L4/Darwin: Evolving UNIX

Geoffrey Lee and Charles Gray
National ICT Australia, Sydney, Australia

Abstract

UNIX has remained a mainstay of modern computing. With its foundations of security, reliability, performance and configurability, UNIX has adapted to and is used in a vast array of environments.

While UNIX fosters robustness, modularity and a "smaller is better" philosophy, that scrutiny is generally not applied to the kernel itself. Modern UNIX kernels have large, unwieldy code bases that do not enjoy the benefits seen in the user environment.

Apple's Darwin kernel is the open-source core of the Mac OS X operating system. Like most modern UNIX systems, the kernel boasts modern features such as 64-bit address spaces, robust hot-plug and support for server and workstation workloads.

L4/Darwin (Darbat), a virtualised Darwin system running on the L4 microkernel, aims to address the problem of the ever-growing UNIX kernel. Using the high-performance L4 microkernel, Darbat can isolate kernel modules, such as device drivers, using hardware protection while maintaining binary compatibility and performance. This modularisation also allows Darbat to use L4 as an advanced hypervisor to support multiple operating system instances for server consolidation.

This paper covers the on-going design and implementation of the Darbat project and the experiences of bringing the strengths of UNIX into the UNIX kernel itself.

1 Introduction

Since its inception at Bell Labs in 1969, UNIX has evolved and grown and has proven to withstand the test of time. Today, UNIX systems are as relevant as ever.

UNIX fosters the creation of small, specific tools, providing powerful mechanisms for joining them together to create more complete and powerful applications. UNIX sockets and pipes are the back-bone of data processing.

UNIX is also robust, in that programs are all isolated from each other. A failure in one program can at worst cause that program to crash, and will not cause the entire computer system to misbehave as a whole.

It is quite interesting to note that these properties of UNIX do not generally apply to the kernel.

Modern UNIX kernels support vastly more features than their ancestors. These features come at the cost of increased kernel size and complexity. As an example, the full Linux kernel tree currently contains millions of lines of code for all drivers and components.

Even though all this code can never be running at once, such large, modular systems have a number of implications. Modules need to be selected, configured and tested to work together. Reconfiguring and restarting an operating system kernel, however, has far more implications than restarting an application service since it typically requires a costly full system reboot. In particular, having to restart a user application is a localised nuisance, however, restarting the kernel will result in downtime for all the user applications that were running on top of it.

System configuration can also be a complicated process. Compile-time options such as resource limits or locking behaviour dictate a performance/functionality trade-off. Unforeseen changes in workload or minor hardware expansions may require costly kernel rebuilds and testing if the original kernel configuration did not account for these changes. The alternative, to conservatively configure your kernel for future workloads will most likely lead to a reduction in peak performance.

Bugs and incompatibilities are also problematic for inflexible kernels. Some drivers or modules may require a specific kernel version or set of sub-optimal global configuration options. Modules may also have bugs which cause problems on SMP configurations or incompatibilities with other modules because it is not possible to ever test all possible configurations.

A common approach to solving this problem is to use virtualisation. VMWare [VMW], Xen [BDF+03] and Parallels [Par] are hypervisors that allow execution of multiple operating systems concurrently on the one computer. Running multiple operating systems side-by-side provides greater flexibility in system composition. Each kernel can be configured to suit the workload of applications which it will serve. A related approach taken by LeVasseur et al. [LUC+05] loads operating system instances inside virtual machines to utilise their device driver functionality.

The virtualised operating system approach goes a long way to improving the UNIX kernel, however it is a very coarse-grained approach to running multiple instances of whole operating systems. In this paper we discuss L4/Darwin (Darbat). Darbat provides a more flexible approach to virtualisation which promotes the "small

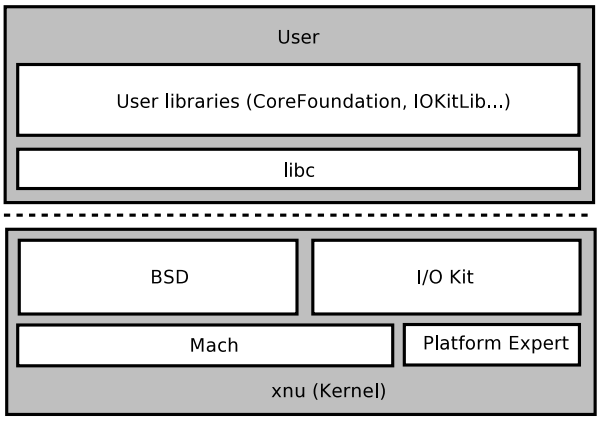

Figure 1: The Darwin system.

is beautiful" philosophy by decomposing the operating system itself. We use the Darwin kernel to demonstrate operating system modularisation on top of the L4 microkernel.

2 Darwin and Mac OS X

Darwin is the UNIX-based open-source core of the Mac OS X operating system. It is a collection of a large number of open-source projects, which are freely available from Apple's open-source website.

The Darwin system has a very long history. Its predecessor, known as NEXTSTEP, was originally conceived at NeXT in the 1980s. When NeXT was acquired by Apple in 1997, the NEXTSTEP operating system became the basis for Apple's replacement to its aging Mac OS operating system. This would come to be known as Mac OS X, pronounced "Mac OS Ten". In 2000, Apple released the source code for the UNIX foundations of Mac OS X under an open-source license [App00].

Figure 1 shows a block diagram of the Darwin system. At the heart of Darwin is xnu. Despite its status as a single project, the xnu project actually contains several separate subsystems. As the kernel, xnu executes on the CPU in privileged mode giving it full control of the system.

Xnu is divided into three major components, the Mach microkernel, the BSD UNIX layer and the I/O Kit device driver framework. While these are logically distinct modules, they are co-located in the kernel address space and invoke each other via regular C function call interfaces.

The kernel also contains other minor modules. The *Platform Expert*, as the name suggests, is responsible for platform services such as boot arguments. There is also a kernel linker for patching in *kernel extensions* (KEXTs) such as drivers at run-time.

2.1 Mach

Mach began as an operating systems research project at Carnegie Mellon University in 1985 [RJO+89]. Mach is a first-generation microkernel, designed to provide flexible abstractions such as *inter-process communication* (IPC) and *virtual memory* (VM) services. Using these kernel services it is possible to build a full operating system as a set of user processes.

The Mach subsystem as shipped with the xnu project is derived from Mach 3.0 and has undergone extensive improvements made by Apple. Features such as performance enhancements, a unified buffer cache and 64-bit support have been added.

Mach is the arbiter of the whole operating system. It provides the low-level exception handlers, threads and run-queue management as well as page-tables, virtual memory objects and address spaces. All system operations, be they in the I/O Kit, BSD or user applications, are managed at the lowest level by Mach.

2.2 BSD

The xnu kernel also contains a BSD subsystem. It implements features that one would expect to find in a any modern UNIX kernel, such as filesystems, a TCP/IP stack, UNIX processes and signals. The kernel BSD implementation is derived from 4.4BSD, although a significant portion of the code base has been modernised with ideas and code from FreeBSD, NetBSD and OpenBSD.

Migrating BSD to run as a module on top of the Mach subsystem creates a lot of duplicated code in the form of trap handlers, VM systems and thread and process management. The xnu BSD has been stripped of this duplication and uses the underlying Mach functionality instead. The BSD layer provides just the high-level UNIX functionality.

2.3 System Calls

In Darwin, kernel requests can be made in three ways: BSD traps, Mach traps, and Mach *remote procedure calls* (RPC). Mach has very few system-call entry points with most Mach operations implemented as Mach RPC, built on top of the Mach IPC infrastructure. This includes such operations as VM, thread or task manipulations.

System-call traps are identified by a system-call number argument which is checked on kernel entry. The system-call number indexes into either the Mach or BSD system-call tables, depending on the sign of the number.

In the case of Mach services exported via an RPC interface, the arguments to the call are packed into a Mach IPC message. The `mach_msg()` Mach system-call is invoked and the message is then dispatched to the correct Mach service. While it is technically possible to manually pack the arguments into a Mach message

and invoke the `mach_msg()` system-call directly, this is rarely done. Instead, such interfaces are usually defined using the *Mach interface generator* (MIG) [DJT89]. MIG is used to generate the appropriate source-code and header stubs for both the server side and the client side.

2.4 I/O Kit

The I/O Kit is the object-oriented device driver subsystem of Darwin, written in C++. In order to make the language suitable for use in the kernel environment, only a small subset of its features are used. Exceptions, multiple inheritance and templates are not used in the I/O Kit nor in I/O Kit drivers. The I/O Kit also uses a *runtime type information* (RTTI) system which is provided by the `libkern` library rather than the default C++ RTTI.

I/O Kit supports many of the features that one would expect to find in a driver framework of any modern general-purpose operating system, such as device hot-plugging, power management, dynamic loading and unloading of driver code and automatic device configuration. Despite the fact that the I/O Kit is written from scratch, it is heavily based on the Driver Kit, the driver framework found in NEXTSTEP 3.x.

The I/O Kit adopts a modular, layered approach that captures the relationships between components in the system. These components can be split into two types of objects: *drivers* and *nubs*. A driver is an object instance that knows how to drive a particular hardware device. An example of this would be a disk controller driver object. This particular object would know how to drive a specific model of disk controller.

A nub object is a connect point between two driver objects. When the I/O stack is built, nubs are created by drivers to represent an attachment point for other drivers. For example, the PCI bridge driver exports nubs which act as connection points for drivers to connect to drive the various devices that sit on the PCI bus. Figure 2 illustrates an example of the driver-nub relationship. For the USB case, it additionally shows the layout of a driver stack.

The I/O Kit is described in detail in the I/O Kit Fundamentals document which is available for download on Apple's website [App04].

3 Darbat

Microkernels were touted to offer a framework for better security, increased flexibility and increased robustness. The microkernel paradigm was to build systems in small component blocks isolated from each other, hence limiting malice and the amount of of damage that any one component can cause, while providing better flexibility of how a particular system should be laid out. However, these goals were never realised out of first generation microkernels such as Mach. Performance problems

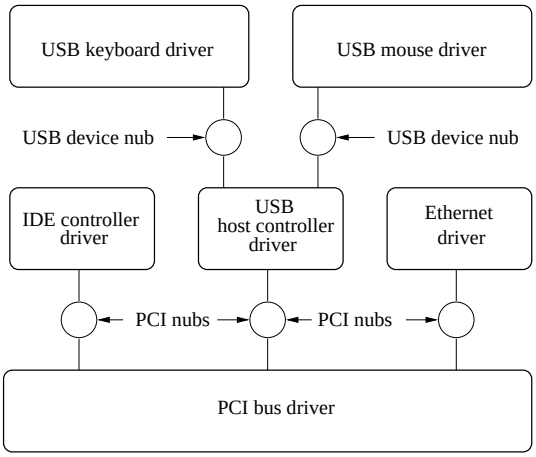

Figure 2: An example of a driver and nub relationship.

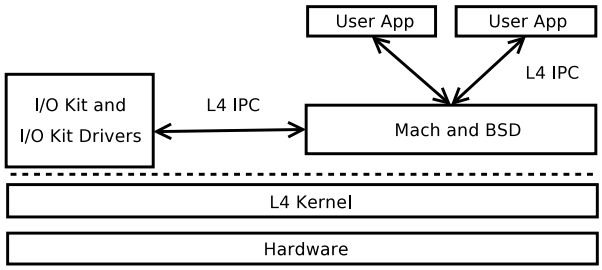

Figure 3: The Darbat architecture.

meant that system designers eventually started to move these system components into the protection domain of the kernel. Indeed, the xnu kernel itself cannot be described as a microkernel-based system as all these separate subsystems run within a single protection domain. This means that they inherit the overhead of running on top of a slow microkernel, with none of the benefits that the microkernel was supposed to provide.

L4/Darwin is a port of Darwin to run on top of the L4 microkernel as a para-virtualised guest operating system. Para-virtualisation is a widely-used technique of virtualisation that involves replacing privileged operations with calls to a hypervisor. Although this technique involves significant engineering cost compared to other methods such as pre-virtualisation as described by LeVasseur et al. [LUC+05], it allows for significant flexibility in the design scope for the L4/Darwin system.

3.1 Darbat Design

The Darbat design is shown in Figure 3. Darbat currently para-virtualises the Darwin kernel into two modules, the Mach and BSD kernel component (L4/xnu) and the I/O Kit. These modules are then run as de-privileged programs on top of the L4 microkernel.

Since xnu and I/O Kit are independent processes, it

is possible to start and stop them as easily as a regular UNIX application. This is the foundation of Darbat's flexibility.

3.2 L4

Darbat uses the L4 microkernel [Lie95] as an advanced hypervisor. L4 provides fast, light-weight mechanisms which allows building operating systems with secure, isolated modules. Isolation ensures that a failure in a single user-land module does not inherently imply the failure of the whole system.

L4 is a high-performance, industrial-strength second-generation microkernel. The L4 design recognises that in order to be fast and yet secure, it must implement the bare minimum required on top of the hardware.

The philosophy behind L4 is different to that of Mach. The Mach microkernel approach is to generalize abstractions to suit all possible uses. L4 does not provide generalised, heavy-weight abstractions but instead acknowledges how the underlying hardware works..

The overly generalised Mach model leads to an implementation that is by design slow and cumbersome. An IPC operation on Mach takes hundreds of thousands of processor cycles to complete, while on L4, IPC performance dominated by hardware costs.

Like Mach, L4 provides address spaces, threads and IPC via message-passing. The VM and IPC models in L4 are very different to that of Mach. Mach provides a *memory object* abstraction which represents a region of VM. This object can then be mapped and shared between address spaces, potentially with *copy-on-write* (COW) semantics. The structure of whole or partial address spaces can also be shared.

In contrast, L4 simply allows the construction of address spaces using power-of-two sized pages. This keeps the kernel small and fast and leaves the complicated data structures to be managed by operating systems at user-level.

The IPC interface in L4 is similarly minimal. Instead of complicated message data structures, L4 only allows a small number of machine words to be delivered in a single IPC operation. On most architectures the average message (around 8 words) can be delivered entirely in CPU registers without having to copy memory. L4 provides only synchronous data delivery without kernel buffering to eliminate kernel buffers and ensure CPU throughput. Using this fast, secure abstraction, other IPC models can be layered on top with shared memory for bulk data transfers. This provides excellent best (and often common) case performance with negligible degradation in the worst case. While providing impressive direct costs for kernel operations, the L4 design also optimises the hidden costs of cache and TLB footprint.

The version of L4 used in Darbat is NICTA::Pistachio-embedded [NIC]. Darbat is built using the L4/Iguana framework. The NICTA::Pistachio-embedded kernel is a derivative of the L4Ka::Pistachio kernel, which is developed jointly by the University of Karlsruhe and University of New South Wales [L4K]. The NICTA::Pistachio-embedded kernel is under active development by National ICT Australia.

3.3 Principle of Least Authority

Modern UNIX kernels are large and contain millions of lines of source code, and they run all this code with full supervisory privileges on the processor. Only a fraction of the code actually requires this privilege, however. Darbat applies the *principle of least authority* (POLA). This is the same principle employed by many UNIX `setuid` applications which drop root privileges once they have acquired the necessary resources to reduce the damage of an exploit. To this end, we have modified the xnu components to run completely de-privileged, using L4 as a proxy for privileged operations as necessary. Compromising a single operating system instance does not allow a hacker to compromise the device drivers or the rest of the machine.

3.4 L4/xnu

L4/xnu is the Mach and BSD components of xnu paravirtualised to run on top of L4. L4/xnu uses L4 IPC and shared memory to communicate with I/O Kit drivers instead of being co-located in the one protection domain.

Since Mach is the lowest level of Darwin, porting xnu to L4 required changing all the low-level bindings. L4/xnu does this by creating a new L4 architecture for Mach, alongside i386 and PPC. This allows L4/xnu to hook into exception handling, memory management, threading, system-calls and the low-level hardware routines. L4/xnu needs surprisingly few changes to the generic code in xnu.

One of the major changes in L4/xnu is the system-call handling mechanism. Instead of standard trap instructions, user applications running on L4/xnu use L4 IPC to invoke system-calls in its instance of the kernel. The L4 message contains the system-call number and arguments normally found on the stack. This is a performance optimisation to remove expensive `copyin` operations.

L4/xnu is free to completely change the system-call mechanism yet remain compatible because Darwin defines a *library* interface rather than a *system-call* interface. Darbat simply uses a slightly modified `libc` library and remains binary compatible with existing Darwin applications.

While most low-level abstractions in L4/xnu have been modified to use the L4 equivalent, one thing which has remained the same is the threading structure. Xnu is a multi-threaded kernel, however unlike user tasks,

L4/xnu multiplexes all kernel threads onto a single L4 thread. This is necessary because xnu contains large amount of legacy code, some of which disables all CPU interrupts to enforce critical sections. For security reasons L4 cannot let any application turn off interrupts. Instead L4/xnu thread switches itself in a single L4 thread with the timer and other interrupt sources generating exceptions. This allows L4/xnu to efficiently emulate interrupt masking and requires no changes to legacy Darwin code.

3.5 L4 I/O Kit

The L4 I/O Kit is a port of the Darwin device driver framework that runs in user mode [Lee05]. The L4 I/O Kit runs as a completely separate service from Darbat and does not depend on its presence.

The main design decisions made in porting the I/O Kit were as follows. First was the early realisation that that the L4-based I/O Kit would have to accommodate binary-only drivers. It was decided that no changes should be made to the publicly exported I/O Kit interface.

Second, it was decided that the I/O Kit should run as a separate service and could serve as a basis for much of the flexibility that we hope to achieve in our system. At the same time, we realized that the internals of the I/O Kit was dependent on Mach for common operating system services such as thread management or synchronisation primitives. In addition, there are some commonly used features of the Mach kernel such as *thread callouts* that many drivers call directly because there is no equivalent in the I/O Kit.

These Mach-based routines which the I/O Kit and drivers depend on are provided via a thin Mach compatibility library called `xnuglue`. This layer provides the necessary OS primitives such as locks, semaphores and threads using L4 abstractions. In particular, effort has been made so that they are ABI compatible with their xnu counterparts. This allowed us to completely separate the I/O Kit into a separate service, while sparing us the effort of having to modify the I/O Kit's underlying Mach-dependent implementation, as well as providing commonly used xnu functions for other I/O Kit modules. Unlike the Darbat kernel, `xnuglue` is fully multithreaded. `Xnuglue` "Mach" threads are emulated as native L4 threads. The actual code in `xnuglue` is approximately 4000 lines of C source code, with a very tiny fraction of assembly code.

3.6 Kernel Composition

Separating the kernel and device drivers adds a lot of flexibility which allows Darbat to solve a number of interesting problems. For example, Darbat can be used to provide standard server consolidation features. The administrator can configure the system to start a single I/O Kit instance to drive the hardware as well as a number of xnu instances which present a standard Darwin system. Each xnu instance can have a different root user, a different setup and access to different physical devices.

In general, Darbat makes the administrator free to create various operating system and driver module instances, assign resources, and hook them up in interesting ways. This may include physical partitioning of the CPUs, RAM and device between different users.

Darbat can also be used to isolate components for security. The system could be configured to run a trusted kernel for sensitive applications and a general-purpose kernel for other services. Kernel migration can be eased with applications transitioned to a new kernel version one-by-one without having to cut the whole system over in one go.

It is well-documented that while drivers account for the majority of OS code, they are error-prone [CYC+01]. A setup similar to LeVasseur et al. [LUSG04] can be used to isolate these errors and recover from them. Under this scheme, multiple I/O Kits can be run alongside each other, each with its own set of drivers. A failure of one driver in one I/O Kit instance can at worst only crash that instance of the I/O Kit, hence greatly improving fault isolation and system dependability.

L4 I/O Kit provides a self-contained driver module without any dependency on the rest of the xnu system. Darbat therefore allows other virtualised operating systems to reuse I/O Kit driver code. Before L4/xnu was stable enough to be used, Wombat [LvSH05], a paravirtualised version of Linux, was routinely used for testing and benchmarking I/O Kit drivers. Ultimately we would like to be able to support Darwin and Wombat running side-by-side with heterogeneous applications sharing the same devices and file systems.

Darbat can also be used to accommodate some interesting trusted applications. Remote management consoles, watchdogs and other low-level utilities are relatively straight-forward to implement in software, however their ability is somewhat limited if they cease to function when the kernel locks up or panics. For this reason such services usually require special hardware and are only available on high-end machines. Under Darbat these services can be built as reliable software solutions. By executing isolated from the main kernel they need not be dependent on the UNIX kernel or other device drivers.

At the lowest level, all L4 processes use IPC messages and shared memory to get their work done. In many respects, this is analogous to how UNIX applications are built up using pipes and files. L4 provides the mechanisms allowing messages to be filtered, modified,

injected or logged. This will provide a powerful tool for administrators, giving them greater dynamic control over the kernel.

Having multiple I/O Kit instances can also be a boon for compatibility. For example, it would allow for the co-existence of both 32-bit and 64-bit I/O Kit drivers, running in a 32-bit instance and a 64-bit instance of the I/O Kit respectively. In addition to this, this setup could be used to solve software compatibility issues. Whenever a kernel interface change is made, one must be careful not to break compatibility with existing users of the interface. This often means implementing work-arounds in the code, giving rise to warts in the implementation of the software in the name of compatibility. In the Darbat system, supporting an older driver is as simple as spawning a separate I/O Kit instance that supports the driver.

3.7 Ease of Debugging

Developing systems code can often be a frustrating and painful process. Unlike user code, which can at worst bring down the buggy program itself, a fault in systems code will usually bring down the whole system. This fact has led to complicated schemes such as debuggers that try to inspect the state of the system with what limited information can be gathered from a system that is by definition in an inconsistent state, and repeating the "crash, debug, fix, rebuild" cycle. With an isolated driver framework, this no longer has to be the case. The code can be tested and fixed on the local machine, and the driver can then be restarted with a new, fixed copy.

3.8 Performance Considerations

Despite the potential advantages that a microkernel-based design would bring, it should not be forgotten that performance considerations in commodity UNIX systems are just as important.

One of the common arguments against a microkernel-based design is that such a design will perform much more work than a monolithic-based design, in the form of context switches due to IPC. Context switches cost processor cycles, and have other adverse effects such as cache pollution. Employing proper microkernel design by laying out the system in such a way that IPC operations are used sparingly is the key to unlocking the performance and power of a microkernel such as L4.

In order to show that IPC operations do not necessarily have to be costly and slow, we modified the system-call path in Darbat to use an L4 IPC-based implementation that has only undergone slight optimisations. Figure 4 illustrates the actions that would be done when a user applications invokes the read(2) UNIX system-call in a Darbat system. Even though in theory the L4-based system has to perform more work, the cost of the L4 IPC-based implementation is similar to the trap-based

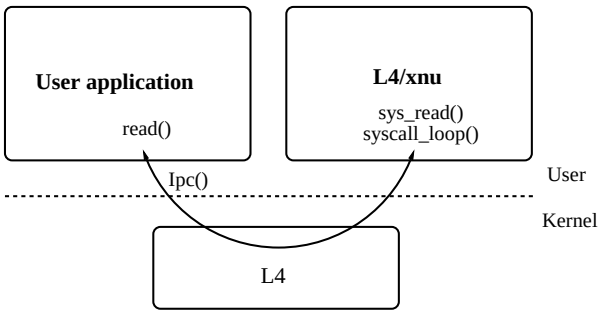

Figure 4: The Darbat system-call path.

implementation found in native xnu.

One reason for this is that Darwin provides the full 32-bit address space to applications. In this case both Darwin and Darbat need to perform a full address space switch to execute the system-call. L4 benefits from its highly optimised context-switch code and fewer `copyin` operations due to the use of the IPC payload.

4 Experiences

4.1 Xnu

An initial Darbat design goal was to remove Mach completely. However, we have found that this is not feasible. Generally speaking, the Darwin system is thoroughly dependent on the Mach infrastructure. Instead, we have chosen to co-locate Mach and the BSD subsystems together and slot L4 underneath them. Originally, we only brought in a subset of Mach. Over time, as we brought the system up more and more up to a usable state, we discovered that we have essentially pulled in almost all of the Mach code base.

Darbat maintains compatibility with existing Darwin applications, even to the extent that we can run unmodified programs as shipped with Mac OS X for Intel. Keeping the Mach code base has allowed us to quickly get L4/Darwin into a usable state, while still demonstrating the advantages of L4.

4.2 L4 I/O Kit

One lesson learned from porting the I/O Kit to L4 was that despite the obvious challenges of trying to isolate a driver framework taken from a monolithic operating system such as Darwin, as long as there are well-defined interfaces, it is possible to do so. It was possible to get L4 I/O Kit up to a benchmarkable state over the course of a couple of months. In contrast, we have made multiple attempts to isolate Linux device drivers in a para-virtualised Linux driver subsystem environment, with little success.

With the release of Intel-based Apple hardware that required drivers that were were only available in binary form, we were forced to make a difficult decision: wait

for the source to become available, write drivers from scratch for the devices on the hardware, some of which have poor to nonexistent documentation, or load the binary drivers provided by Apple. We were pleasantly surprised how little effort was required for the latter. We have managed to load a fairly large set of device drivers, some of which we heavily depend on. The drivers themselves are unaware they are running outside a vanilla xnu system and in user-mode. The consistent interface provided by the I/O Kit provides the guaranteed ABI that allows binary drivers to remain portable across multiple releases of an operating system and vastly different runtime environments.

Despite the enormous flexibility that this gives us, the `xnuglue` layer means that only very minimal changes had to be made to the I/O Kit. All but two of the changes were compile-time changes due to slightly different headers supplied by the xnu source tree and our own version of them.

4.3 Performance

Conventional wisdom is that Mac OS X is slow, as a direct result of Darwin, and specifically the use of Mach. Upon beginning this project we were fairly confident that it would be easy to demonstrate drastic speed-ups with the shift to a much faster microkernel. Conventional wisdom seems to be wrong, however.

Generally speaking, xnu is well optimised. One main reason for deciding to pull in Mach instead of writing a compatibility layer is that re-writing Mach features sufficiently to support existing applications would almost certainly result in a bigger, slower, buggier version of Mach. Our performance enhancements will be demonstrated by optimising around Mach, not Mach itself. While this may not be as easy as first thought, there are system-wide speed-ups to be made in IPC and thread operations.

4.4 Portability

The L4 variant used in Darbat, Pistachio, was explicitly designed with portability in mind. There are ports to a large number of architectures including IA-32, x86_64, IA-64, MIPS, Alpha, ARM and PowerPC to name a few. Because Darbat uses L4 predominantly as a hardware abstraction layer, this means that Darbat on L4 is relatively easy to port to other architectures supported by L4. Our experiments with Darbat components on other platforms show promising results, and with much less effort than would be required to make a native port.

4.5 Module Independence

Although splitting the I/O Kit and the rest of xnu into stand-alone modules initially took substantial effort, we have found it to be an extremely useful feature. During development it has been very useful for debugging to be able to boot the system with xnu or the I/O Kit disabled to test each module in isolation. This allowed us to quickly isolate bugs to either module, or the interface between them. Once it is known which module a bug is in, and a few vague symptoms, it can generally be reduced to a subclass of bug which makes it easier to track down. This greatly reduces the amount of time it takes to hunt down and fix even the nastiest kernel bugs.

Occasionally during development, however, and especially when bringing up new interfaces, some temporary dependencies between the two modules may creep in. Invariably this causes more problems than the "quick fix" was trying to solve, typically leading to one or both modules randomly failing on startup. Usually, when implemented, the correct solution turns out to be neater and far more reliable.

Splitting the I/O Kit and the rest of xnu is not simply a matter of initially decomposing the system to run independently, as interfacing with the separate modules now involves more work than simply making a direct function call. One instance of this which we caught relatively early was the glue between BSD and the I/O Kit. In native Darwin, disks are exported to user applications using the traditional BSD-style block (`bdevsw`) and character device (`cdevsw`) switches. User applications that indirectly or directly access these devices eventually end up in the BSD client, where it is then possible to call into the I/O Kit class hierarchy to perform the operation.

This code is problematic because this glue layer deals with BSD data structures which are not a part of the I/O Kit class structure. Bringing up an interface usually meant devising a rather complicated protocol to communicate between the I/O Kit and xnu and replicating some BSD data structures.

Using the C++ class hierarchy to our advantage we can actually move the very top of I/O Kit into xnu. This will allow us to further free the I/O Kit driver stack of duplicated BSD dependencies, as well as provide a much cleaner interface between the I/O Kit and external modules such as xnu or Wombat.

4.6 Minimise Changes

Another hard-learned lesson from Darbat is to avoid taking short-cuts in the name of expediency and short-term convenience. It is tempting to think that a feature could simply be emulated by implementing a naive version with the required interface. However, as the project progresses and more subsystems are brought up, it can cause problems due to missing features or incorrect semantics in the emulation.

Some specific examples of this are trying to short-circuit the bootup path, trying to remove Mach, using an ELF-based development toolchain instead of the na-

tive Mach-O toolchain and trying to modify the Mach scheduler without fully understanding it.

5 Related Work

Virtualisation, and especially para-virtualisation are well known techniques. There is an abundance of work in this area. In this section we discuss a few very particular systems, however there is a lot more knowledge than would fit in this paper.

5.1 Linux on L4

There have been various attempts at porting a monolithic operating system system to run as a guest on top of L4. One such attempt has been done by Härtig et al. [HHL+97]. In their work, they modified the Linux kernel to run on top of L4. More recently, Leslie et al. [LvSH05] ported version 2.6 of the Linux kernel to run as a para-virtualised guest on top of L4. TU-Dresden also has a port of the 2.6 series of the Linux kernel to their implementation of the L4 microkernel [Tec].

5.2 Mach-based Operating Systems

Various attempts have been made to port UNIX to the Mach microkernel, such as the work done by Helander [Hel94] and Golub et al. [GDFR90], who ported UNIX to run as a guest operating system on Mach. Darwin itself is a UNIX on top of the Mach microkernel, although nowadays it is virtually devoid of any features of a microkernel-based design. It should be noted, however, that these efforts usually ported the entire kernel, including device drivers, to run as a separate process on top of the microkernel.

5.3 Multiserver Operating Systems

The multiserver approach to building operating system revolves around designing the system in such a way that subsystems are separated into isolated components. Figure 5 illustrates the design of a multiserver system. One of the most well-known multiserver OS with UNIX compatibility is MINIX. The latest release of MINIX, version 3, boasts that it is targeted at real-world embedded applications which makes it different from being an academic curiosity that previous versions have frequently been accused of. In the MINIX system, subsystems such as the memory management subsystem, the file subsystem and various drivers would all reside in separate protection domains.

However, refactoring existing operating systems to such an extent will incur significant engineering costs. It is possible to design a system from scratch, but that requires significant effort and it may prove difficult to leverage the work that has already been done on existing systems. The SawMill Linux project [GJP+00] is one example of an existing system being re-factored into a multiserver-based operating system.

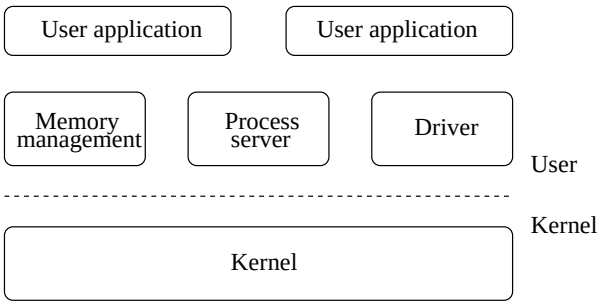

Figure 5: A multiserver operating system.

5.4 Full Virtualisation

Full virtualisation allows for the execution of multiple guest operating systems on top of a host kernel or hypervisor with no modifications required to the guest. It is particularly suitable to cases where the guest cannot be modified. Full virtualisation involves rewriting and emulating sensitive instruction at run-time in the hypervisor.

While such a virtualisation technique would allow for many different type of guests to run without any modifications, it also leads to sub-optimal performance since you cannot batch hypervisor calls.

Parallels [Par] and VMWare [VMW] are examples of products that implement full virtualisation.

5.5 Para-virtualisation

Para-virtualisation recognizes that performance using full virtualisation is sub-optimal, and addresses this by modifying the guest operating system to be hypervisor-aware. Para-virtualisation involves porting the guest operating system to an interface specified by the hypervisor. The hypervisor in para-virtualisation is a specialised kernel that runs with supervisory privileges that handles privileged calls on the guests' behalf. Performance-wise, para-virtualisation performs better than full virtualisation, however, it incurs the cost of porting the guest to the hypervisor API, as well as being only applicable to operating systems where the source is available, or where a vendor is willing to port the operating system to the hypervisor API. Xen [BDF+03] is a very well-known example of a hypervisor that utilises the para-virtualisation technique.

While typical hypervisors such as Xen and L4 share a lot in common in terms of functionality, there are also subtle differences. Xen provides an abstraction of the machine hardware. This means that an operating system is ported to run on top of Xen for a particular CPU type, and porting to another architecture implies another port of Xen.

L4, however, provides are more general abstraction providing threads, messages and paged address spaces with little overhead. This allows us to move modules running L4 to other L4 architectures with very little effort. This also means that isolated modules on top of L4 are cheaper since they need not worry about maintaining their own threads or address space since it is already provided.

6 Current Status and Future Work

Currently, the L4/Darwin system is very much a work in progress. However, the system is already able to boot into into single user mode and can run a vast set of unmodified command-line tools. On the driver side, since the release of production Intel-based Apple hardware, the I/O Kit has been made to work with various binary drivers as shipped by Apple without modification. Although many drivers have not been tested and some will require further work for correct operation, we routinely use various Apple-supplied drivers for our day-to-day testing and development work, including disk controller drivers and USB controller drivers. Other types of drivers, including graphics drivers, ethernet drivers and sound drivers are known to load correctly but have not been extensively tested.

The L4/Darwin project is already able to partially act as a drop-in replacement for the original Darwin kernel. L4/Darwin boots to single user mode on a standard Mac OS X install by simply replacing three files, namely the kernel, the dynamic linker and libc.

Darbat is currently under heavy active development and various features are planned for the next few releases. This includes full para-virtualisation of both the Mach/BSD portion and the I/O Kit, demonstration of restartable device drivers by running multiple I/O Kit instances, and more complete interfacing support that would allow us to run a full Mac OS X system on L4/Darwin.

Although we have found we cannot simply replace Mach-based IPC with L4 IPC in general, we also plan to identify any expensive IPC messages for specific optimisation. This may involve techniques such as modifying the MIG code generator or run-time detection of suitable messages.

7 Conclusions

The UNIX "small is beautiful" philosophy does not generally extend into the kernel itself. This has left the UNIX kernel with a large, unwieldy and inflexible code base.

In this paper we have presented Darbat, a modularisation of the Darwin kernel. Darbat partitions the existing kernel into driver and kernel modules, each of which run as de-privileged processes for fault isolation. This allows Darbat to bring the flexibility and robustness of a user environment to the kernel itself. A flexible kernel design will ultimately lead to modular code providing standard virtualisation features.

The Darwin kernel has proven to be an excellent candidate for this work. The already modular nature of the kernel has made it relatively straight forward to componentise. Even with drastic changes to the runtime environment, however, Darbat maintains binary compatibility with user-mode applications and kernel-mode drivers.

At the cost of reliability, the Darwin kernel co-locates kernel modules with Mach for performance. Our experiments using L4 have so far shown that performance is a problem with the Mach design and not microkernels in general.

Although a work in progress, our result so far show that the Darbat approach provides flexibility in kernel composition while maintaining competitive performance and binary compatibility with native systems.

7.1 Code availability

The code and documentation for the L4/Darwin project is available at `http://www.ertos.nicta.com.au/software/darbat/`. The current release at the time of writing is 0.2, which is based on the Darwin 8.2 source code.

References

[App00] Apple releases Darwin 1.0 open source. `http://www.apple.com/pr/library/2000/apr/05darwin.html`, 2000.

[App04] Apple Computer Inc. *Introduction to I/O Kit Fundamentals*, 2004.

[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.

[CYC+01] Andy Chou, Jun-Feng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on OS Principles*, pages 73–88, Lake Louise, Alta, Canada, October 2001.

[DJT89] Richard P. Draves, Michael B. Jones, and Mary R. Thompson. MIG - the MACH interface generator, 1989.

[GDFR90] David B. Golub, Randall W. Dean, Alessandro Forin, and Richard F. Rashid. UNIX as an application program. In *USENIX Summer*, pages 87–95, 1990.

[GJP+00] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J. Tidswell, L. Deller, and L. Reuther. The sawmill multiserver approach, 2000.

[Hel94] Johannes Helander. Unix under Mach: The Lites server. Master's thesis, Helsinki University of Technology, 1994.

[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.

[L4K] L4Ka Team. L4Ka::Pistachio kernel. `http://l4ka.org/projects/pistachio/`.

[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.

[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.

[NIC] NICTA::Pistachio-embedded kernel. `http://www.ertos.nicta.com.au/software/kenge/pistachio/latest/`.

[Par] Parallels, Inc. Parallels. `http://www.parallels.com/`.

[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.

[Tec] Technische Universität Dresden. L^{4}Linux. `http://os.inf.tu-dresden.de/L4/LinuxOnL4/`.

[VMW] VMWare, Inc. VMWare. `http://www.vmware.com/`.