: Mean write latency penalties, expressed as the relative increase compared to OS X. Sample standard deviation and the standard error of the mean are also given.

tive Mac OS X, and darbat1 exhibits extremely write high latencies. Darbat0 has lower latencies than darbat-novirt.

Darbat-novirt spends noticeably more time in the disk driver than darbat0, and also spends more time in the function that copies data in and out of the buffer shared with I/O Kit. Darbat0 takes longer to communicate with I/O Kit than darbat-novirt, but this is outweighed by the places where darbat-novirt spends more time.

5.2.4 Analysis

To obtain a relative performance metric, each of the Darbat configurations' write latencies was expressed as a percentage increase compared to Mac OS X. This was calculated by taking the difference between each Darbat value and its corresponding OS X value, and dividing by the OS X value. Averaging these results across the whole samples produces the numbers shown in Table 5.2.

Figure 5.5 shows the relative write latency increases compared to Mac OS X for the tested range of record sizes, with a representative file size of 16 MiB.

Uncertainty Analysis

In this section, I will calculate the uncertainty in the results and the values derived from them.

IOzone claims that its timing is accurate to the nearest 1 μs. However, it truncates its floating point results when it converts them to integers for output, rather than rounding them correctly. The reported values may therefore be up to 1 μs less than they should be. Therefore the uncertainty in each value is $\pm 1.5\mu$s. Adding 0.5 to each value corrects the downward bias from the truncation, making the error in the corrected values $\pm 1\mu$s.

The mean is the sum of all the sample elements, divided by the number of elements. Uncertainties add in quadrature when their corresponding values are added, and are divided by exact constants along with their values. Therefore the uncertainty in the mean is the quadrature sum of all the

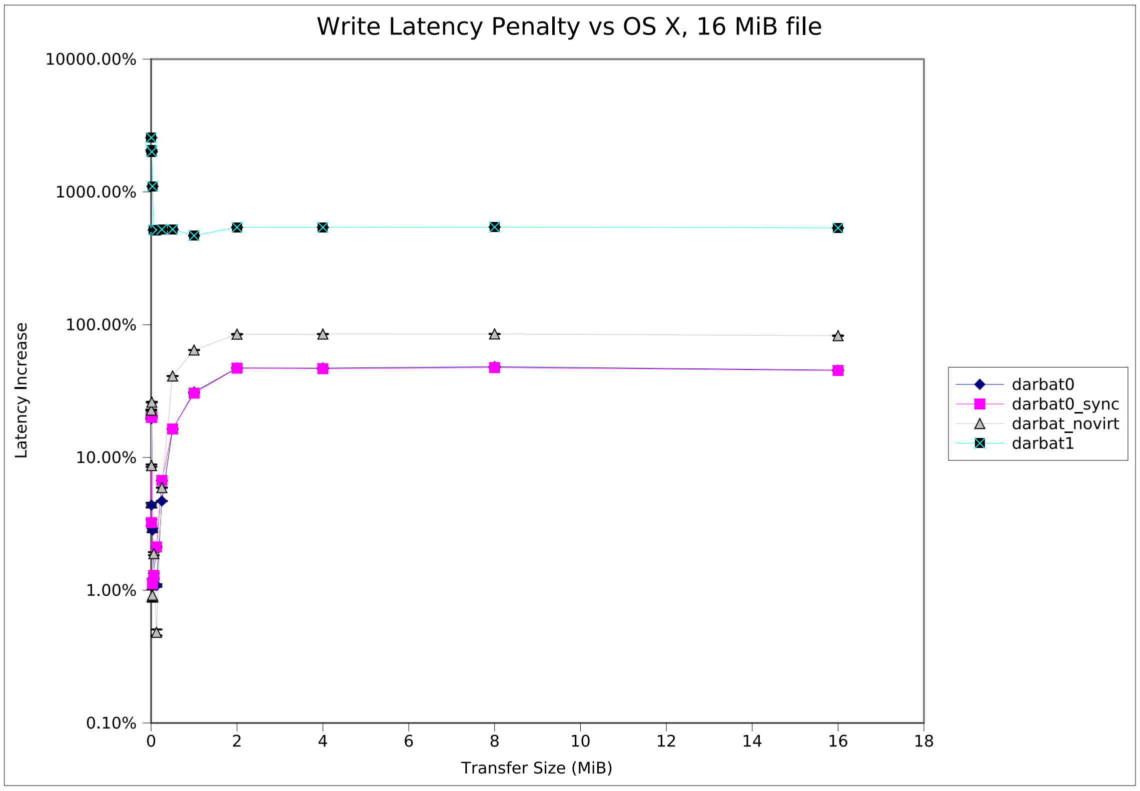

Figure 5.5: Write latency penalties relative to OS X.

values divided by the number of values:

$$\frac{\sqrt{1^2+1^2+1^2}}{3} = \pm\frac{\sqrt{3}}{3} \approx \pm 0.577\mu s$$

Standard deviation is calculated as the root mean square difference between the measurements and the sample mean. To calculate the uncertainty in the sample's standard deviation, we therefore add each measurement's uncertainty to that of the mean in quadrature. We then double the result (due to the squaring), sum all such results in quadrature and divide the sum by $N-1$ (corresponding to the calculation of the mean), then halve the result due to taking the square root to get σ from σ^2. Symbolically, this process is as follows:

$$\frac{\frac{\sqrt{3\cdot\left(2\cdot\sqrt{1^2+\Delta\mu^2}\right)^2}}{2}}{2} = \pm 1.0$$

Therefore, on the graphs that show latencies, I have used error bars indicating $\pm\sigma + 1.0$. At the scales used, they are almost too small to see.

Taking the difference between the set of means for two different configurations makes the uncertainties add in quadrature:

$$\Delta(\mu_1 - \mu_2) = \sqrt{\Delta\mu_1^2 + \Delta\mu_2^2} = \pm\sqrt{\frac{2}{3}} \approx \pm 0.816\mu s$$

When we normalise all the differences to a percentage increase, the relative uncertainties add in quadrature. Letting the difference in means be d, we get:

$$\frac{\Delta X}{X} = \sqrt{\left(\frac{\sqrt{\frac{2}{3}}}{d}\right)^2 + \left(\frac{\frac{\sqrt{3}}{3}}{\mu_{osx}}\right)^2}$$

This relative uncertainty has been used for the error bars on the graphs in this section that show latency increases compared to OS X. At the scales that have been used, the bars are almost too small to see.

5.2.5 Discussion

Relative performance

Darbat is currently in an unstable state and is missing major features, so it has not been subjected to a concerted optimisation effort. It is thus not surprising that it performs worse than Mac OS X in these tests. It is also not surprising that asynchronous I/O makes things slightly slower. Only one I/O operation is being issued at a time, so it will be completed more quickly if the kernel immediately waits for I/O Kit to complete it, rather than performing other operations before getting back to receiving the reply. The most surprising result is that the Darbat systems using vblock are in fact faster than those without it.

The measurements in Table 5.1 were taken to aid in understanding this result. Darbat-novirt spends significantly more time in the disk driver, and moderately more time in the outer disk_read / disk_write functions. Intuition is at least satisfied by the fact that darbat0 spends more time in the function that performs the IPC, as this also includes the time spent in the vblock server.

More information is needed to satisfactorily explain what is happening, but a rough guess may be hazarded. The two places that exhibit unexpected slowness have a feature in common, namely that they access the shared buffer. Darbat-novirt has a single 64 KiB buffer, while darbat0 allocates buffers as needed, and only makes them as large as is needed at the time. It may be that darbat-novirt's buffer has an unfortunate placement, and that some memory system effect, perhaps cache aliasing or a quirk of the page-fault handling, is causing the loss of performance.

It would be interesting to see what difference it would make to eliminate two IPC calls per I/O operation by having vblock's clients and providers send messages to each other directly most of the time, only using the vblock server to initially find each other. It seems likely that the 8 μs difference between vblock_rw and disk_rdwr is close to the true overhead added due to the use of vblock.

Effects of record size

The nonlinear behaviour seen in Figure 5.3 shows that the behaviour of the filesystem is complex. With larger record sizes, the time taken to transfer the data dominates the overall latency, so irregularities such as these become less apparent.

The levelling-off of the curves in Figure 5.5 occurs at 2 MiB, which is the size of the processor's L2 cache. The more of the data that fits in the cache, the less penalty there is for copying it between buffers. The fact that exhausting the cache capacity makes the performance so much worse compared to native OS X suggests that Darbat's performance could be markedly increased by eliminating the copy between the buffer cache and the buffer shared with the device provider. This could be done by implementing scatter-gather I/O, which would require Darbat being able to determine the real physical addresses of pages in its buffer cache. A list of physical frames could then be sent to I/O Kit, which could perform DMA directly from the memory backing the buffer cache pages.

Darbat1's performance

Darbat1 exhibits very poor performance, both in terms of overhead per operation and cost per byte. The main factor that explains this is that the vblock daemon must poll for messages because the Darbat kernel cannot

Configuration	Latency (μs)	Penalty vs. native
Linux	66.0	N/A
Xen	68.2	3.3%
VMware	85.6	29.7%
UML	250	279%

Table 5.3: Filesystem latency for creating a 10 KiB file in native Linux, XenoLinux, Linux running under VMware Workstation, and User-Mode Linux. Measurements reproduced from [BDF$^+$03].

preempt it. It may be possible to partly alleviate this problem by adjusting the daemon's sleep intervals, but effort would be better spent in fixing the underlying problem by implementing preemption.

The other main factor contributing to darbat1's poor performance is the fact that it must copy the data three more times than darbat0: from userland into darbat1's buffer cache, from there into the buffer shared with the vblock daemon, and from there into a buffer that darbat0 will accept in a read/write syscall. After that, it must of course go through the usual sequence for darbat0's disk I/O. It may be possible to eliminate one copy from darbat1's I/O path by writing a custom pager. This could make it possible to map an arbitrary piece of physical memory into a user process's address space. It does not appear to be possible to do that with the standard Mach virtual memory system.

The performance of disk I/O using the vblock daemon should be re-evaluated when Darbat gains the ability to preempt its user processes.

Effects of the buffer cache

While the measured write latencies showed the cost of sending data all the way to the disk, due to the use of the `O_SYNC` flag, the read latencies are measuring a different quantity. Read latencies for all of the configurations are both very close together, and very low compared to the write latencies. This is because the data being read was all able to fit in the buffer cache.

This illustrates that even with extremely poor performance of actual disk I/O such as that exhibited by darbat1, many file operations will, in practice, not perform any worse. Of course, it would be a different story if the capacity of the buffer cache had been exhausted.

Comparison with other virtualised systems

Unfortunately, disk I/O latency does not appear to be a popular metric for evaluating VMMs in the literature. Most papers focus on throughput and CPU utilisation with multiple concurrent requests. However, the original

2003 Xen paper [BDF$^+$03] does have a comparison of filesystem latencies for native Linux, XenoLinux, Linux running under VMware Workstation, and User-Mode Linux. The data for XenoLinux was gathered when running it as Domain0, and no data for other domains using devices provided by Domain0 is given. Table 5.3 reproduces their results for the creation of a 10 KiB file.

These few data points are not sufficient to compare my system with these others in any meaningful way, especially since I negated the benefits of the buffer cache for my tests, and these did not. They do, however, appear to suggest that Darbat0's 44% penalty over native (without the benefit of the buffer cache) may be quite competitive. This penalty could in fact be greatly reduced if copying between guests and I/O Kit were avoided, as described earlier in this section.

What was not measured

Disk I/O latency is only one part of the performance impact of the modifications I have made. CPU usage and throughput are the other metrics that tend to be used to evaluate such changes. Unfortunately, measuring these would have been considerably more difficult than measuring the latency for a single process. Most throughput benchmarks involve many processes or the asynchronous I/O API, and Darbat is unable to run them in its current state.

It was also not possible to measure the complete disk I/O performance characteristics of the systems, since files large enough to exhaust the capacity of the buffer cache could not be used.

5.2.6 Performance Summary

A virtualised Darbat instance using the real disk takes about 44% longer than native Mac OS X to complete a disk I/O operation. A Darbat instance without the virtualisation servers took significantly longer, but it appears likely that this is only due to chance. The real overhead added by the virtualisation servers is around 8 μs per operation, possibly slightly lower. It should be possible to reduce this by having Darbat communicate with the providers of its vblock devices directly, rather than going through the vblock server every time.

It appears that the higher latency for Darbat compared to Mac OS X is mostly due to the fact that the data is copied an extra time. It should be possible to eliminate this extra copy by implementing scatter-gather DMA.

A Darbat instance using a disk image provided by the vblock daemon running under another Darbat instance exhibits extremely high latencies. This is explained by the fact that it needs to poll for messages due to Darbat's inability to preempt its user processes, combined with the fact that it must copy the data several more times.

Chapter 6

Future Work

The work presented in this thesis will enable many future developments. This chapter will discuss some of them. Figure 6.1 shows the structure of a possible future virtualised version of Darbat. The Darbat kernels are run in external address spaces, and there are multiple I/O Kit instances. The latter may also be run in external address spaces. Wombat is run alongside Darbat, and makes use of the same virtualised devices.

6.1 Vblock improvements

It should be possible for vblock to send the details of each device's provider to guests. Then, the guest could communicate with the provider directly, rather than going through the vblock server in every call. This should decrease disk I/O latency by some amount.

The vblock server should also have a protocol for advertising the presence of newly-added devices to guests, so that hot-plugging drives will work.

6.2 External address spaces for guests

It is planned that guests will eventually be launched in external address spaces. This will relieve the address space scarcity problems that currently affect Darbat, since the XNU kernel expects to have a full 4 GiB address space to work with. It will also remove the need to have multiple copies of the same kernel that only differ in having been relocated to a different address, since each address space will be able to have the same kernel image loaded in-place.

When this happens, the supervisor will need to be the exception handler and scheduler for the guests, a role that is presently filled by the Iguana server. Depending on the security model that is chosen, the supervisor may also act as redirector as well.

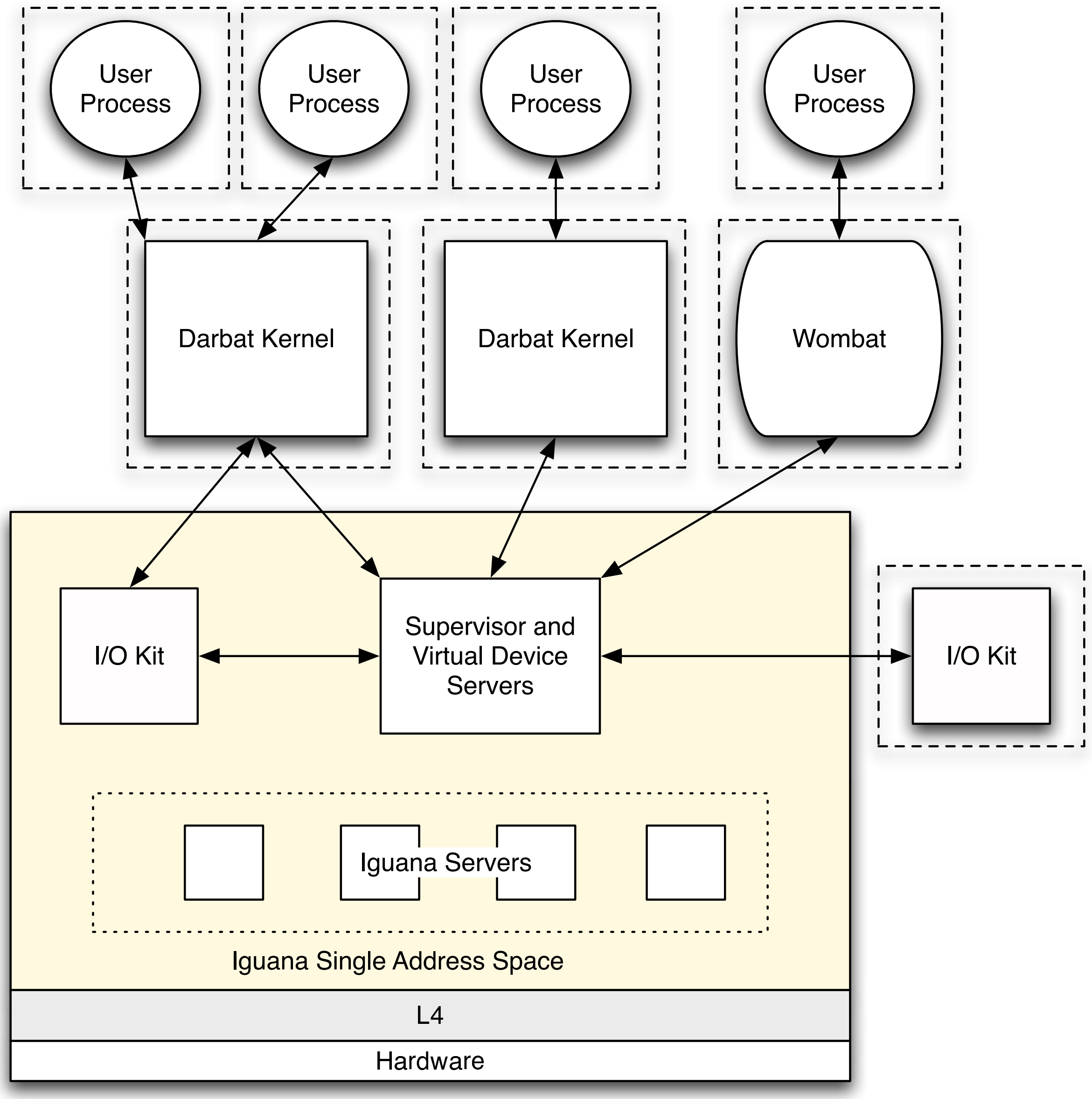

Figure 6.1: Possible future structure of a virtualised Darbat system. Dashed lines indicate external address spaces.

6.2.1 Memory Server

This server will act as the pager for the guests when they are launched in external address spaces. It will give memory to guests on demand until they reach their maximum allocation. It may also communicate with balloon drivers in the guests to reclaim memory when needed.

The memory server will be important in enabling guest instances to be brought down cleanly, since it will keep track of the memory resources that each one uses.

6.3 Network

A network virtualisation server could be implemented, to allow all guests to have network access. It would likely implement a virtual NIC per guest and a virtual ethernet switch. Packets could be sent and received in much the same way as block devices are read and written, except that a MAC address will be given rather than a block number.

6.4 Video and Audio

Servers could be implemented to multiplex video and audio devices between guests. These servers would assign video and audio devices to guests in much the same way as the keyboard is assigned. More complex schemes could also be explored, for example, optionally mixing audio output from multiple guests simultaneously. The video display could be shared using a system such as Blink [Han06] or Nitpicker [FH05].

6.5 Multiple I/O Kits

Multiple I/O Kits could be used to isolate unreliable drivers. This would require a system for configuring which devices each I/O Kit should handle. The vdev server would also need to be extended to keep track of which devices are provided by which I/O Kit instance. For security, it could also keep track of which guests should be allowed to access which devices.

To realise the benefits of isolated drivers, it would also be necessary to implement a system to detect when they fail, and to restart them with as little impact on the correct operation of user processes as possible.

6.6 Other future work

Listed here is selection of other interesting work that may follow on from the work presented in this thesis.

- fair scheduling of guest kernels and applications;
- support for overcommitting memory to increase utilisation, possibly via balloon drivers;
- hot-plug support for all relevant devices;
- runtime starting (and stopping) of I/O Kits for new drivers and devices;
- a secure user interface for VM management;
- support for running Wombat guests (and other suitably modified operating systems, including hardware-assisted VMMs for legacy OS support).

Chapter 7

Conclusions

In this thesis, I have described the design and implementation of a paravirtualised system supporting multiple instances of the Darbat OS, which runs in the Iguana environment on the L4 microkernel. I have reported on the successful operation of the system, and measured a part of the performance impact of the new features. I have also described some of the future work that will be enabled by the previous achievements.

The qualitative goals have been met quite successfully, as multiple Darbat instances have been run simultaneously, and have been given multiplexed access to a basic set of devices.

The performance impact has not been fully characterised, but initial results are promising. The impact on disk I/O latency due to the use of the virtualisation servers appears to be small compared to other factors. Strategies have been identified to overcome the factors that lead to reduced performance compared to native Darwin. It does not seem unreasonable at this stage to think that disk performance comparable to native could be achieved with careful optimisation.

Bibliography

[AA06] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, San Jose, California, USA, October 2006.

[AIM] Aim benchmarks. http://sourceforge.net/projects/aimbench.

[App] Apple Darwin Releases. http://www.opensource.apple.com/darwinsource/.

[App06a] Apple, Inc. Darwin project FAQ. http://developer.apple.com/opensource/faq.html, 2006.

[App06b] Apple, Inc. Introduction to I/O Kit fundamentals. http://developer.apple.com/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/, November 2006.

[App06c] Apple, Inc. Kernel programming guide. http://developer.apple.com/documentation/Darwin/Conceptual/KernelProgramming/, November 2006.

[App06d] Apple, Inc. Mac OS X technology overview. http://developer.apple.com/documentation/MacOSX/Conceptual/OSX_Technology_Overview/, June 2006.

[BDF$^+$03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on OS Principles*, pages 164–177, Bolton Landing, NY, USA, October 2003.

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, Anaheim, CA, April 2005.

[Cap06] Don Capps. IOzone filesystem benchmark. http://www.iozone.org/, October 2006.

[CB93] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on OS Principles*, pages 120–133, Asheville, NC, USA, December 1993.

[CH05] Matthew Chapman and Gernot Heiser. Implementing transparent shared memory on clusters using virtual machines. In *Proceedings of the 2005 USENIX Technical Conference*, pages 383–386, Anaheim, CA, USA, April 2005.

[Chu04] Peter Chubb. Get more devices drivers out of the kernel! In *Ottawa Linux Symposium*, Ottawa, Canada, July 2004.

[Cla85] David D. Clark. The structuring of systems using upcalls. In *Proceedings of the 10th ACM Symposium on OS Principles*, pages 171–180. ACM Press, 1985.

[Cre81] Robert J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, 1981.

[Dik00] Jeff Dike. A user-mode port of the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, Georgia, USW, October 2000.

[FH05] Norman Feske and Christian Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 85–94, Washington, DC, USA, December 2005. IEEE Computer Society.

[GLB06] Charles Gray, Geoffrey Lee, and Tom Birch. Darbat release 0.2 notes. http://ertos.nicta.com.au/downloads/darbat/ReleaseNotes-0.2.pdf, 2006.

[Han06] Jacob Gorm Hansen. Blink: 3D multiplexing for virtualized applications. Technical Report 06/06, Dept. of Computer Science, University of Copenhagen, January 2006.

[HHL$^+$97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on OS Principles*, pages 66–77, St. Malo, France, October 1997.

[HL04] Gernot Heiser and Ben Leslie. Iguana: A protection and resource manager for embedded systems. http://www.ertos.nicta.com.au/software/kenge/iguana-project/latest/iguana_talk.pdf, August 2004.

[HR96] Michael Hohmuth and Sven Rudolph. Steps towards porting a Unix single server to the L3 microkernel, April 1996.

[HUL06] Gernot Heiser, Volkmar Uhlig, and Joshua LeVasseur. Are virtual-machine monitors microkernels done right? *Operating Systems Review*, 40(1):95–99, January 2006.

[Lee05] Geoffrey Lee. I/O Kit drivers for L4. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 2005.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

[Lie96] Jochen Liedtke. Towards real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[Lin06] Itanium Linux-on-Linux. http://ertos.nicta.com.au/software/virtualisation/lol.pml, December 2006.

[LUC+05] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical Report PA005520, National ICT Australia, October 2005.

[LUSG04] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, USA, December 2004.

[LvSH05] Ben Leslie, Carl van Schaik, and Gernot Heiser. Wombat: A portable user-mode Linux for embedded systems. In *Proceedings of the 6th Linux.Conf.Au*, Canberra, April 2005.

[Mac07] Mac-on-Linux web site. http://mac-on-linux.sourceforge.net/, February 2007.

[MS05] Steve McDowell and Geoffrey Strongin. Pacifica – next generation architecture for efficient virtual machines. http://developer.amd.com/assets/WinHEC2005_Pacifica_Virtualization.pdf, 2005.

[NIC05] National ICT Australia. *NICTA L4-embedded Kernel Reference Manual Version N1*, October 2005. http://ertos.nicta.com.au/Software/systems/kenge/pistachio/refman.pdf.

[NIC06a] Magpie. http://ertos.nicta.com.au/software/kenge/magpie/latest/, June 2006.

[NIC06b] Project: Iguana. http://www.ertos.nicta.com.au/software/kenge/iguana-project/latest/, June 2006.

[Obj04] Object Management Group, Inc. *CORBA 3.0.3 Specification*, March 2004.

[OJG91] D. L. Osisek, K. M. Jackson, and P. H. Gum. ESA/390 interpretive-execution architecture, foundation for VM/ESA. *IBM Syst. J.*, 30(1):34–51, 1991.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):413–421, 1974.

[RI00] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, August 2000.

[RJO$^+$89] R.F. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: a system software kernel. *Spring COMPCON*, pages 176–8, 1989.

[UNR$^+$05] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[VMw07] VMware, Inc. VMware products. http://www.vmware.com/products/, 2007.

[Wal02] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, USA, 2002.

[Won03] Ka-shu Wong. Mac OS X on L4. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, December 2003.

[WSG02] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 2002.

Appendix A

Vblock interface definitions

A.1 The libvblock interface

The header file `vblock.h`, which defines the interface to libvblock, is reproduced below.

```
/*
 * vblock.h
 * client interface for Darbat virtual block device server
 *
 * Joshua Root <jmr at cse.unsw.edu.au>
 * created 2006-10-25
 * Copyright 2006 National ICT Australia.
 */

#ifndef _VBLOCK_H
#define _VBLOCK_H 1

/*
 * Initialise libvblock. Should be called before any other
 * libvblock functions. 'completion_func' is a function that will
 * be called if libvblock receives a completion message during
 * other operations. Note that completion messages must also be
 * handled by the client, as libvblock does not handle them unless
 * they are received while one of its functions is executing.
 * Return value is 0 on success, or a BSD error code on failure.
 */
int vblock_init(void (*completion_func)(char *, int));

/*
 * Obtain a buffer of at least 'size' bytes that can be used with
 * the read and write functions. The return value is an opaque
 * reference to the buffer that can be passed to libvblock
 * functions. The actual base address of the buffer is returned
 * in 'base', and should be used for copying data in or out.
 * Returns a valid pointer on success, or NULL on failure.
 */
void * vblock_get_buffer(unsigned int size, char **base);

/*
 * Return a buffer, identified by 'buf_ref' (which should have
 * been obtained from vblock_get_buffer()), to the pool of
 * available buffers.
 */
void vblock_return_buffer(void *buf_ref);

/*
 * Get the geometry of the device identified by 'devnum'. On
 * success, 'nblocks' is set to the number of blocks on the
 * device, and 'blocksz' is set to the device's block size. On
 * failure, both 'nblocks' and 'blocksz' are set to 0.
 */
void vblock_get_geometry(int devnum, unsigned long *nblocks,
                         unsigned long *blocksz);

/*
 * Read or write 'size' bytes from/to location 'offset' in the
 * device identified by 'devnum', into/from the buffer specified
 * by 'buf_ref' (which should have been obtained from
 * vblock_get_buffer()).
 *
 * The non-async variants wait until the I/O is actually complete
 * before returning, while the async variants return as soon as
 * the request has been enqueued (or if an error occurs while
 * enqueuing it). At some point after a successful async call,
 * the calling thread will receive a completion message that
 * indicates that the I/O operation has actually completed.
 *
 * Return value is 0 on success, or a BSD error code on failure.
 */
int vblock_read(int devnum, void *buf_ref,
                unsigned long long offset, unsigned int size);
int vblock_read_async(int devnum, void *buf_ref,
                      unsigned long long offset, unsigned int size);
int vblock_write(int devnum, void *buf_ref,
                 unsigned long long offset, unsigned int size);
```

```
int vblock_write_async(int devnum, void *buf_ref,
                       unsigned long long offset, unsigned int size);

/*
 * Returns the number of devices currently available through vblock.
 */
int vblock_ndevs(void);

/*
 * Ask the device's provider to flush its buffer cache.
 * Return value is 0 on success, or a BSD error code on failure.
 */
int vblock_sync(int device_number);

#endif /* _VBLOCK_H */
```

A.2 The Vblock RPC interface

The IDL4 definition of the RPC interface to the vblock server, from the file `vblock.idl4`, is reproduced below.

```
/*
 * vblock.idl4
 * RPC interface to Darbat virtual block device server
 */

import "iguana/types.h";
import "vblock/interface_uuids.h";

/* client interface */
[uuid(INTERFACE_VBLOCK_CLIENT_UUID)]
interface vblock_
{
      /*
       * Register with the server. Must be called before using any
       * other vblock functions. Returns the number of devices
       * available to the client in 'ndevs'. Return value is 0 on
       * success, or a BSD error code on failure.
       */
      int init(out int ndevs);

      /*
       * Gets the geometry of the device specified by 'device_number'.
       * Returns the number of blocks in the device in 'nblocks' and
```

```
        * their size in 'blocksize'.
        * Return value is 0 on success, or a BSD error code on failure.
        */
       int get_geometry(in int device_number, out unsigned long nblocks,
                        out unsigned long blocksize);

       /*
        * Read or write 'size' bytes from/to the device specified by
        * 'device_number', into/from the address given in 'buffer',
        * starting at the offset given in 'offset'. The operation is
        * asynchronous, so the function returns as soon as the request
        * has been enqueued, or when an error occurs while attempting to
        * enqueue it. The calling thread will be sent a completion
        * message when the operation actually completes.
        * Return value is 0 on success, or a BSD error code on failure.
        */
       int read(in int device_number, in char *buffer,
                in unsigned long long offset, in unsigned long size);
       int write(in int device_number, in char *buffer,
                 in unsigned long long offset, in unsigned long size);

       /*
        * Ask the device's provider to flush its buffer cache.
        * Return value is 0 on success, or a BSD error code on failure.
        */
       int sync(in int device_number);

       /*
        * Sends the capability 'cap' to the provider of the device
        * specified by 'device_number'. This function should be called
        * with a capability for every buffer that is to be used for read
        * or write calls.
        * Return value is 0 on success, or a BSD error code on failure.
        */
       int add_cap(in int device_number, in cap_t cap);
};

/* interface for clients to export devices */
[uuid(INTERFACE_VBLOCK_PROVIDER_UUID)]
interface vblock_provider
{
       /*
        * Register to provide a device that has 'nblocks' blocks of size
        * 'blocksize'. The id number of the new device is returned in
```

```
         * 'dev_id'.
         * Return value is 0 on success, or a BSD error code on failure.
         */
        int add_device(in unsigned long nblocks,
                       in unsigned long blocksize, out int dev_id);

        /*
         * Indicate that the thread 'client' should be allowed to access
         * the device specified by 'device_number'. The permitted
         * operations are encoded in 'permissions', which can be the
         * values READ_MASK or WRITE_MASK, or their bitwise union.
         * Return value is 0 on success, or a BSD error code on failure.
         */
        int give_client_device(in L4_ThreadId_t client,
                               in int device_number, in int permissions);

        /*
         * Indicate that the device specified by 'dev_id' is no longer
         * available for use. Fails if not called by the provider of the
         * device or the supervisor.
         * Return value is 0 on success, or a BSD error code on failure.
         */
        int remove_device(in int dev_id);
};

/* supervisor interface */
[uuid(INTERFACE_VBLOCK_SUPERVISOR_UUID)]
interface vblock_supervisor
{
        /*
         * Register the thread 'tid' as a client of the vblock server.
         * Return value is 0 on success, or a BSD error code on failure.
         */
        int add_client(in L4_ThreadId_t tid);

        /*
         * De-register the thread 'tid' as a client of the vblock server.
         * Return value is 0 on success, or a BSD error code on failure.
         */
        int remove_client(in L4_ThreadId_t tid);
};
```

Appendix B

List of new and modified files

The Darbat code base is very large, so it may be difficult to distinguish between code that was written for this thesis and code that existed previously. To help address this problem, this appendix lists all of the files that I have added or changed during thesis development, along with descriptions of the new or modified functionality in each.

B.1 New directories

These are the directories that were added to the Darbat source. Most of the files they contain are also completely new, though not all. Notably, vconsole contains substantial amounts of code taken from Mach.

libs/vblock The libvblock library.

virt Directory containing virtualisation-specific programs.

virt/supervisor The supervisor.

virt/utils Directory containing user-mode virtualisation utilities.

virt/utils/guest_launcher Program that launches a new guest.

virt/utils/vblock_daemon The vblock daemon.

virt/vblock The vblock server.

virt/vconsole The vconsole server.

virt/vdev The vdev server.

B.2 New files

These are files that are new, but are in a pre-existing directory. New files in new directories are not listed.

darbat/igcompat/src/iokit/vblock_disk.c Sits between diskdev.c and libvblock. This is the virtualised counterpart to disk.c, and handles async I/O properly.

darbat/igcompat/src/iokit/disk.h Declarations common to disk.c and vblock_disk.c.

darbat/osfmk/console/darbat_vconsole.c Handles communications with the vconsole server.

iguana/init/src/launch_darbat.c Listens for messages from the supervisor and launches new Darbat guests when requested.

B.3 Modified files

These are pre-existing files that I modified in some way.

SConstruct Build multiple Darbats, build virtualisation servers and library, pass virtualisation config option to sub-projects to allow conditional compilation.

darbat/SConstruct Conditionally build virtualisation code, set a unique target name so there can be multiple Darbats.

darbat/bsd/kern/bsd_init.c Initialise vconsole.

darbat/bsd/kern/kern_mib.c Code for virtualisation-related sysctls.

darbat/bsd/kern/kern_sysctl.c Code for virtualisation-related sysctls.

darbat/bsd/kern/subr_log.c Forward log_putc output to vconsole.

darbat/bsd/kern/sysctl_init.c Code for virtualisation-related sysctls.

darbat/bsd/sys/sysctl.h Define the virtualisation-related sysctls.

darbat/igcompat/machvm/pmap.c Added a non-panicking stub for pmap_remove_some_phys().

darbat/igcompat/machvm/vm_resident.c Don't immediately panic in vm_page_wait().

darbat/igcompat/src/console.c Don't define cnputcusr here when using vconsole.

darbat/igcompat/src/darbat.c Virtualisation-related sysctl initialisation code, and allocation of the I/O buffer mapping table. Send darbat_cons_write characters to vconsole.

darbat/igcompat/src/ig_stubs.c Removed or conditionalised the stubs for symbols that I added implementations of.

darbat/igcompat/src/iokit.c Enable setup of multiple disks when using vblock, explicitly send capabilities for shared memsections to IOKit, separate the network, disk and userclient proxy memsections when virtualised, send messages to IOKit via vdev when it's there.

darbat/igcompat/src/iokitstubs.c Changed the communication protocol with IOKit to send the address of the return area so multiple clients can work; also saved a message register by making use of the label.

darbat/igcompat/src/iokit/disk.c Changes to stay compatible with diskdev.c and IOKit when their interfaces changed.

darbat/igcompat/src/iokit/diskdev.c Added async I/O support, implemented sync ioctl, turned writability on.

darbat/igcompat/src/main.c Don't initialise video console when using vconsole, initialise virtualisation-related sysctls, only insert thread ID into naming when not virtualised.

darbat/igcompat/src/sysloop.c Added mapcap syscall, made pager function handle mappings set up with mapcap. Handle disk I/O completion messages (CT_REQ_COMPLETE_IO).

darbat/osfmk/console/i386/serial_console.c Send console output to vconsole.

darbat/osfmk/i386/cpu_data.h Fixed a syntax error in the CPU_DATA_GET macro.

darbat/osfmk/kern/clock.c Silenced a printf in clock_deadline_for_periodic_event that was happening every 100 ms.

darbat/osfmk/l4/pcb.c Add non-panicking stubs for a couple of functions called by the paging code.

darbat/osfmk/mach/i386/vm_param.h Halved VM_KERNEL_SIZE so more than one Darbat could run.

darbat/pexpert/gen/pe_gen.c Don't initialise the built-in video console when using vconsole.

iguana/init/src/startup.lua.template Start the virtualisation servers, and don't automatically start Darbat when using virtualisation.

iguana/init/SConstruct Compile new source file.

iguana/init/src/init.c Removed the hack where all tasks were given the capability for the memsection shared between IOKit and Darbat.

iguana/iokit/SConstruct Added virtualisation defines.

iguana/iokit/src/iokit.c Made IOKit act as a vblock provider. Added a thread to do disk I/O asynchronously, with requests added by the main thread to a producer/consumer queue. Added the ability to receive new capabilities. Remove the assumptions of a single client with a single shared memsection. Added the disk sync operation. Intercept some special keyboard characters and send them to the supervisor. Fixed some bugs.

libs/xnuglue/include/iguana/iokit_disk.h Added new disk op defines.

libs/xnuglue/src/l4compat/hacks.c Adjusted the offset of the return area for the virtualised case in vm_map_copyin_common().

tools/build.py Added the KengeEnvironment.RenamedApplication() method so that multiple, differently-named Darbats can be built from the same source.