THE UNIVERSITY OF NEW SOUTH WALES
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Native OKL4 Android Stack

Michael Hills

Bachelor of Engineering (Computer Engineering)

Supervisor: Gernot Heiser

Co-supervisor: Nicholas FitzRoy-Dale

Assessor: Kevin Elphinstone

November 2, 2009

Abstract

Embedded devices such as mobile phones are offering increasingly more powerful environments to users, but the hardware is still characterised by low-power processors, limited system memory, and less than optimal battery life. Second generation microkernels like OKL4 optimise for embedded hardware and offer high-performance with a minimal memory footprint.

The Android mobile phone operating system currently uses Linux as its kernel and offers a Java runtime environment using the Dalvik virtual machine. OKL4 offers improvements that could benefit the performance of Android. OKL4 is small, sports a high-performance IPC mechanism, provides support for superpages, and provides the flexibility of being able to build a minimal operating system environment.

In this thesis, we port various parts of the Android operating system to run Android applications on the OKL4 microkernel. Using the Android Developer Phone 1, system comparisons are made between Android running on OKL4 and Android running on Linux. We conclude that OKL4's fast IPC mechanism and superpage support can provide performance improvements to Android, but measured gains are mitigated by the poor performance of the Dalvik virtual machine.

Acknowledgements

I would especially like to thank Nicholas FitzRoy-Dale, my thesis would not have been possible without his platform port to the ADP1. Most of all he was always available to provide guidance throughout the entirety of the thesis.

I would also like to thank my supervisor Gernot Heiser, for his commentary, encouragement and the opportunity to undertake such an interesting and challenging thesis.

Contents

1 Introduction **5**

1.1 Motivation 5

1.2 Goals 6

1.3 Thesis Structure 6

2 Background **8**

2.1 Android 8

2.1.1 Linux 8

2.1.2 Bionic 9

2.1.3 Dalvik VM 10

2.1.4 Binder IPC 11

2.1.5 The Role of IPC in Android 12

2.1.6 Licensing 12

2.2 OKL4 3.0 14

2.2.1 Small Memory Footprint 14

2.2.2 High Performance IPC 14

2.2.3 ARM Support 15

2.2.4 Transparent Superpages 16

3 Approach **18**

3.1 Overview 18

3.2 OS Personality 19

3.2.1 Component-based Design 19

3.2.2 Drivers 20

3.3 Bionic-compatible C Library 20

3.4 Dalvik VM 21

3.5 System Server 21

3.6 IPC Mechanism . . . 22
3.7 File System . . . 22

4 Implementation 23
4.1 OS Layer . . . 24
4.1.1 File System . . . 24
4.1.2 Virtual Memory Subsystem . . . 24
4.1.3 Process Creation . . . 25
4.1.4 Thread Creation . . . 26
4.1.5 Futexes . . . 26
4.1.6 Timer Driver . . . 28
4.1.7 Video Driver . . . 28
4.1.8 Input Drivers . . . 29
4.2 Dalvik Application Framework . . . 30
4.2.1 UI Layout . . . 30
4.2.2 Image Resources . . . 31
4.2.3 Event Loop . . . 31
4.3 System Server . . . 32
4.3.1 InputDeviceReader . . . 33
4.3.2 InputDispatcher . . . 33

5 Evaluation 35
5.1 Improving Performance with OKL4 . . . 35
5.1.1 Input Driver Framework . . . 35
5.1.2 IPC . . . 36
5.1.3 Superpages . . . 36
5.2 Evaluation Environment . . . 36
5.2.1 Android Developer Phone 1 . . . 36
5.2.2 Android . . . 37
5.2.3 OKL4 . . . 37
5.2.4 Linux . . . 37
5.3 Benchmarks . . . 38
5.3.1 CaffeineMark 3.0 . . . 38
5.3.2 2D Drawing with Superpages . . . 39
5.3.3 IPC Micro-benchmarks . . . 40

5.3.4 Input Framework 41
5.4 Results and Analysis 43
5.4.1 CaffeineMark 3.0 43
5.4.2 Superpages 45
5.4.3 IPC Micro-benchmarks 47
5.4.4 Input Framework 52
5.5 Conclusions & Discussion 56

6 Future Work **59**
6.1 Limitations 59
6.1.1 File System 59
6.1.2 Dynamic Linker 59
6.1.3 C++ Support 60
6.2 Additional Features 60
6.2.1 Transparent Superpages 61
6.2.2 Binder 61
6.2.3 Zygote 62

A System Calls **63**

Chapter 1

Introduction

1.1 Motivation

Mobile phones of today are becoming increasingly powerful and are now offering a feature-rich environment allowing users to connect to the internet, check their email, play 3D games, and much more. These phones are equipped with large complex operating systems derived from solutions that originally targeted desktop architectures. The Apple iPhone uses a cut-down Mac OS X operating system called OS X iPhone, and Google's Android utilises the standard 2.6 Linux kernel with some additions [1]. Such mobile phone operating systems were not designed from the ground up to address the issues of embedded systems which include a low-power processor, limited system memory, small caches and possibly operating on battery power.

Embedded systems are not required to provide the complex environments found on desktops, and compact solutions in the form of microkernels aim to make the most of embedded hardware. The OKL4 microkernel, developed at Open Kernel Labs, is a member of the L4 family of microkernels. OKL4 has been designed for embedded systems to be small and fast and could be used as a base for the user environment on a mobile phone.

Google's Android mobile phone operating system is an initiative of the Open Handset Alliance, a consortium of business companies aiming to develop open standards for mobile devices. Android has been open-sourced and would make an ideal candidate for replacement of its Linux kernel with an OKL4-based system. OKL4 offers a small memory footprint, a high-performance inter-process communication (IPC) mechanism, and has been optimised for processors found in mobile phones. It also runs on the modem software stack for many phones today, including Motorola's single core phone. This makes it possible to produce a single-core OKL4-based Android phone, reducing the cost and power requirements of a handset. By making the most of the limited hardware available on mobile phones, it may be possible to run more complex software on cheaper hardware providing a feature-rich platform that is more affordable for end-users.

The main motivation for this thesis is that a minimal Android implementation could reduce memory usage, and improve performance by executing less cycles, saving power as a result.

1.2 Goals

The primary goal of this thesis is to port enough of Android to OKL4 to be able to run Android applications. This system will provide a basis on which comparisons can be made between an OKL4-based Android and a Linux-based Android. A proof-of-concept system will be able to demonstrate that microkernels can power large systems without sacrificing performance. To achieve this goal I need to:

- Build an operating system on top of OKL4
- Develop device drivers and a user framework for input, video and timing
- Port the Dalvik virtual machine
- Port the user and system Java runtime framework
- Benchmark the system and make meaningful comparisons to Linux-based Android

The true measure of the success of this work will be in whether we have learned how the change in kernel affects real-world performance in Android. As a result, we will be able to determine whether it is worth continuing to port Android to OKL4.

1.3 Thesis Structure

Chapter 2: Background

This chapter provides an overview of the OKL4 microkernel, and the Android mobile phone operating system. This provides necessary background information for the rest of the thesis.

Chapter 3: Approach

This chapter describes the high-level approach taken to build an OKL4-based operating system to support Android, and discusses what components of Android are required to be able to run user applications.

Chapter 4: Implementation

This chapter details the low-level implementation of my OKL4-based system and the design trade-offs chosen. Issues faced while porting Android are also discussed.

Chapter 5: Evaluation

This chapter presents an evaluation of my OKL4-based system, and provides a comparison against Linux-based Android. Benchmarks are presented and analysed.

Chapter 6: Future Work

This chapter discusses the shortcomings of my Android port, and proposes future work.

Appendices

Appendix A provides the list of Linux system calls that were required for my Android port.

Chapter 2

Background

2.1 Android

The Android platform is a competitor to Apple's iPhone and Windows Mobile. Backed by 50 industry partners such as ARM, Google and Qualcomm, Android has begun to see an increasing presence in the market. Google predicts there will be at least 18 phones on the market by the end of 2009 [2]. This makes it possible to compare OKL4 and Linux in a mobile phone environment and in a real-world commercial system.

2.1.1 Linux

The general work of this thesis involves replacing the Linux kernel with an OKL4-based system. Linux has been continuously evolving for over a decade and provides an environment that is complete enough for desktop machines. While it can be used for embedded systems and has been over the years, Linux includes many features that are not required for a mobile phone environment. For example, the highly complex signal API and process groups serve little purpose for mobile phone applications. There are also many overlapping IPC mechanisms, D-Bus, pipes, sockets and Binder. While some have specific uses such as sockets to access the network stack, the amount of code needed to perform these operations could easily be reduced by separating policy from the mechanism.

With the kernel consisting of millions of lines of code [3] compared to OKL4's measly 15,000 [4] there is likely dead weight, such as mentioned above, that does not serve the requirement of a mobile phone. An OKL4-based system would be a minimal system containing only what is required for the platform to operate. OKL4 itself is a

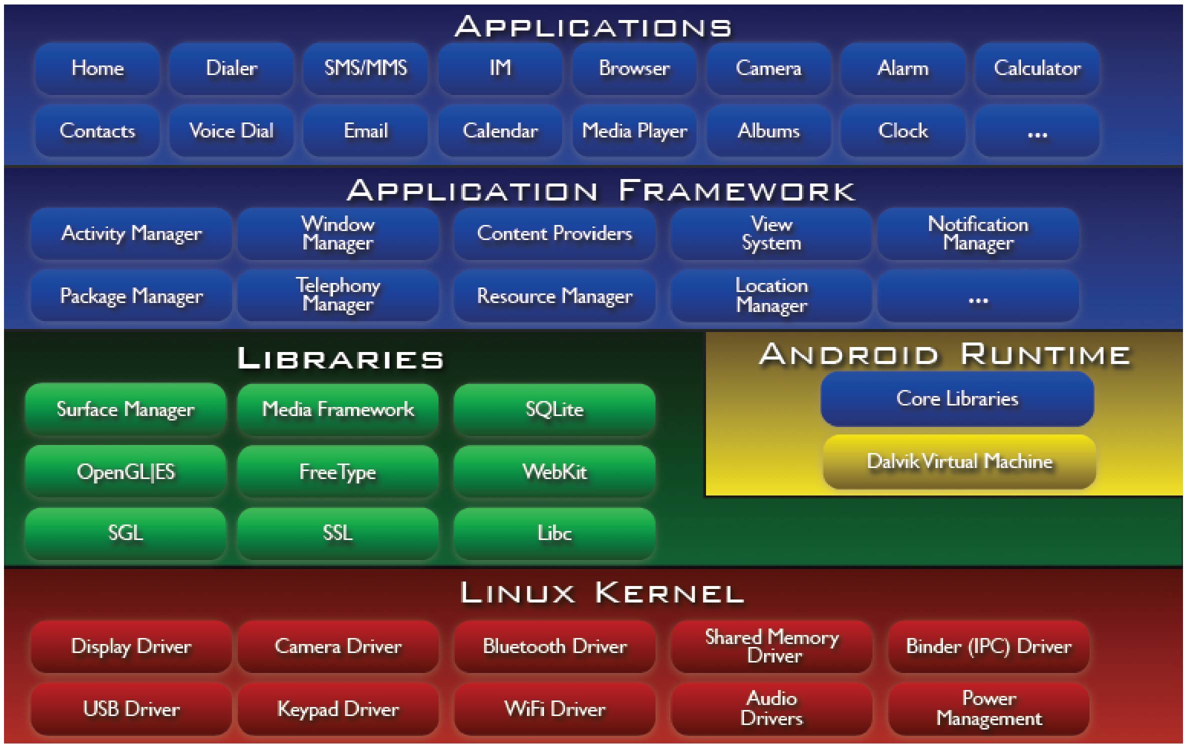


Figure 2.1: A system overview of Android

minimal kernel, and the userspace-support to have a complete operating system has to be written. This bottom-up approach may well have an impact on performance and memory usage.

2.1.2 Bionic

Google developed a custom C library for Android called Bionic which provides a C system interface to the Linux kernel. It has a minimal pthreads implementation to support thread creation and synchronisation, and contains some Android-specific additions such as system properties and logging. These changes make Bionic not entirely compatible with GNU libc [1].

Apart from the new features, Google's aim was to have a small and fast libc implementation more suited to embedded hardware. The stripped shared library of Bionic is 238KiB in size, in comparison to the 1.11MiB stripped GNU libc implementation compiled with the *arm-unknown-linux-gnueabi-4.2.4* toolchain. It is important that Bionic be small as it is linked against every application, and mobile phones have far less system memory than desktop machines. The Android Developer Phone 1 (ADP1) has only 128MiB of RAM, and more importantly has no swap space.

2.1.3 Dalvik VM

The core of the user environment is the Dalvik virtual machine. The Dalvik VM executes Dalvik byte-code and was written by Dan Bornstein, a Google engineer. Java class files, the compiled format of Java code, are converted into Dalvik dex files through the use of a developed tool. The end result is that Dalvik VM can execute applications written in Java. However, the current implementation of Dalvik does not have a just-in-time compiler. All Dalvik applications are interpreted and therefore will execute much slower than native code.

Dalvik includes several time- and space-saving optimisations. Firstly, the dex file format provides several benefits over the standard Java jar format (a collection of compiled Java class files). An uncompressed dex file is approximately the same size as a compressed jar file [5]. This provides a memory saving as Dalvik executables do not need to be decompressed into system memory, and they can also be executed from a memory-mapped file. The latter means that portions of the dex file that are not commonly in use can be ejected from memory, and reloaded again on-demand.

Another important optimisation in Android is the Zygote process. The Zygote is a Dalvik VM process that has loaded core Java libraries and is ready to begin executing byte-code. Its purpose is to spawn new Dalvik VM instances whenever a new application is launched. In that event, the Zygote forks with copy-on-write semantics [5]. This improves process start-up time because new applications have all core libraries pre-loaded and initialised as done previously by the Zygote, and can begin executing immediately.

There is also a memory saving as the child process shares Dalvik text and data with the Zygote. The aforementioned Java libraries are quite large. For example, the optimised dex of the Java core library is 3.5MiB in size. This is not something you would want duplicated in every Dalvik VM process.

The Zygote optimisation is possible due to the design of Dalvik where each process executes its own VM instance, as opposed to running all Java applications in one process. This guarantees that if one Java application crashes it will not bring down all Java applications and is made possible by the process isolation provided by the OS. This is quite important when all user applications run on Dalvik, as having the user environment crash due to one faulty application would not provide an enjoyable user experience.

2.1.4 Binder IPC

The *System Server* process provides access to system services such as the Package Manager and the Power Manager. It also delivers events to applications to notify them of user input. This process is written in Java and runs on a Dalvik VM instance. User applications perform Inter-Process Communication (IPC) to the System Server to gain access to the provided services. It is also the case that user applications themselves can provide services to other user applications. This is all made possible by the Binder IPC driver addition to the Linux kernel by Google.

Binder in Android is a reduced custom implementation of OpenBinder, a system-level component architecture developed at Be Incorporated and then Palm Incorporated. OpenBinder was intended to be a complete distributed object environment with abstractions for processes and its own shell to access the Binder environment.

The implementation of Binder on Android allows processes to securely communicate and perform operations such as remote method invocation (RMI), whereby remote method calls on remote objects look identical to local calls on local objects. Performing a remote function call means the function arguments first need to be delivered to the remote application. This process of converting data into and back out of a transmissible format is called marshalling, and is achieved through the use of the Android interface definition language (AIDL). The necessary code to perform marshalling is automatically generated for both the server and the client to enable seamless remote function calls.

Each registered service runs a thread pool to handle incoming requests. If no threads are available, the Binder driver requests the service to spawn a new thread. This way multiple requests to a single service will not result in denial-of-service (DOS).

Binder can be used to facilitate shared memory between processes. One of the additions to the Linux kernel was an *ashmem* device driver. Using ashmem it is possible to allocate a region of memory, represented by a file descriptor. This file descriptor can be passed through Binder to other processes. The receiver can pass the ashmem file descriptor to the *mmap* system call to gain access to the shared memory region. The key feature of ashmem is that the kernel can reclaim the memory region at any time. This only occurs if the region is marked by users as "purgeable".

A distributed object model API is provided in the C++ and Java environments to use Binder. One example using this API is the *android.hardware.SensorManager* class. The SensorManager is instantiated as a local object, but it maintains a private variable of type *ISensorService* that uses RMI through Binder to gain access to the device's sensory information.

Binder implements security by delivering the caller's process id and user id to the callee. The callee can then validate the sender's credentials [6].

In Binder, remote objects are referred to by Binder references. A reference is a token, or capability, that grants access to a remote object. Having access to a remote object lets you perform RMI on that object. One process can pass a reference through Binder to give another process access rights to a remote object. The Binder driver takes care of this, and blocks a process from accessing a remote object if it does not have the correct permissions.

2.1.5 The Role of IPC in Android

The layout of an Android system at runtime is shown in Figure 2.2. All services that run in the System Server are accessed through the use of IPC. If an application wants to launch a new activity (an application), Binder is used to request this operation of the Activity Manager. Requesting that the screen be kept on requires an IPC to the Power Manager. Flushing the contents of a window to the screen requires notifying the *Surface Flinger*, and again this is done using IPC.

Each of these managers provides a service by registering with a system component called the *Service Manager*. The Service Manager keeps track of different services in the system and provides dynamic service discovery to user applications. Services are requested by name. To avoid the chicken and egg problem, the Service Manager is given a unique identifier, the integer zero. This way user applications do not need to request of the Service Manager access to the Service Manager.

The most commonly-occurring use of Binder is to notify the Surface Flinger of window updates, and to dispatch input events from the System Server to user applications.

2.1.6 Licensing

It is important to note that the state of licensing in the Android project does affect the design and performance of Android. Linux is distributed under the GNU General Public License (GPL) Version 2 which means the source code is available to the general public [7]. The restrictions of this license are that source code must be released for any additions made, including drivers in the kernel.

Manufacturers like Qualcomm protect their intellectual property (IP) by providing compiled binaries instead of source code. The GPL license means that generic drivers are inserted into the kernel to deliver data generated by a device to a close-sourced binary executing in userspace [1], and this is bound to have an effect on the system design as well as performance.

All Android userspace code is either licensed under the Apache License or the BSD License which are more permissive than the GPL and do not create any issues that affect manufacturers and their intellectual property.

Under the OKL4 commercial license, there is no obligation for manufacturers to distribute any source code, and hence no restriction on the system design and implementation.

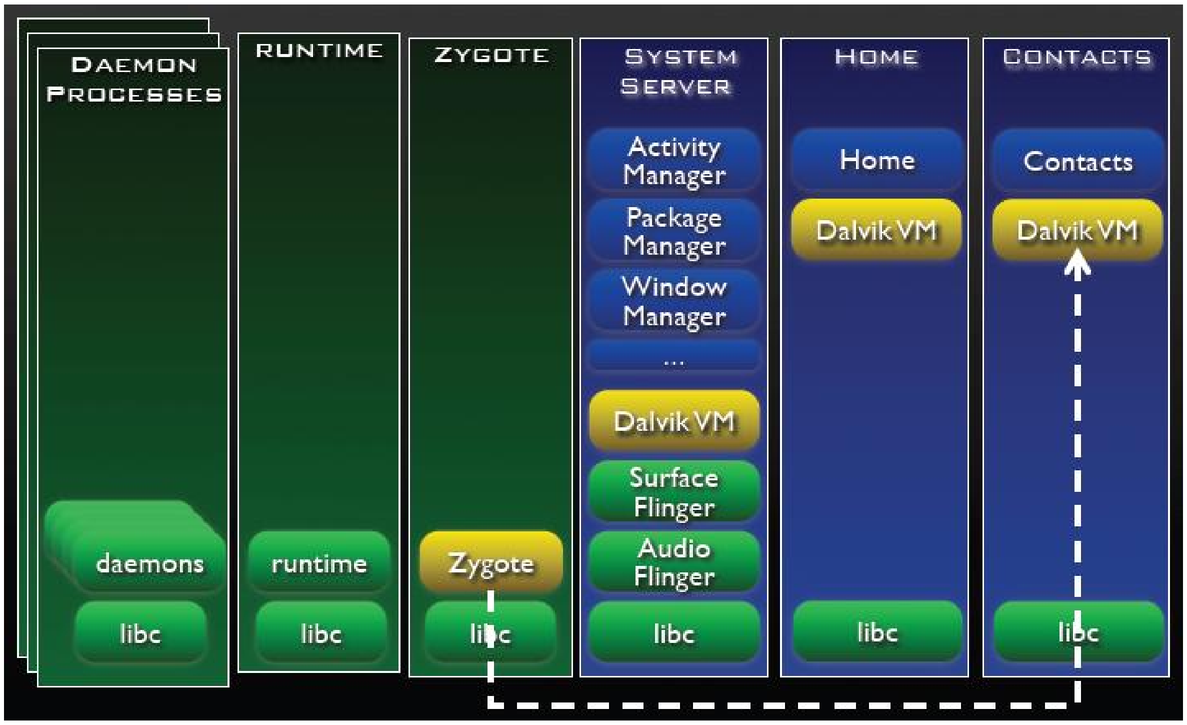


Figure 2.2: A runtime overview of Android.

Figure 2.2 shows the System Server that applications must IPC to in order to gain access to the Package Manager, the component responsible for managing all installed application. New Dalvik VM instances such as the *Contacts* Java application are spawned from the Zygote.

2.2 OKL4 3.0

OKL4 is a high-performance microkernel. It provides a minimal, but portable set of abstractions to the hardware. Its design and API derives from the L4 family of microkernels.

2.2.1 Small Memory Footprint

OKL4 targets embedded systems which consists of a wide range of hardware with varying requirements. In general this means power-saving hardware, limited system memory and perhaps operating on battery power. As a result OKL4 is very small and tries to minimise its presence in the caches allowing it to execute with less cache misses and leaves more room for user applications.

The ARM implementation of OKL4 fits in 78KiB of memory [4], far short of the 3MiB kernel image supplied with the ADP1. Of course this does not include the userspace support that is necessary for building an operating system on top of OKL4, but a minimal implementation will be far less featureful and will be smaller as a result. In short, this presents a lesser memory requirement for the operating system, providing more memory to applications. Alternatively, lower-end phones with less memory will be better able to run Android.

2.2.2 High Performance IPC

The performance of inter-process communication (IPC) can have a large impact on performance in a microkernel-based (or componentised) system. All primary system services must be implemented in userspace and IPC is used for processes to access these services. OKL4 recognises the importance of IPC and offers a highly optimised hand-crafted assembly IPC fast-path to achieve maximum performance. Compared to first-generation microkernel implementations such as Mach, OKL4's IPC mechanism is a factor of 20 faster [4]. This is in part due to the bare-bones nature of OKL4 IPC. It provides a simple mechanism to transfer data, no more, no less.

As discussed previously in Section 2.1.4, the design of Android is somewhat componentised with processes receiving input events from another userspace process. A high-performance IPC mechanism can help to improve performance for these common operations.

Message Registers

Data is transferred from one thread to another using message registers. The platform implementation of OKL4 I am using provides 32 message registers, allowing an IPC operation to transfer at most 32 words (4 bytes per word) of data. The message registers consist of several CPU general-purpose registers, with the rest located in the thread's user-level thread control block (UTCB) in main memory. The pre-determined location of the message registers by the kernel allows for a faster transfer as it already knows where the data is. The use of shared memory is encouraged for larger transfers that do not fit in the message registers.

2.2.3 ARM Support

OKL4 supports many processor architectures. More importantly it is optimised for ARM processors. Cores based on the ARMv5 and ARMv6 architectures are readily found in mobile phones and this makes OKL4 a great choice for replacing Linux in the Android platform.

A particular advantage of OKL4 on ARMv5 chipsets is its superior context-switch handling. One issue with the ARMv5 is that TLB entries do not have an address-space identifier (ASID). This means TLB entries of different processes cannot be differentiated from one another and therefore the TLB must be flushed clean to prevent processes from accessing each other's memory.

The second issue is the use of virtually-indexed virtually-tagged (VIVT) caches. Different address spaces can have the same virtual address mapping to different physical addresses. This is especially common in multiple address space operating systems such as Linux. To avoid cache aliasing where an incorrect cache entry is present, there should be only one mapping per virtual page, and therefore the caches must be flushed on every context-switch.

However, it was demonstrated by van Schaik and Heiser [8] that it is possible to avoid flushing the TLB and caches on every context-switch on the ARMv5 architecture by making full use of ARM domains and the Fast Context Switch Extension (FCSE). It was pointed out that the Linux maintainers refused the offered kernel patches to take advantage of these features forcing Linux to flush the caches on every context-switch for ARMv5.

The ARMv6 features virtually-indexed physically-tagged (VIPT) caches and TLB entries do contain an ASID. The TLB translation process outputs the physical address being mapped to, and this is used to select the correct cache line allowing multiple mappings of the same virtual page to co-exist. The use of ASIDs ensures processes access only their own mappings without needing to flush the TLB.

The Android Developer Phone 1 that will be the basis of the Linux/OKL4 comparison utilises an ARMv6 processor, and as such these performance benefits will not be available. However, the ARMv5 is still a popular architecture for low-power and low-cost applications and should not be overlooked.

The Motorola Evoke QA4 is one such phone that uses an ARMv5 chipset. Its single-core ARM9 processor runs a virtualised Linux environment on top of OKL4. A native implementation of Android on OKL4 would allow running Android on a single CPU like in the QA4. Such a system would mean Android could run on cheaper hardware and would be more affordable for end-users.

2.2.4 Transparent Superpages

OKL4 supports the mapping of all page sizes for the ARMv5 and ARMv6. From the ARM11 manual [9], the following page sizes are supported: 4KiB, 64KiB, 1MiB, and 16MiB.

As a microkernel, OKL4 supplies mechanisms without policy where possible. This is true of the use of superpages in an OKL4-based operating system. The userspace OS personality is responsible for allocating system memory to user processes, and with the mechanism supplied by OKL4, it can implement transparent superpages.

Transparent superpages means that user applications allow the operating system to decide what page size(s) should be used for memory allocations. This is not restricted to using a single page size, in fact a mixture of page sizes could be used depending on the implemented policy.

The implications of using large page sizes rather than the standard 4KiB are several. A larger page size means higher TLB coverage, and therefore less TLB misses and less time spent processing them. It also means that internal fragmentation, where many pages are allocated but each does not have complete utilisation, may be higher. A process that allocates 64KiB for a heap may only use 32KiB, and when using a 64KiB page size, 32KiB of physical memory has gone to waste. A smaller page size would not have this problem, as 32KiB fits into 8 4KiB pages.

There is a trade-off between performance and memory usage due to internal fragmentation, but any imaginable policy can be implemented. It is feasible that a policy tailored to Android specifically could be constructed to provide a balance between performance and memory usage.

Superpages can be accessed in Linux through the HugeTLB library [10]. Through examination of the Android source, superpage support is not compiled into the kernel and is it is not used by any userspace code in Android.

Chapter 3

Approach

3.1 Overview

Primarily the thesis will involve porting Android components to my OKL4-based operating system and benchmarking the performance of the resulting system. A complete port of Android to OKL4 was not feasible in the time frame. Therefore I decided to port a subset of Android that would provide a comparison for benchmarks. Investigation early on in the thesis revealed there were several core system components required to build a minimal but functional Android system that could run a small subset of applications.

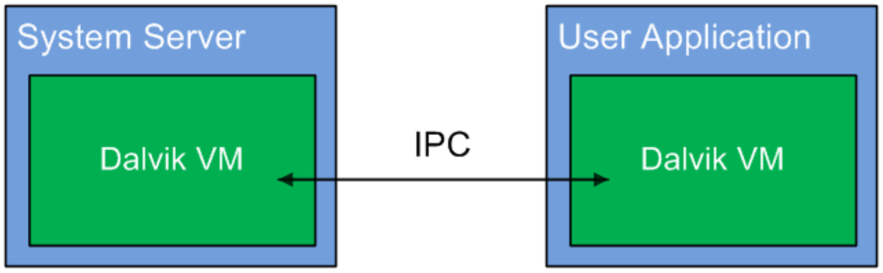


Figure 3.1: High-level requirement of an OKL4-based Android system.

As shown in figure 3.1, a high-level view of a basic functioning Android system requires the System Server process communicating to a Dalvik application in a separate process. A top-down design approach can be applied to determine the lower-level system requirements.

Firstly, a form of IPC is needed to facilitate the communication between the application process and the system process. Secondly, Dalvik VM requires a Bionic-compatible

C library and operating system support for Linux system calls. Thirdly, the System Server needs access to drivers, such as video and input, to be able to perform its role. Finally, Dalvik needs to load Java libraries at runtime, and this requires a file system.

Therefore the basic requirements to supporting the Android runtime on the OKL4 microkernel are:

- OS Personality (with drivers and a file system)
- IPC mechanism
- Bionic-compatible C library
- Dalvik VM
- System Server

3.2 OS Personality

A microkernel does not provide a complete operating system. Instead, most standard OS services are implemented in userspace. This collection of userspace servers form what is known as the operating system personality. Almost all system policy is implemented in the OS personality, bar a few things which still live in the kernel. For example, OKL4 still dictates how threads in the system will be scheduled.

3.2.1 Component-based Design

An OKL4-based Android system designed from scratch would appear fundamentally different to a Linux-based system. Microkernels encourage the design of a componentised system running in userspace. To make progress in the system, components communicate via the provided IPC mechanism.

Native code written for Android expects a POSIX API as provided by their libc implementation (Bionic) working with the Linux kernel. The OKL4 system call API does not provide POSIX semantics. Therefore, my system will need an OS personality running in userspace to be able to provide the required functionality normally provided by Linux and Bionic. This OS layer will also need to provide user applications access to devices.

The OS personality will consist of a server loop processing Linux system call requests from userspace applications. The thread that handles this main server loop will be referenced as the *rootserver*.

The design will be monolithic to simplify programming and to maintain performance. Drivers will conceptually be implemented as separate components with their own thread of control. This is to facilitate the componentisation of unrelated drivers into separate protection domains at a later date, whereby driver components can be easily swapped out with different implementations depending on which hardware platform you are targeting.

The OKL4 3.0 distribution provides a library layer on top of the L4 API called *libokl4* to assist in the construction of OS personalities. It performs a lot of work common to all OS personalities, such as protection domain and thread creation, and is the recommended method to maintain compatibility with newer OKL4 releases. The use of libokl4 will save having to re-invent many wheels.

3.2.2 Drivers

Very few drivers had been written for the Android Developer Phone 1 at the beginning of the thesis. The approach taken was to implement drivers on a need-by-need basis.

As briefly mentioned in the previous section, each driver will be implemented as a separate conceptual component in userspace with its own thread of control. This was chosen because implementing all drivers to be handled in a single thread of control destroys code portability due to the implementation being affected by the behaviour of each driver.

On OKL4 there is more processing overhead because the kernel delivers interrupts using IPC. This requires a context-switch that is not necessary in a monolithic kernel. However, the overhead is minimal and interrupt latency is largely not an issue with the devices available on a mobile phone.

3.3 Bionic-compatible C Library

There are two approaches to developing a Bionic-compatible C library. One is to port Bionic to run on my OKL4-based system by redirecting system calls to the OS personality. The other is to port any missing features from OKL4's minimal libc implementation.

OKL4's libc proved to be lacking in the necessary features and would have required a large porting effort, this approach would also not guarantee the same semantics offered by Bionic and could lead to compatibility issues that would need to be resolved.

For the above reasons, I chose instead to port Bionic to run on OKL4 and provide emulation for system calls as needed by the applications running on my system.

3.4 Dalvik VM

The Android runtime is powered by the Dalvik virtual machine. It was written to be portable to different architectures and different operating systems. As such it links to very few other libraries, with Bionic libc being the only prominent one.

Dalvik presents one other major porting issue. The core Java libraries are stored in the file system in an unoptimised format. When Dalvik first loads these libraries it optimises them and stores a copy into a cache on the file system. As other Dalvik instances start up, they can use the optimised copy from the cache so the optimisation process need only be performed once. To perform this optimisation Dalvik invokes the *dexopt* tool using fork/exec. This requires a write-capable file system which I did not have (discussed in Section 3.7), and implementing these system calls would have taken time away from other porting work.

However, after some investigation I discovered an alternative method. It is possible to upload the libraries to the Android simulator and invoke the optimisation process by launching Dalvik with specific parameters. The pre-optimised libraries can then be downloaded back. I used this method to avoid spending a lot of time implementing and debugging features that might not be used anywhere else.

3.5 System Server

Porting the entire System Server is a non-trivial exercise as it involves parts of the system that are not available on my OKL4-based system such as power management. Instead, I need only a partial port containing the components required for benchmarking. The parts of the system I wished to benchmark were IPC, video and input. At most the System Server would be ported with these components.

3.6 IPC Mechanism

Investigation into the use of Binder in Android revealed that it is used for dispatching input events to user applications. Completing a full port of Binder is not necessary to make a comparison for this rather straight-forward duty. However, obtaining a pixel buffer shared with the video subsystem does require Binder's shared memory via file descriptor feature. Again Binder can be bypassed by replacing dynamic allocation of shared memory buffers with static allocation by my OS at start up. This is possible because my port will launch pre-determined applications for benchmarking.

Binder is also used to let user applications provide their own services to other user applications, and as described in section 2.1.4, Binder references can be passed around to distribute access to a service. This framework cannot simply be substituted by OKL4 IPC, and without good reason, does not warrant porting the Binder driver, which consists of 3,500 undocumented lines of source code.

3.7 File System

At minimum a read-only file system is necessary to be able to load Java libraries at runtime. The OKL4 build system allows you to place files into memory and access them using libokl4 and the OKL4 environment. The OKL4 environment stores build-time information, such as the memory location of files that were inserted into the boot image. The ADP1 does have an SD card device that could have been used, but a driver and file system implementation did not yet exist.

Using a real device and a real file system implementation provides more flexibility. Files can be created and written to at runtime, but this functionality is not required for an initial Android port. The solution of having an in-memory file system was chosen because it was simpler to implement and allowed me to get started on porting the rest of the system much sooner than if I had stopped to write an SD card driver and port a file system implementation.

Chapter 4

Implementation

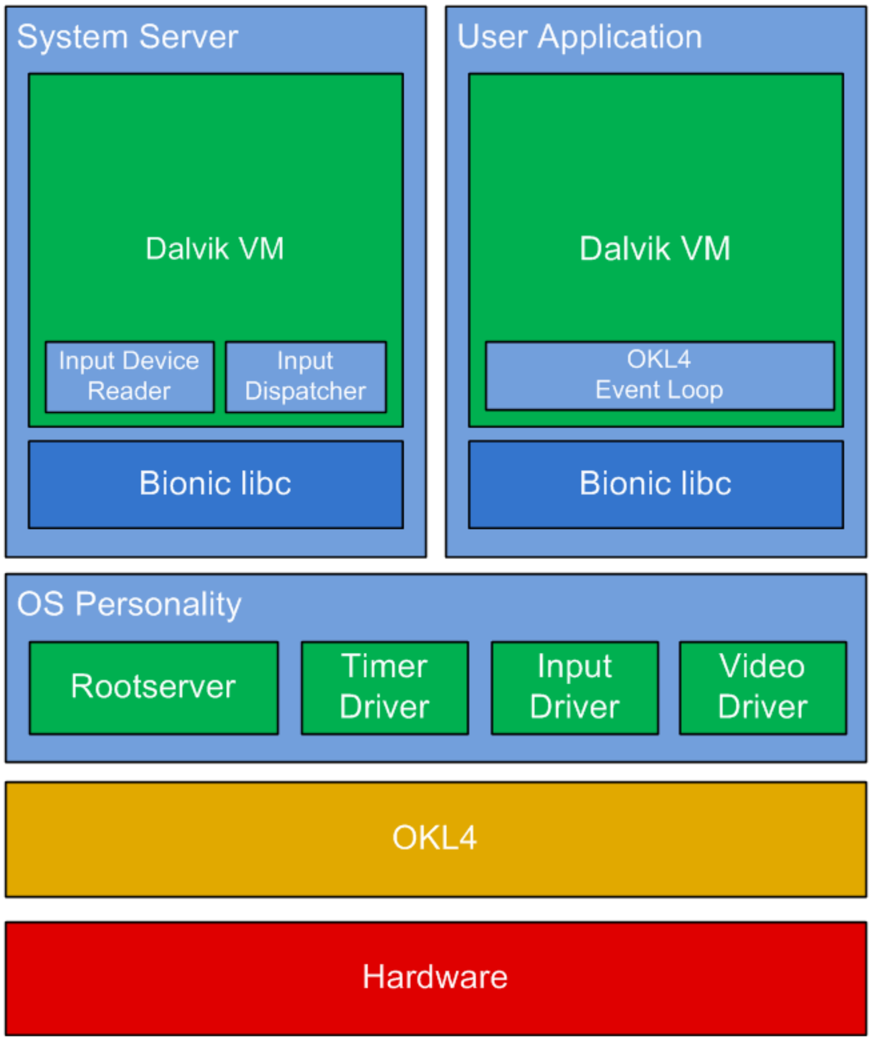


Figure 4.1: System overview of OKL4-based Android.

4.1 OS Layer

Building an OS personality on top of OKL4 to support Android requires numerous subsystems to be implemented. I developed subsystems with the policy of keeping the implementation minimal and simple, and not to succumb to premature optimisation. Optimising early not only complicates the design, but may be a complete waste of time if that component is not in a critical code path. Optimisation of the system would come at a later date if performance issues presented themselves. A list of Linux system calls that have been implemented is available in Appendix A.

4.1.1 File System

A read-only in-memory file system as described in Section 3.7 was implemented. Extra framework was required to let user processes access this file system. As such, each process is allocated a file table to access files in the in-memory file system. An entry in the file table contains the state about an open file. This state is simply the address of the file data in memory, the size of the file, and the current seek position within the file. This was all that was necessary for Dalvik to operate.

4.1.2 Virtual Memory Subsystem

The VM subsystem is responsible for managing virtual and physical memory. It needs to handle the demands of Android applications, which most commonly is the mmap system call. POSIX semantics of the mmap system call expect allocated memory regions to either be zero-ed out or contain the contents of a specified file.

I was able to implement the required semantics by making use of the callback support of libokl4 *memsections*, where a memsection in libokl4 is a block of memory consisting of a base, a range and a variety of other attributes such as access permissions and page size. The callbacks enable you to choose when to map a virtual page to a physical frame. This can be done when the memsection is created, or in response to a page fault. The physical memory frame can be zero-ed out before allocation to a user. Meeting the requirement of backing memory pages with the contents of a file was simple due to the use of an in-memory file system. File contents are already in memory and can be copied directly without having to access a disk device.

As a result, my system supports pre-mapped and lazily-mapped memsections that match mmap semantics. Lazy-mapping of pages was a necessity to reduce memory usage. Dalvik allocates several large buffers that are megabytes in size, and my in-memory file system grew to over 50MiB.

Superpage Support

Specifying the page size of memsections is supported by libokl4. However, I was only able to implement explicit superpage support. A bug in the platform code meant that the physical addresses I passed to OKL4 for mapping were only aligned to 4KiB blocks, causing the mapping of 64KiB pages to fail. For example, the physical address 0x00050000 would translate to 0x1004F000. The misalignment also varied between different compiles and I was only able to successfully map large pages by manually offsetting my addresses by the size of the last misalignment.

There was little time remaining to investigate as to why this was happening, but mentioned work-around was used to enable user applications to allocate regions of memory using a 64KiB page size. This was sufficient for basic benchmarking purposes. A simple function *malloc_64k* was made available to userspace, and has the same interface and semantics as the standard *malloc*.

4.1.3 Process Creation

Linux creates new processes using the *fork* system call to duplicate the address space of the caller. This can be followed by a call to *exec* to load a new executable. The requirements for fork that have been discussed so far are to launch the Dalvik optimisation tool, dexopt (section 3.4), and for the Zygote optimisation (section 2.1.3).

Porting enough of Android to be able to perform the Zygote optimisation was not required and remains as future work, and so process creation was limited to my OS deciding at build time which applications to execute.

New processes are created by parsing executables located in the in-memory file system with a custom-written ELF loader. The ELF loader was ported from my project code in COMP9242 Advanced Operating Systems. File-mapped memsections could then be created for the text and data segments, and a zero-mapped memsection for the bss segment. All system calls to the OKL4 microkernel to create new address spaces were handled by libokl4 which provides the *protection domain* abstraction for address spaces.

It was necessary to implement program arguments (*argc* and *argv*) to be able to launch multiple Dalvik VM instances in my system from the same binary. This was achieved by copying the argument data into the new process's address space, and pointing the *L4_UserDefinedHandle()* (thread-local storage for OKL4 threads) to this piece of memory. When the process starts up, it finds the arguments and passes them to the program's *main* function.

4.1.4 Thread Creation

Android makes use of multiple threads in its applications and in the system framework. The process of starting up new threads is a combination of OS and user library code.

In my OS personality, libokl4 is used to create and start new kernel threads. The pthreads library in Bionic is responsible for creating new threads, and providing synchronisation tools for a multi-threaded environment. During thread creation, the pthreads library chooses a stack location and places information about which function to execute above the given stack pointer. The clone system call on Linux, which handles process and thread creation, consisted of an assembly routine to do the system call, and then sets up the registers to make a call to the pthreads __thread_entry function. A different return value from clone is used to force the parent and child to take different code paths out of the system call.

On OKL4 this process needs to be a little different due to the lack of a fork implementation. Start up information (which function to execute) is still stored above the stack pointer with the area below to be used for the stack. However, the instruction pointer given to the newly-created thread always points to a specific assembly routine. The routine reads the function address and arguments from above the stack pointer, and correctly passes it to the pthreads __thread_entry function.

4.1.5 Futexes

Linux provides a fast userspace mutex (futex) API as construction blocks for more sophisticated thread-synchronisation tools, such as semaphores and condition variables.

Semantics

```
int __futex_wait(volatile void* ftx, int val, const struct timespec* timeout);
int __futex_wake(volatile void* ftx, int count);
```

The semantics involve synchronisation using the value and physical address of a single word as pointed to by the variable *ftx* in the above function declarations.

The semantics will be explained by providing an example. Thread A calls futex_wait to go to sleep until Thread B calls futex_wake on the same futex variable. In the event that Thread B calls futex_wake first, the futex semantics say that Thread A should wake up immediately. This is implemented in futex_wait by comparing the expected value of the futex word pointed to by *ftx* to its real value. The expected value is passed as the integer *val*. If the values match, Thread A will go to sleep. However, if Thread B changes the futex value and then calls futex_wake before Thread A performs the futex_wait call, then the value expected by Thread A of the futex word is now wrong, and a call to futex_wait will cause Thread A to wake up immediately as intended.

The important fact to learn from the above, is that in a futex_wait, the kernel compares the expected value of the futex word, to the *real* value of the futex word. This requires my OS personality to maintain page tables to be able to translate the word's virtual address into a physical address to access its value. More information on futex semantics can be found in Ulrich Drepper's article, *Futexes Are Tricky* [11].

Implementation

The OKL4 3.0 user's manual states that querying OKL4's page-table entries is deprecated. To stay compatible with future OKL4 releases, the remaining choices were to re-implement the Android pthreads package to not use futexes, or to simply maintain userspace page tables in the rootserver. It was foreseen that fork may need to be implemented at some point during the thesis, and a fork implementation would require userspace page tables. Therefore I made the decision to implement userspace page tables.

As semantics state that the value must be accessed on a call to *futex_wait*, the page containing the *ftx* variable must be mapped in and accessed. The page can be mapped in via a system call to OKL4, and then accessed, but it is not necessary to un-map the page. This would mean the page would not need to be mapped in on the next call to that futex because it was never un-mapped. This would result in less overhead. However, my aim was to build a solution that worked correctly and I was not interested in premature optimisations. As such I implemented the simpler map in, access, and un-map solution. Implementing the performance optimisation of leaving pages mapped in is future work.

I was presented with more implementation decisions regarding how to store the state of a futex. An example of futex state is the list of threads sleeping on that futex. There is no formal process of creating and destroying a futex. When a new futex is seen by the rootserver, the necessary state must be created. Processes in my system do not exit, so no implementation was required to destroy futexes.

A futex is identified by the 32-bit physical address of the futex word. To access the state of a futex, the 32-bit address can be used as a key to an associative container such as a hash table or an associative list. It was unclear whether futex performance would be a key factor in overall system performance, and it most likely would not be the case on software designed for a single CPU. Again, I was not interested in premature optimisations and took the simpler route of implementing an associative list to store futex state.

4.1.6 Timer Driver

The ADP1 has two timer devices on the bus, the general-purpose timer (GPT) and the debug timer (DGT). The GPT is already being used by OKL4 to implement timer ticks for context switching. This left the DGT free for me to use to implement timestamps and a sleep functionality.

It should be noted that the registers for the GPT and DGT are memory-mapped to the same virtual page. By mapping in this page to utilise the DGT, it is exposing kernel functionality into the timer driver. Without knowledge of other timers on the platform, this is a necessary evil. In this case it would be best to isolate the driver into a separate protection domain from the OS personality main component and verify that it does not tamper with the GPT. However, this is not in the scope of the thesis.

4.1.7 Video Driver

Video can be lazily pushed to the display as it has an internal buffer storing the last sent frame. This is different to VGA monitors which always require a new frame to display, and makes sense for the performance and power requirements of embedded hardware.

Therefore the video driver simply needs to provide three functions. It needs to provide access to the frame buffer, allow invoking of Direct Memory Access (DMA) operations to update the screen, and a notification for when the DMA is complete. DMA complete notifications were never implemented due to the sizable reverse engineering effort required to decipher some of Qualcomm's protocols on the bus. Instead, a busy wait has been substituted. This makes it impossible to include the video subsystem in real-world comparisons. However, measurements on the client rendering of their window are still possible.

The original driver was written by my co-supervisor Nicholas FitzRoy-Dale. My implementation is a port of his driver from the kernel to userspace.

4.1.8 Input Drivers

The ADP1 has several sources of events. There is the touch screen, the keyboard, the trackball, and various other buttons along the side of the phone. Each device requires its own driver, and each button press or touch generates an interrupt. As all work is interrupt-driven, only a simple IPC wait loop was required.

Userspace drivers for the keyboard, trackball and the touch-screen were written by my co-supervisor Nicholas FitzRoy-Dale. I ported them into my OS and implemented the framework that is described below.

It should be noted that each touch event consists of several events in the Linux input framework. These are the absolute x and y coordinates, the touch pressure, the width of the press, and a sentinel event to mark the end of a complete touch event. The OKL4 touch driver does not operate exactly the same as on Linux. Reading events from the touch device in */dev/input* reveals the Linux driver interleaves events (reusing older data) resulting in less data being transferred. It is most likely that data which has not changed is not sent again and it is up to the userspace code to maintain the entire state of the touch data. The OKL4 touch driver sends all events every time due to time constraints. I completed enough of the implementation to be able to detect touches, but did not interpret the data read from the driver. Instead, hard-coded touch sub-events were generated per touch interrupt. This was satisfactory to be able to take benchmarks of the input framework of Android on OKL4.

There were two sane implementations on OKL4 that I could choose from to move event data out of the driver. The first approach is to synchronously transfer each event using the OKL4 IPC message registers. The second approach is to use asynchronous notifications and a fixed-size shared memory region where parties read and write event data from a single-reader single-writer lock-free queue.

Synchronous IPC can cause the driver to block until the receiver is ready, and may result in devices dropping events if interrupts are not processed quickly enough. However, the same issue can occur if the fixed-size shared memory region is full and the driver has nowhere to place the data. The use of shared memory offers more flexibility than synchronous IPC by allowing the receiver to poll for events rather than block and wait for them. This increased flexibility is at the single cost of the minimum page size of 4KiB required for the shared memory region. It should be noted that the input driver framework was developed for the Quake demo that was displayed at NICTA Techfest 2009. For single-threaded game applications, polling of input is desired as blocking results in the game being unresponsive. This meant the asynchronous approach using shared memory was chosen.

For the Android input framework, either approach is acceptable but there was little gain to be had by changing the existing implementation.

4.2 Dalvik Application Framework

Starting up new Dalvik applications is a process that requires the System Server to provide several services including the Package Manager, the Window Manager and Binder. Due to time constraints it was necessary to find a way to launch applications without the full runtime stack. As such, I implemented my own user application framework under the guise of the standard API to develop a proof-of-concept demonstration that Android applications can run on an OKL4-based system. This reimplementation is used only for user applications and not the System Server.

I wished to benchmark the input framework of Android and so the System Server with all other components disabled. A discussion on the changes made is included below.

4.2.1 UI Layout

Android applications can supply an XML file describing their UI layout. Entries in the layout include platform-supplied classes such as *TextView* or *SurfaceView*, or even application-specific classes that inherit from Android framework classes, such as *LunarView* (in the Lunar Lander application) which extends from *SurfaceView*. When an application is packaged for release, this layout file is compiled into binary form for faster parsing during application launch.

I was unable to locate in the Java framework how this is properly handled and decided on an alternate approach, to reverse engineer the binary layout file. This enabled me to extract the class names that formed the UI layout and use Java reflection to construct Java objects based on these classes.

4.2.2 Image Resources

The method by which Android applications access image files is by passing an auto-generated enumeration to the Java framework. An object representing the image is returned. Without proper application launching, which involves the Package Manager in the System Server, this too required trickery to implement.

The auto-generated enumeration is in a class R (short for Resources) located in the same Java package as your application. Here is an example of the Resources class for the Lunar Lander application.

```
public final class R {
...
    public static final class drawable {
        public static final int app_lunar_lander=0x7f020000;
        public static final int earthrise=0x7f020001;
        public static final int lander_crashed=0x7f020002;
        public static final int lander_firing=0x7f020003;
        public static final int lander_plain=0x7f020004;
    }
...
}
```

The enumeration name is generated from the file path of the image. For example, *res/drawable/app_lunar_lander.png* is given the enumeration *app_lunar_lander*. Using Java reflection, it is possible to convert the enumerated integer back into a string that contains the enumeration name. Using this string I could re-create the file path and load the correct image.

4.2.3 Event Loop

User applications in Android consist of an event-loop that processes messages from a queue. Events are pushed onto the queue either in response to external events such as from input devices, or internally from the user application itself. My event loop waits on OKL4 IPC to receive key and touch events from the System Server, and thus replicates the behaviour of native Android applications.

4.3 System Server

The many system services that allow access to devices such as input, audio and video all run in the System Server. My original aims were to port the video and input frameworks, and substitute calls to Binder IPC with OKL4 IPC where possible. However, time permitted only a port of the input framework.

The input framework is in charge of getting events from the OS and delivering them to the appropriate application. It consists of two main components and are given the names used by Android in the code. These components are threads of control and are called the *InputDeviceReader* and the *InputDispatcher*.

The InputDeviceReader reads events from the kernel and pushes them onto a queue for the InputDispatcher to do the remaining work whereby the event is eventually dispatched to the user application using Binder IPC. My implementation also uses these two components, but has replaced event reading from the OS with my OKL4 equivalent solution as outlined in section 4.1.8, and has also replaced the Binder IPC dispatch with OKL4 IPC.

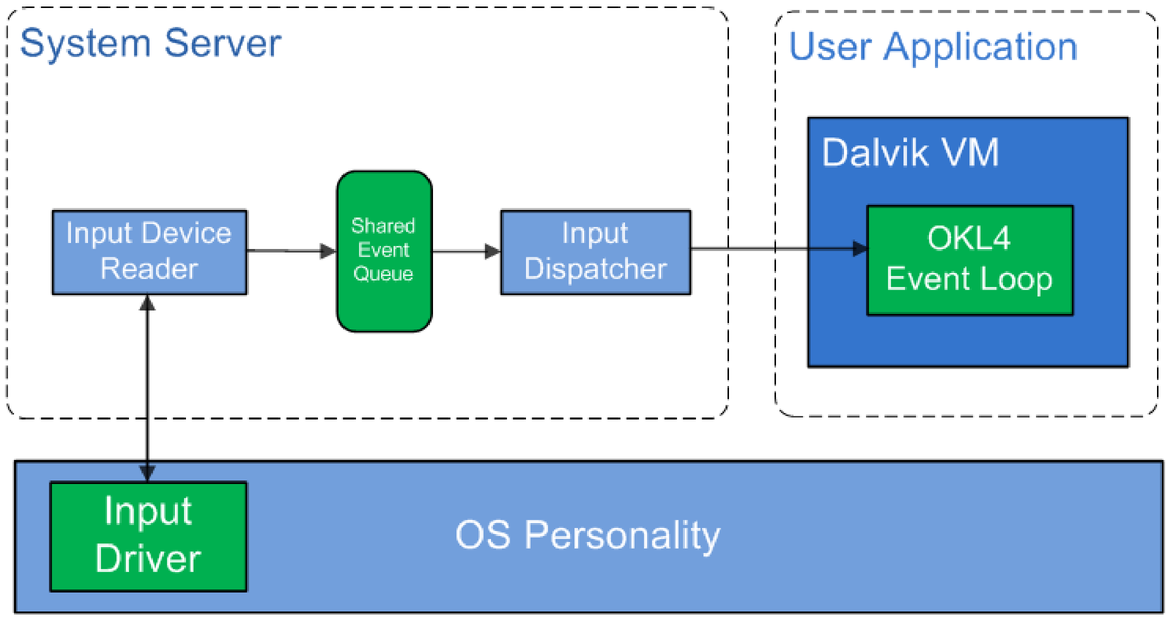


Figure 4.2: The input framework on OKL4. The system has been replaced at the input driver and IPC interfaces.

4.3.1 InputDeviceReader

This component is responsible for reading events from the OS and appropriately dealing with input events that may affect the system, such as turning the screen back on, hanging up a phone call, or just switching applications. If the environment is ready to accept user input, standard events are placed onto a queue shared with the InputDispatcher. Obtaining events is handled by native C++ code, and event processing is handled by Java code.

On Linux, a pair of system calls are used to read events from the kernel. First *poll* is used to know when events are available, then each event is extracted manually one-by-one using *read*. As mentioned in section 4.1.8, each touch event consists of multiple sub-events containing the data about different parameters. Reading each sub-event on Linux takes one system call and this will be slower than reading the entire event at once.

The OKL4 implementation for this component uses asynchronous notifications and shared memory as discussed previously in section 4.1.8. To obtain a complete touch event in this implementation requires only two system calls, an IPC *notify* by the driver and an IPC *wait* by the receiver, and the data is read from shared memory. On Linux anywhere from 3 to 6 system calls are required, one poll, and 2 to 5 touch sub-events with the variation resulting from the interleaving described earlier. This should improve the time taken to extract event data from the OS.

Due to missing a lot of system functionality on my OKL4 system, various portions of code had to be removed to make the event routing path of the InputDeviceReader functional. This involved removing all calls to power management, battery statistics and replacing checks to whether the display was on, off or dim, and whether the phone was locked.

4.3.2 InputDispatcher

This component is responsible for routing events to user applications. First the correct application is determined and then the event is marshalled and dispatched over IPC.

At the point where the event is ready to be marshalled and sent across process boundaries, I have replaced the dispatching. Android uses a *Parcel* class to convert data into a transmissible form. A Parcel is a dynamically-sized array container used for packing data to be sent over Binder. It provides functions to place primitives and Java built-in classes (such as Java Strings) into the array container. The Java implementation of a Parcel mostly enters native code using JNI to perform C++ calls on the C++ Parcel class.

My implementation also uses the Parcel class for marshalling, however the Binder call has been replaced with an OKL4 equivalent function that passes the IPC capability, and the Parcel object to native code, where the Parcel object is written into the IPC message registers before being dispatched to the event loop on the other side. It should be noted that the Binder framework transmits the service interface name as a UTF-16 Java string, which uses two bytes per character. The full name of the interface used to dispatch input events is *android.view.IWindow*. Including a null terminator this string is 21 characters in length, and alone consumes 11 of the available 31 OKL4 IPC message registers (there are 32, but one is for the message tag header). Combined with the *MotionEvent* data which consumes 21 message registers, this was one word too many. To make the data fit, the interface was shortened by one character to form *android.view.IWindo*. This is a limitation of OKL4 IPC, where the use of shared memory is encouraged for larger transfers. The modification is not ideal for the purposes of benchmarking and comparison, however the performance difference of transferring 2 bytes is unlikely to have a noticeable influence on the results. A complete implementation, making use of shared memory, is future work.

Like with the InputDeviceReader, various portions of the InputDispatcher had to be removed to get it up and running on my system. This included calls to power management, battery statistics, and a large section of code that determines the destination of the input event. In my system I had only one client application to send events to. I also had to remove a feature that stores event history within a MotionEvent, the class that represents touch events. Extra events are stored together to avoid them being dropped, but investigation into this matter revealed this occurs rarely on Android and the sending of multiple events together can be safely ignored.

Chapter 5

Evaluation

The focus of the thesis up until now had been on porting enough of Android to be able to run unmodified Android applications so that comparisons could be made between the two systems. Full system benchmarks such as browsing the web would best stress an Android system as it requires many subsystems working together. However, this requires a full system implementation which I did not have time for. Instead, I performed some micro-benchmarks and subsystem benchmarks of the input framework.

5.1 Improving Performance with OKL4

My objectives were to investigate in what areas OKL4 can improve over Linux, and to compare real-world performance of OKL4 and Linux in Android. The input framework is a good candidate for microkernel-based optimisation because data needs to be moved from the driver, to a subsystem component, and then finally to a user application. A fast IPC mechanism can improve performance in this area. OKL4 also offers the ability to experiment with different page sizes.

5.1.1 Input Driver Framework

Section 4.1.8 introduced two designs for delivering input events to the System Server component. Both designs aimed to reduce the processing time it takes to retrieve touch events compared the implementation on Linux. This is of interest because early investigation revealed touch events generate a lot of traffic in the input subsystem.

5.1.2 IPC

As discussed in section 2.1.4, one of Binder's primary roles in Android is to dispatch input events from the System Server to user applications. Improving IPC performance can reduce the amount of processing done by the dispatch stage of input routing, and therefore reduce the overall processing time of an input event. Less processing by the system means there are more cycles available to user applications. It is of interest to evaluate the performance of both Binder IPC and OKL4 IPC in this role.

5.1.3 Superpages

Section 2.2.4 discussed the benefits of superpages and the possibility of having *transparent* superpage support. The benefits of increasing TLB coverage on Android could have a noticeable effect due to the sheer size of Dalvik VM and the Java runtime framework. With less cycles wasted on TLB misses, overall processing time could be reduced resulting in improved performance across the entire system.

5.2 Evaluation Environment

In this section we describe the hardware and software environments used to evaluate the two systems.

5.2.1 Android Developer Phone 1

The Android Developer Phone 1 is available for purchase from the Android Market website after signing up. More information can be found on the Android website [12]. It is powered by a Qualcomm MSM7201A system-on-a-chip (SOC) which has an ARM1136js processor that Android normally runs at 384MHz. It also has an ARM9 processor for the baseband software. All experiments are run using the ARM11 processor which is for the user environment. In terms of processor support for the ARM11, OKL4 and Linux are on a level playing field (discussed in section 2.2.3).

The ARM1136js has separate 4-way 32KiB instruction and data caches. It also features a two-level TLB hierarchy with the first level consisting of two 10-entry fully-associative MicroTLBs implemented in logic for both instruction and data. The second level contains a single Main TLB made up of two memories, an 8-entry fully-associative block and a 64-entry 2-way block. All TLB miss measurements made are Main TLB misses to get an idea of the magnitude of the working set of the benchmark scenarios. The ARM1136js has event counters on-chip that allow you to count cycles, TLB misses and various other events. These event counters were used to obtain the benchmark measurements. From this point on, the use of the term TLB miss will refer to Main TLB misses and not MicroTLB misses. Where both types of TLBs are discussed, their full names will be used to avoid confusion.

5.2.2 Android

Android version 1.5 release 2 was used as a basis for the port to OKL4, and for benchmarking and comparing the two systems.

5.2.3 OKL4

OKL4 was compiled with all performance options switched on and kernel tracing disabled. Dalvik was compiled with profiling disabled.

A performance monitoring framework was placed in the OKL4 rootserver (the main OS thread of control) to be able to take benchmarks across multiple processes. By performing an IPC to the rootserver, applications can mark the start and end points of a measurement. A measurement records either cycle count or Main TLB misses, but exclusively one or the other to reduce the amount of noise in the system. At the end of a measurement the result is stored in a buffer. When the buffer reaches its maximum capacity, the results are printed over the serial cable for collection. This minimises the possible impact on results by not printing each measurement over the serial as it comes.

For simple benchmarks that are taken within one process only and do not need any results stored, the OKL4 kdebug interface can be used to access the event counters without invoking my OS personality.

5.2.4 Linux

The Android source was set to release mode, and to match the changes made to the build in OKL4, Dalvik was also compiled with profiling disabled.

Linux has a built-in profiling framework called OProfile that lets you configure the event counters on the ARM11 CPU. It is not compiled into the kernel with the default build. After enabling it I encountered problems getting the OProfile daemon to communicate to the kernel properly. Instead, I constructed my own framework and placed the code inside the Binder device's ioctl system call. Binder's ioctl is the gateway to all Binder services and is by no means slim. This was not an ideal solution because it introduces extra code into the Binder IPC path which I wished to measure. Ideally an extra system call would have been created, but I ran out of time to pursue this option. However, I performed preliminary benchmarks of Binder IPC to confirm that there was no measurable difference as a result of the extra branch condition.

Similar to the OKL4 framework, applications can mark the start and end of measurements. Once the benchmark is over, a second application is used at the command line to copy the results out of the kernel. This helper application can also toggle the type of measurement taken between cycle count or Main TLB misses. For long-running benchmarks where the performance monitoring framework would have little impact on the results, both cycle count and TLB misses were measured together.

5.3 Benchmarks

5.3.1 CaffeineMark 3.0

Before making comparisons between the systems, a sanity check was performed by running the CaffeineMark 3.0 JVM benchmarking suite. The purpose of this was to ensure the performance of Dalvik was similar on both systems. Using the same hardware, and Dalvik VM implementation, but a different OS, it can be expected that performance should be similar. Wild variations could be attributed to serious issues with my ported system that would need rectifying before making comparisons with Linux. It evaluates performance by running the following series of tests:

Sieve	Uses the *Sieve of Eratosthenes* algorithm to find prime numbers.
Loop	Uses sorting and sequence generation to measure compiler loop optimization.
Logic	Tests the speed with which the VM executes decision-making instructions.
String	Measures memory-management performance by constructing large strings.
Method	Uses recursion to see how well the VM handles method calls.
Float	Simulates a 3D rotation of objects around a point.

While running this benchmark a phenomenon was discovered whereby the number of loaded Java libraries affected the String score substantially. As a result, multiple runs were taken using two and then five libraries.

CaffeineMark was modified to run on my OKL4 system by redirecting the results to stdout. This required modifying the Java framework, as normally results are displayed on the screen. Dalvik was also instrumented to load CaffeineMark's *onCreate* function which runs the test. On Linux this benchmark was carried out by launching the application from the menu, and then at the command line using the same instrumentation and modified Java libraries as used on OKL4. Each benchmark was run 5 times to minimise variance.

5.3.2 2D Drawing with Superpages

One advantage of building an OKL4-based OS from the ground up is that superpages can be taken advantage of very easily. The ARM11 CPU in the ADP1 has 72 TLB entries in the main-TLB. A system using a 4KiB page size has a TLB coverage area of 288KiB. Using a larger page size can help increase this significantly to reduce TLB misses to improve performance. For example, the TLB coverage area using a 64KiB page size is 4.5MiB and can accommodate much larger working sets.

The video subsystem of Android uses a compositing window manager [1] and this means each application on the display has its own pixel buffer. In addition to this, images used in applications also consume a lot of memory. A background image or frame-buffer using the native resolution and pixel format of the ADP1 (480x320, RGB565) is 300KiB in size. This buffer alone exceeds the TLB coverage area. As a result, drawing a 2D scene will effectively flush the TLB at least once, and most likely multiple times for more complex applications.

I made modifications to the pixel buffer allocation code in the Android graphics framework to make use of malloc_64k (discussed in section 4.1.2). The Lunar Lander application was used as a basis for this experiment. It is launched and kept at the start up screen drawing a background image, a rocket ship, and a fuel bar. Physics are disabled and the game loops continuously redrawing the scene. Figure 5.1 displays this scene.

The number of cycles and TLB misses were measured on separate runs to minimise noise, and 128 measurements are taken of each. Measurements on drawing a single game frame were taken using a 4KiB page size for the entire system, followed by measurements when utilising 64KiB pages for large pixel buffers only. In the Lunar Lander application there were two large buffers of size 300KiB, one for the background image and one for the application's window. The application's window is copied to the frame-buffer between

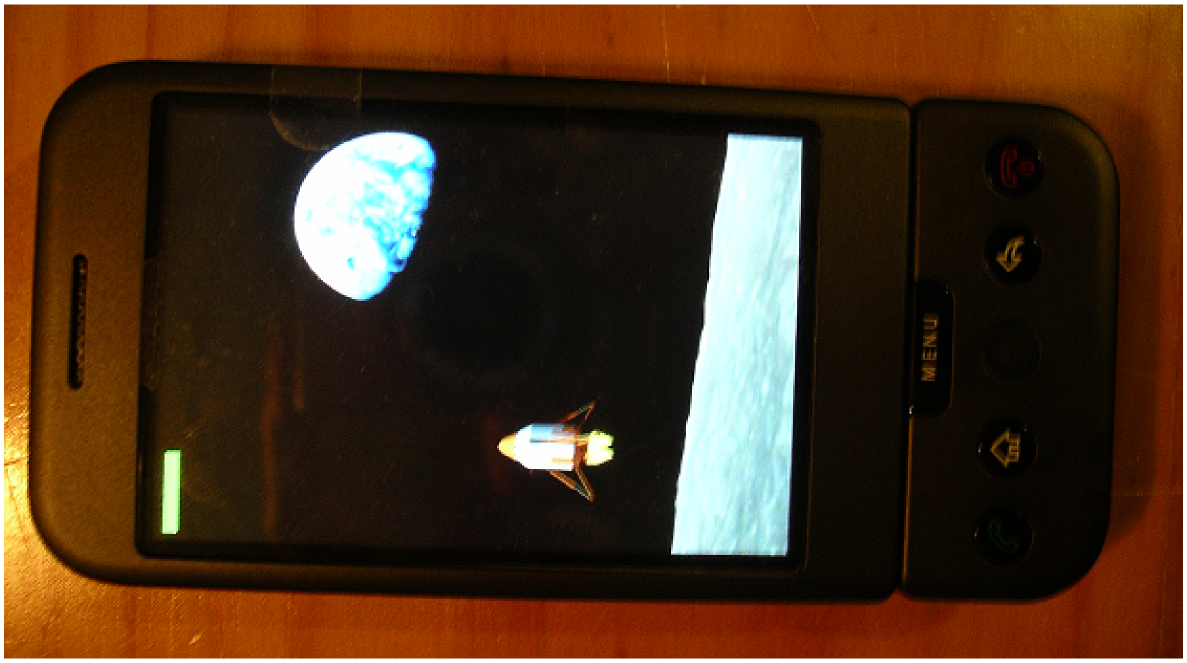


Figure 5.1: The Lunar Lander application.

measurements to visually verify the scene is being drawn correctly, and this is kept consistent for both runs.

5.3.3 IPC Micro-benchmarks

IPC is used in the input framework to dispatch input events to the user application (discussed in the next section). With OKL4 IPC being a minimal implementation of a fast IPC mechanism, and Binder providing a lot more functionality, performance of the two are likely to be very different. Therefore it is important to measure the isolated performance of both IPC mechanisms.

It needs to be considered that IPC performance on native code will be different than on Java code which needs to use the Java Native Interface (JNI) framework to access the native IPC functions. As Android applications all run on Java, it is useful to know the performance penalty on IPC through Java JNI compared to native code. Micro-benchmarks also present a best-case performance figure that can be used to estimate how much of an improvement is possible on the current implementation of IPC on Java.

Binder's framework allows you to set up a simple remote procedure call (RPC) service. This allowed me to create a simple ping-pong service to measure the round-trip-time for an RPC. The client sends an integer to the server, who simply sends the same value back. This is repeated thousands of times in a tight loop to minimise noise and variance in the results. The same scenario was constructed on OKL4 using OKL4 IPC.

The Binder driver API is incredibly complicated and makes it difficult to measure the raw IPC performance, as the kernel can schedule any number of arbitrary tasks between send and receive. Android provides a C++ and Java framework API to make it easier to use Binder. The API allows you to register a remote procedure call (RPC) service with the system Service Manager. Client applications request a handle to the service based on the service name, and can then perform RPCs on that service.

An RPC service is one layer of abstraction above delivering raw uninterpreted data, and data needs to be interpreted in a real environment to make progress. Therefore, all IPC benchmarks are based on the simplest server/client RPC scenario of executing a ping-pong.

All tests on Binder and OKL4 IPC are carried out as follows:

1. Set up the server
2. Client executes a test ping to check the server is alive
3. Client executes the ping-pong loop (thousands of iterations)

The last step is wrapped within performance monitoring functions that measure cycle count and main-TLB misses. The average round-trip-time is calculated by dividing the total number of cycles by the number of ping pong iterations. One-way IPC times can be estimated by halving the round-trip-time. The average Main TLB misses is calculated in the same way.

Benchmarks using 1 word and 31 words (4 bytes and 124 bytes respectively) for the payload (same size payload sent both ways) were run on both native C/C++ code and on Dalvik using Java. Each benchmark was repeated 5 times to minimise variance. At the server, every transferred word is passed to a ping function that returns the same value before preparing it for the return journey. The client ignores the values of the returned data.

5.3.4 Input Framework

All measurements are based on touch events only, as key events do not occur frequently enough to have much impact on system performance. All measurements were taken while touching the screen. This generates approximately 80 events per second which flow through the input framework. To be able to make fair comparisons between OKL4 and Linux, the input framework has been divided into three parts. They are as follows:

1. First the InputDeviceReader obtains an event from the touch driver.
2. The event is then passed back up to Java code and is processed to determine whether it has some special purpose, e.g. turn the screen back on, or to deliver it to a user application.
3. Finally, the event is dispatched over IPC to the user application.

The above scenarios will now be discussed in further detail.

Obtaining Events

This process is handled differently on OKL4 and Linux. OKL4 using a single system call to wake up before accessing the event data from shared memory. Linux uses a read system call to access the data.

The process of obtaining events from the touch driver is handled differently on OKL4 and Linux. As discussed in section 4.3.1, Linux uses a pair of system calls, poll and read. OKL4 also uses a system call to wait for a notification from the driver, but it does not need the second system call to read the event data. This data is read from shared memory.

Measurements were taken from the point the InputDeviceReader wakes up knowing it has an event to process, to the point right before it returns the event to the Java code. Between these points the event is read from the driver, and converted into the format as expected by Android.

Java Processing

Once the event reaches Java code, there is too much missing in the input routing path of the OKL4 implementation of the System Server to be able to make a fair comparison to the Linux implementation. However, measurements were taken anyway to help create a performance profile of the entire input framework. Measurements were taken from the end of the previous scenario, until the point right before the event is marshalled and dispatched.

Dispatching the Event

Events are marshalled and dispatched to user applications over Binder IPC. Preliminary IPC benchmarks showed that OKL4 IPC out-performs Binder IPC and could help to improve dispatch performance. Measurements were taken from the point right before the touch event is marshalled and dispatched in the System Server, until the point right after the event has been reconstructed and identified as a touch event on the user application side. This is a one-way dispatch.

To determine the cost of marshalling, measurements were taken on the OKL4 system without marshalling the data. Instead, dummy data was dispatched to the user application.

Entire Path

The entire userspace input framework path was measured to be able to determine the effect on overall performance of the results that we are interested in, obtaining and dispatching of touch events. Measurements were taken from the first measured point in obtaining an event, to the last measured point of having reconstructed the touch event in the user application.

5.4 Results and Analysis

5.4.1 CaffeineMark 3.0

	Linux (Menu)	Linux (ADB)	Linux (ADB)	OKL4	OKL4
Libraries	5	5	2	5	2
Sieve	467	466.6	467	490	490
Loop	553	553.2	553	557	557
Logic	397.4	398	397.8	400	400
String	454.8	611.2	633.4	585.4	815.6
Float	342.2	337	340.8	351	356
Method	401	400.6	400.8	404	404
Overall	430.6	451.4	454.8	456	483

Table 5.1: CaffeineMark 3.0 results for both systems.

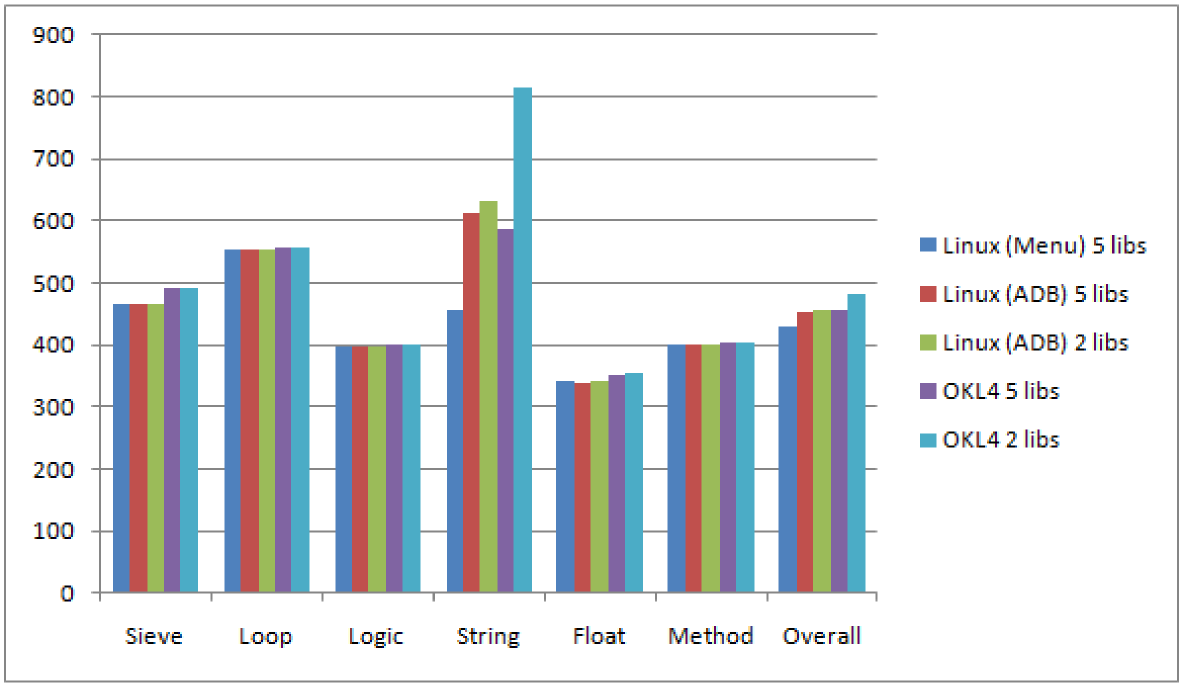


Figure 5.2: Graph of the results in table 5.1.

The label Linux (Menu) refers to launching CaffeineMark from the phone's menus and the label Linux (ADB) refers to launching CaffeineMark at the command line over the Android Debug Bridge (ADB). All results are averaged over 5 iterations.

A phenomenon was discovered that resulted in large variations of the String score depending on how many Java libraries were pre-loaded. Android has five libraries that together form the Java runtime framework. However, for the purposes of CaffeineMark 3.0, only two libraries were required.

Initial results used only two libraries and this showed much better performance in the String score, which stresses the memory-management subsystem of Dalvik by constructing large strings. The standard method of launching applications from the phone's menus showed the worst performance with a score of 454.8. For this reason I re-ran the benchmark at the command line using the same modified libraries I used on OKL4, as to minimise variance in the two setups. This improved the String score by over 30% compared to launching the application from the menus. The score achieved by OKL4 was much higher again. Due to the nature of the String test stressing the memory-management subsystem, it suggested that the memory allocator was performing worse when heap utilisation was higher, but it required further investigation.

After re-running the benchmarks using all five Java libraries, the String score dropped for both OKL4 and Linux, indicating that the number of Java libraries loaded affects the performance of the memory-management subsystem. Using all five libraries also brought all benchmark scores within close proximity to one another, and this convinced me there were no serious issues with my implementation. The Sieve score was consistently better on OKL4 by about 7%, but I was unable to determine why. The workload of the input framework is quite different to the type of tests performed by this benchmark. It does not execute a tight-looped algorithm for finding prime numbers, and does very little memory allocation, thus the variances observed in these results should not adversely affect the input framework benchmarks.

The results of this benchmark show overall performance with 5 libraries to be very similar, meaning that all further benchmarks should be carried out using all five libraries. The phenomenon exists on both systems and definitely requires further investigation, but I did not have any more time to spend chasing this down.

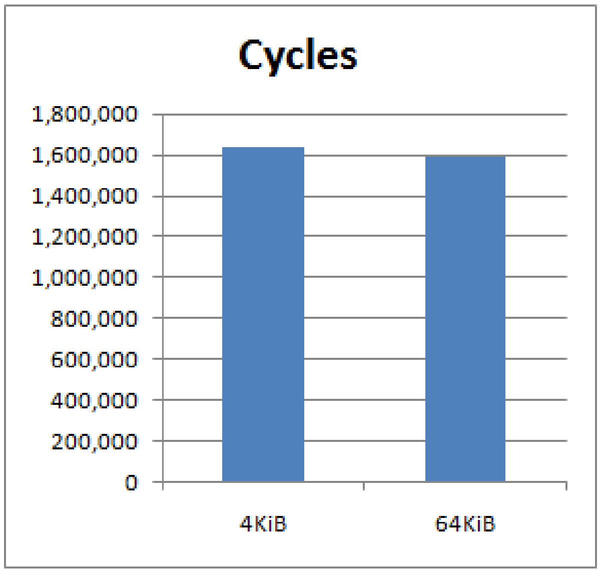
5.4.2 Superpages

Page Size	4KiB	64KiB
Avg Cycles	1,633,980	1,594,831 (-2.4%)
Avg Time(μs)	4,255	4,153 (-2.4%)
Avg TLB Misses	489	297 (-39%)

Table 5.2: Benchmarks of Lunar Lander using a different page size for large pixel buffers.

Page Size	4KiB	64KiB
Cycles	3265 (0.2%)	3619 (0.23%)
Time(μs)	8.50 (0.2%)	9.42 (0.23%)
TLB Misses	2.42 (0.49%)	9.45 (3.18%)

Table 5.3: Standard deviation of the results in table 5.2



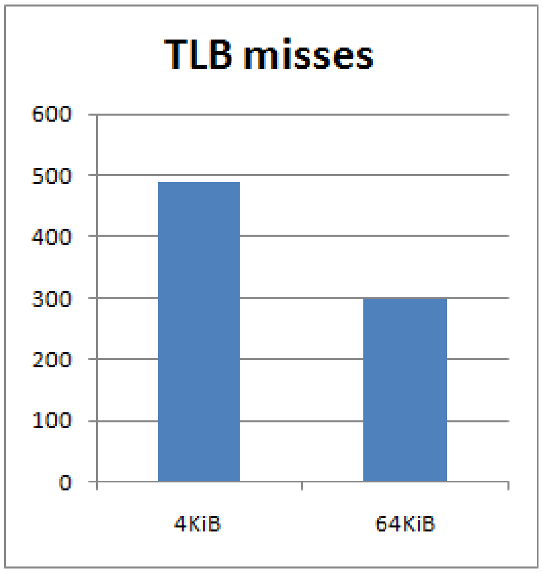


Figure 5.3: Graphs of the results in table 5.2.

As expected, the number of TLB misses was reduced dramatically. However, the effect on performance was only minimal. It is very likely that performance is being bottlenecked in the data cache. Reading and writing from large buffers as is done in the test, will be effectively flushing the 32KiB data cache several times over just to draw the background, which requires accessing 600KiB of data sequentially. Without measurements of the data cache misses, and the cost of a data cache miss, I can only speculate. However, there is still some useful information that can still be drawn from the results.

Given that the *only* difference between the two benchmark runs was the page size for two buffers, a rough estimate of the cost of a TLB miss can be calculated. The standard deviation as shown in table 5.3 is extremely low for all measurements, but TLB misses when using a 64KiB page size does show a little more variance than the others.

There are 192 less TLB misses and there is a saving of 34,149 cycles (102µs) in processing. This suggests the cost of a TLB miss is 204 cycles (531ns). Accounting only for the two larger standard deviations in the results (TLB misses), this places the TLB miss cost between 192 and 217 cycles which is a narrow range.

To confirm this calculation, I measured the cost of uncached memory reads. This was done by measuring the number of cycles it takes to perform 4 reads to uncached memory in a tight loop for 16384 iterations. There are 4 reads in the loop to amortise the cost of the branch instruction. The measured cost of a single read to uncached memory was 95 cycles.

A TLB look-up on the ARM11 CPU first begins by accessing one of the MicroTLBs. On a MicroTLB miss, a look-up is performed on the Main TLB. A second miss results in a walk of a two-level page-table by the hardware. This walk requires two uncached memory reads, and based on the calculated cost of one of those reads, the total cost is 190 cycles. However, the miss latency of the look-up on the Main TLB from a MicroTLB is still unaccounted for, and is simply mentioned to take a variable number of cycles in the ARM11 ARM Reference Manual [9]. It is likely the remaining 14 cycles can be attributed to this process. This confirms that my calculated complete TLB miss cost of 204 cycles is likely to be near the real cost.

5.4.3 IPC Micro-benchmarks

	OKL4	Binder	OKL4	Binder
Iterations	16,384	16,384	16,384	16,384
Payload (bytes)	4	4	124	124
Avg Cycles	1,592	93,053	2,146	106,663
Avg Time (μs)	4.15	242.33	5.59	277.77
Avg TLB Misses	3.59	71.26	3.23	73.56

Table 5.4: C/C++ IPC ping-pong micro-benchmarks.

	OKL4	Binder	OKL4	Binder	JNI
Iterations	16,384	8,192	16,384	4,096	8,192
Payload (bytes)	4	4	124	124	124
Avg Cycles	16,753	284,349	30,700	725,748	379,456
Avg Time (μs)	43.63	740.49	79.95	1,990	988.17
Avg TLB Misses	57.74	500.64	61.43	1,546	841.06

Table 5.5: Java IPC ping-pong micro-benchmarks.

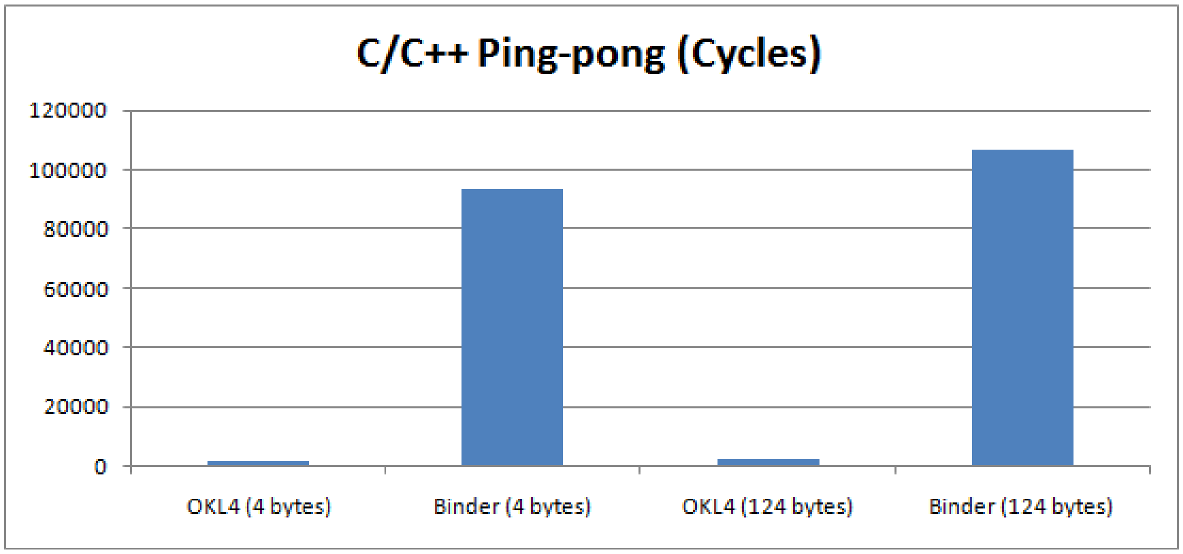


Figure 5.4: Average cycles for a two-way ping pong on C/C++.

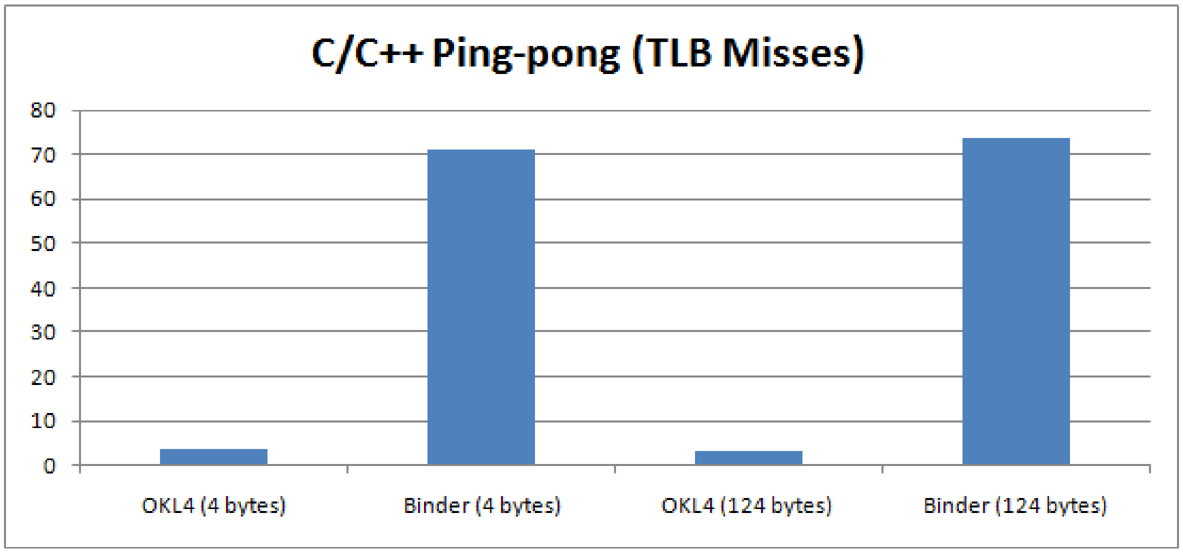


Figure 5.5: Average TLB misses for a two-way ping pong on C/C++.

Analysis of C/C++ IPC Micro-benchmarks

The results in table 5.4 show that OKL4's IPC mechanism is much faster than Binder IPC. Binder is 58 times slower in sending and receiving a 4 byte payload, and 49 times slower using OKL4's maximum payload (124 bytes).

Why is OKL4 IPC is much faster than Binder IPC? Firstly, Binder is just another device in the file system, and the ioctl system call for the Binder device is overloaded to handle all calls to Binder. In this design there is no IPC fastpath through the kernel

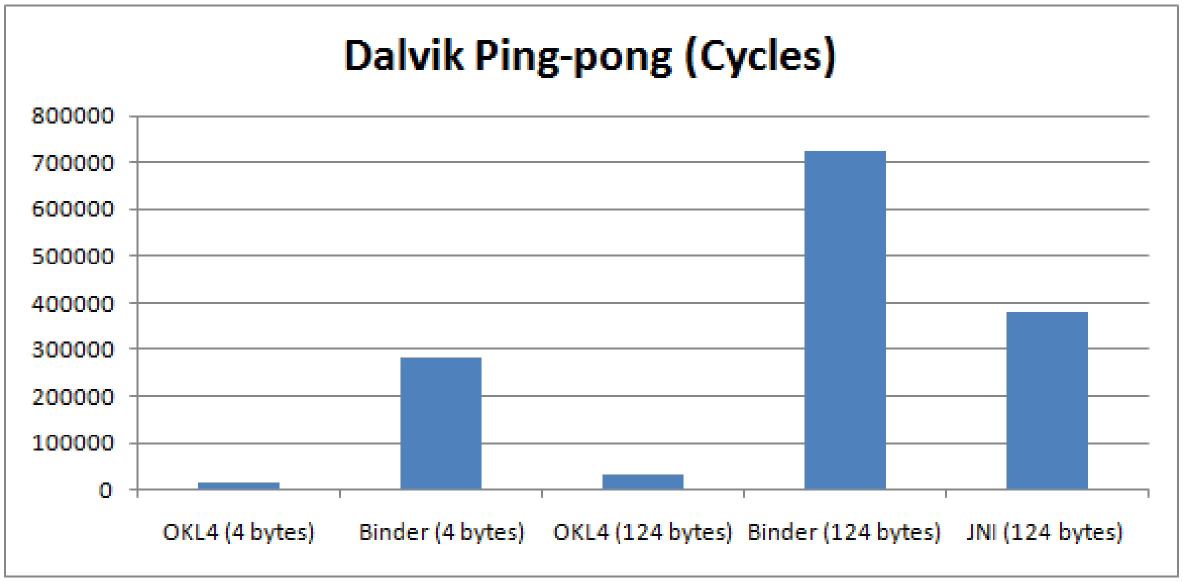


Figure 5.6: Average cycles for a two-way ping pong on Dalvik.

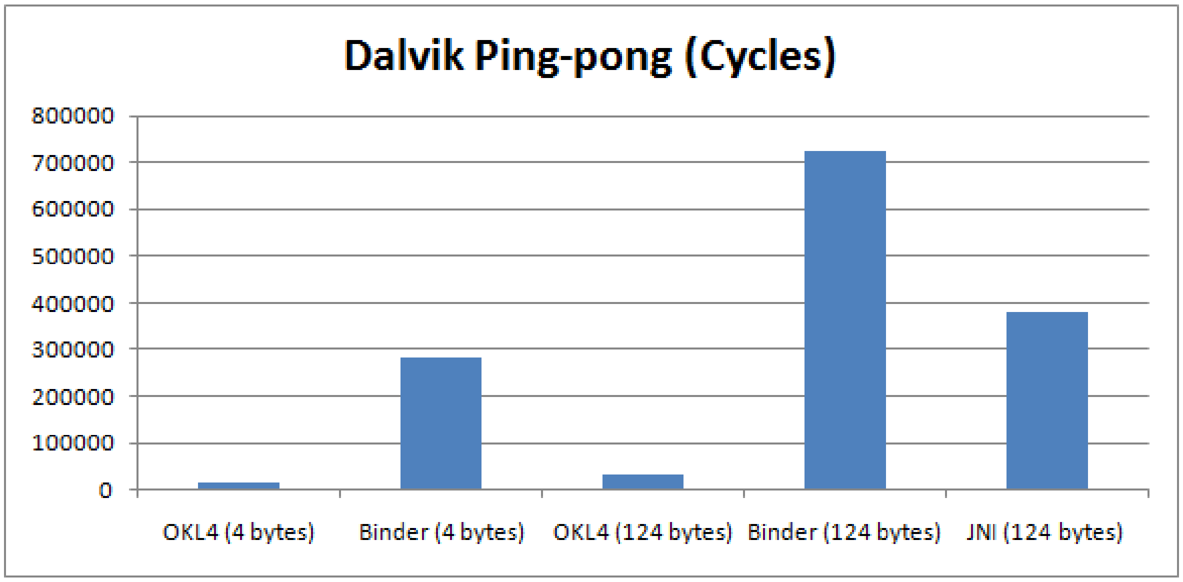


Figure 5.7: Average TLB misses for a two-way ping pong on Dalvik.

and this really limits the maximum performance of Binder IPC.

A call to Binder's ioctl is a series of commands that is interpreted by the Binder driver. The IPC command is called BINDER_WRITE_READ (BWR) and signals to the driver that the user process wants to do a write, or a read, or a write followed by a read. This path is written entirely in C, and this command could be one of several in a row.

Furthermore, I made an interesting discovery that could affect performance in a large way when using BWR to do a write followed by a read. Each ping-pong was taking double the number system calls than required. Investigation into the issue revealed the following:

1. The write phase generates a TRANSACTION_COMPLETE (TC) message that is appended to the calling thread's own job queue.
2. During the read phase immediately following, this TC message is read and returned back to userspace.
3. The userspace Binder framework ignores this message, notices it has not yet received the reply it was looking for, and initiates another BWR to restart the read operation.

All TC messages in Android appear to be ignored, yet removing the generation of the TC message causes the system to fail to boot. It's hard to say whether this is part of the design or a bug, or perhaps it is a remnant from the original OpenBinder implementation. Whatever the reason, this occurrence contributes to Binder's poor performance.

The Binder implementation is also programmed to allow multiple user applications to call into Binder at the same time. A global Binder lock must be acquired to make progress. Threads that are waiting on a read operation release the lock, and wait in the driver to be woken where they reacquire the lock and continue. Finally, the location of a user process's payload can be anywhere in the user's address space and of unpredictable size. In contrast, the OKL4 IPC implementation does have an IPC fastpath, it is not multi-threaded, and accesses user data from a predetermined location (the thread's user thread control block, or UTCB). Binder offers not much in terms of optimising IPC speed, and perhaps this was never a design goal of the developers. Interestingly, the Android documentation markets Binder as a *"lightweight remote procedure call mechanism"* [13], but did not use it to implement a truly component-based system.

The lack of an IPC fastpath can explain why TLB miss rates on Binder are much higher than OKL4, effectively flushing the TLB once per iteration. This has a large negative impact on Binder's performance. Based on the calculated cost of a TLB miss from section 5.4.2, the cost of 71 TLB misses is 14,484 cycles, or 15.6% of the processing time. This is quite a large portion of the processing spent handling TLB misses. The use of an IPC fastpath in OKL4 means it can minimise the amount of text and data touched to reduce TLB misses.

Analysis of Java IPC Micro-benchmarks

It should first be noted that iterations were varied in these benchmarks to prevent the cycle counter from overflowing and complicating the measurements. Several thousand iterations is still plentiful to normalise variance. The results are also orders of magnitude apart which makes noise less of an issue.

The OKL4 IPC results on Java required implementation of a JNI interface to OKL4 IPC. The payload was represented by an integer array, to mirror the OKL4 IPC message registers, and to minimise the number of JNI calls required to perform IPC. The reason for doing this is described at the end of this section.

In the C/C++ benchmark, Binder took an extra 35μs when moving from a 4 byte payload to 124 bytes, but on the Dalvik VM it increased by over 1ms. The change in payload size does not account for the difference on Dalvik. The only other change is the use of JNI to fill up the payload of 31 integers. Using a Parcel, the standard Java class used by Binder to send IPC data, one JNI operation is required to load each integer for delivery. This is also the case for integer arrays where each integer is still loaded one-by-one. The last column of table 5.5 measures the cost of all the payload JNI calls when using a 124-byte payload. This benchmark was conducted only on Linux due to time constraints. At each iteration the following is performed, two sets of 31 integers are written through JNI, and two sets of 31 integers are read through JNI, and two calls through JNI are made to reset the Parcels for the next loop. This is a total of 126 JNI calls, and takes almost a millisecond to perform. A rough estimate of the cost of a single call can be calculated by dividing the total time by 126. This results in 7.84μs per JNI call, which is longer than the round-trip-time of a 4-byte ping pong using OKL4 IPC on C.

From these results it is clear that Java JNI has a devastating effect on IPC performance. The number of JNI calls for the OKL4 benchmark was minimised to only one. The performance benefit of doing this is highlighted with OKL4's 124-byte round-trip-time on Java outperforming Binder on C++ using a 4-byte payload.

The TLB miss rate while performing JNI is also incredibly high. Binder in C++ already places considerable stress on the Main TLB, effectively flushing it once per iteration. Together with Dalvik and JNI, this appears to be the cause of the sharp drop in performance for the 4-byte Binder round-trip-time on Java. Again using 204 cycles as the cost of a TLB miss (from section 5.4.2), 500 TLB misses costs 102,000 cycles, or 35.8% of the processing time. This also explains why the 4-byte round-trip-time on OKL4 did not experience as large of a performance drop as Binder.

5.4.4 Input Framework

For the following tables, the first number presents the average and the second number in brackets is the standard deviation.

	OKL4	Linux
Cycles	3,119 (710.17)	11,825 (9,376)
Time(μs)	8.12 (1.85)	30.79 (24.42)
TLB Misses	2.22 (1.17)	7.93 (1.64)

Table 5.6: Results for obtaining an event.

	OKL4	Linux
Cycles	159,952 (11,722)	797,831 (84,361)
Time(μs)	417 (30.53)	2,078 (220)
TLB Misses	200 (8.66)	1,409 (33.65)

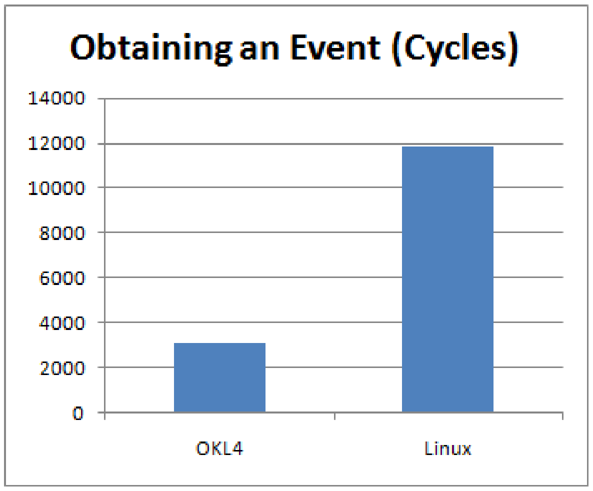
Table 5.7: Results for the Java processing stage.

	OKL4	OKL4 (no marshalling)	Linux
Cycles	237,868 (22,736)	96,934 (12,329)	291,365 (49,723)
Time(μs)	619.45 (59.21)	252.43 (32.11)	758.76 (129.49)
TLB Misses	405.65 (28.84)	149.69 (5.7)	424.41 (11.33)

Table 5.8: Results for the IPC dispatch stage.

	OKL4	Linux
Cycles	398,687 (32,457)	1,102,831 (100,692)
Time(μs)	1,038 (84.52)	2,872 (262.22)
TLB Misses	603.63 (32.68)	1,847 (26.69)

Table 5.9: Results for the entire input path.



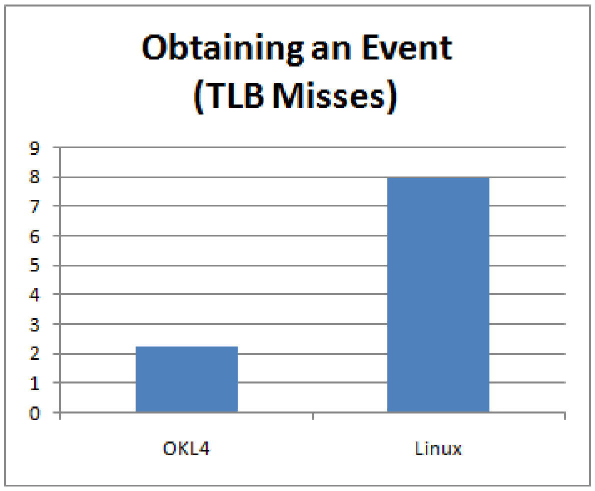
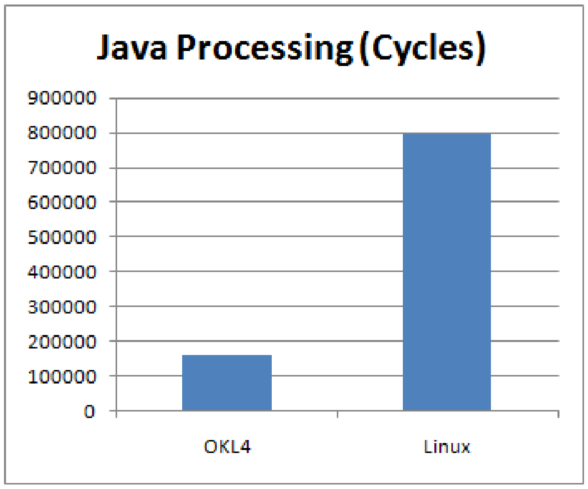


Figure 5.8: Graph of the averages from table 5.6.



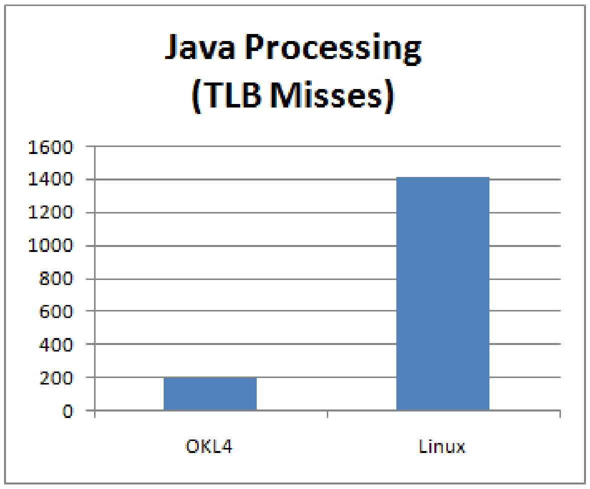


Figure 5.9: Graph of the averages from table 5.7.

Analysis of Obtaining an Event

The only real difference between the two implementations is that on Linux a read system call is used, whereas the OKL4 implementation uses shared memory. Without even running these benchmarks it is obvious that reading from shared memory is much

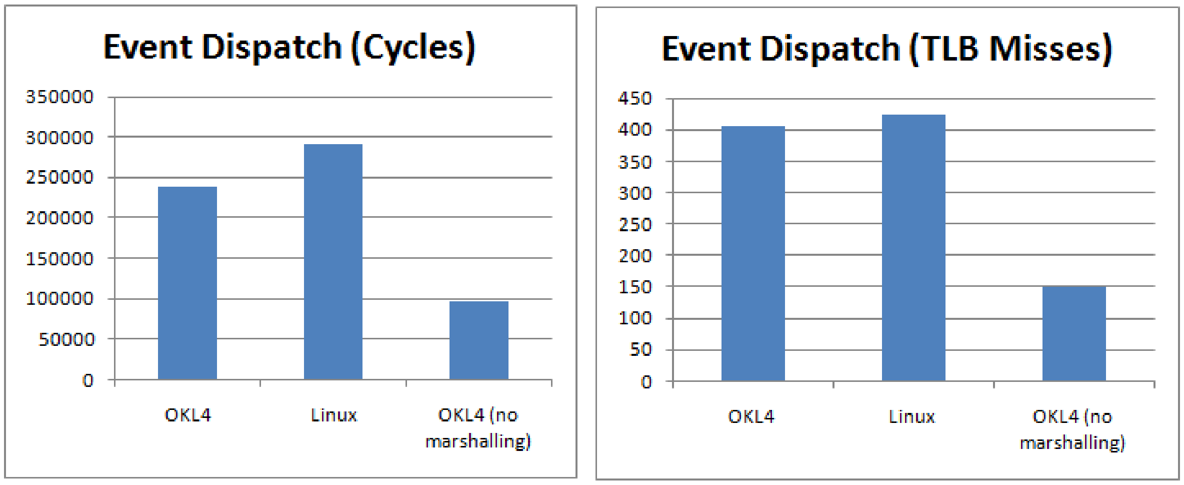


Figure 5.10: Graph of the averages from table 5.8.

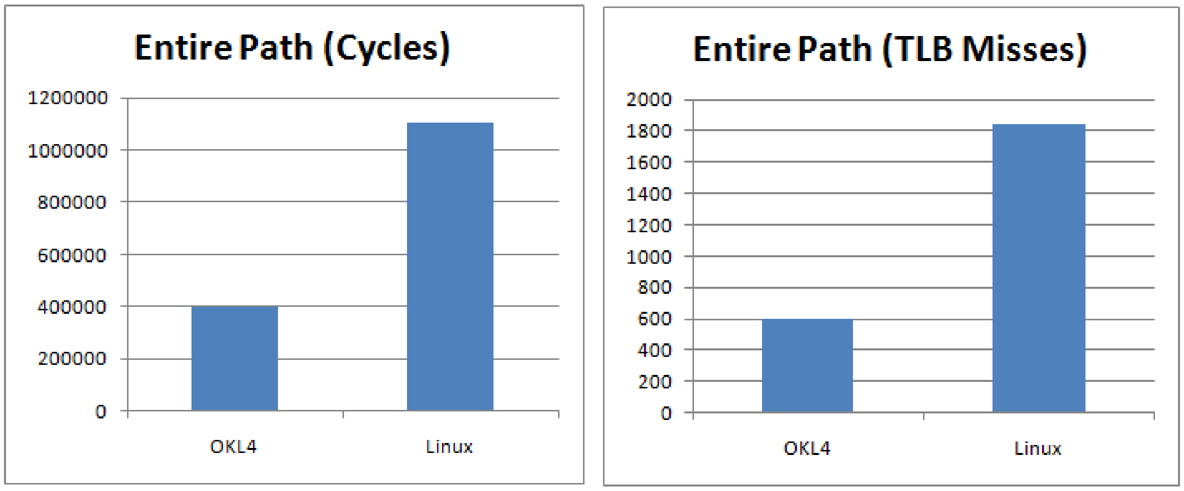


Figure 5.11: Graph of the averages from table 5.9.

faster than performing a system call.

However, as discussed in section 4.3.1, a touch event consists of multiple sub-events. On Linux there are 2 to 5 sub-events depending on the interleaving being applied. This means that before a touch event is generated in the Java code and propagated through the system, this stage of reading an event from the driver is performed multiple times. However, the benchmark measures the cost of reading a single sub-event and not a whole event. The saving on OKL4 is much more substantial if the results are multiplied to cover a whole touch event, but this does not provide any useful data due to the large variance observed in the Linux measurements. The code being measured was native C/C++ and extremely small. It is more than possible the performance monitoring framework was influencing the results.

To properly measure the input cost, another benchmark needs to be run that takes measurements until a complete touch event has been read. This also needs to include the processing done by the operating system to process the interrupt and wake up the user process. Unfortunately, I did not have further time to carry this out.

Looking at the rest of the results, especially for the entire input path (table 5.9), it is obvious that this stage of the input framework is not the bottleneck. Improvements here will make little difference to the overall processing time.

Analysis of Java Processing

As discussed in section 5.3.4, the implementations on OKL4 and Linux vary, and so no fair comparison between the two systems can be made here. However, the large difference in the results is quite interesting. Android normally takes 2ms to process a touch event, but the OKL4 implementation does it in 417µs. The steps in this stage include:

1. Checking whether the event has a special purpose (but touch events always go to user applications when the display is on).

2. Notifying the power management service that there is user activity happening.

3. Logging the event with the battery statistics service (I did not have time to investigate the purpose of this).

4. Placing the event onto a shared queue, and notifying another thread.

5. Reading the event off the queue and examine what type of event it is.

6. Determining which process to send the event to.

My implementation included only stages 1, 4 and 5. This means the other parts of this stage are very likely to be responsible for the factor of 5 slow-down experienced on Linux. The number of TLB misses is a factor of 7 times more, and this will definitely be contributing to the long processing time. Further profiling is required to determine the true source of the issue, but JNI is likely to play a part. As observed in the IPC micro-benchmarks, heavy use of JNI destroys performance.

Analysis of IPC Dispatching

The IPC micro-benchmarks more or less present best-case performance figures, but are no indicator of real-world performance. The IPC dispatch results in table 5.8 are benchmarks of real-world performance. As expected, OKL4 IPC outperforms Binder IPC. However, the large differences shown in the micro-benchmarks is now but a small gap in a real application. Knowing beforehand how expensive JNI can be, an extra set of results was taken on OKL4 where dummy data was dispatched and no marshalling was performed. This was done to be able to differentiate the cost of IPC from the cost of JNI.

This extra data set again shows that JNI is responsible for the majority of the processing time and TLB misses. In light of this, OKL4 is still 139µs faster than Binder. Unfortunately, this saving is mostly dwarfed by the overall processing time of the entire input path. Replacing Binder IPC with OKL4 IPC will improve the overall time by about 5%. This is still a sizable result, but hardly comparable to the results of the micro-benchmarks.

It is possible however to achieve near the performance of the results that do not do marshalling. A custom JNI interface can be written to perform the entire marshalling operation in one go in native code. This would yield roughly a further 12.8% increase in performance by removing the JNI marshalling cost. This optimisation can be applied to work around the slowness of Dalvik for all critical paths.

Interestingly, there have been complaints on the Android Developers Google group about touch on Android destroying game performance [14]. Applying the above optimisation could help reduce the impact of touch processing on the system. Android is addressing this issue for future releases by throttling the frequency of touch events to 35 per second.

5.5 Conclusions & Discussion

The goal of this thesis as outlined in section 1.2 was to port sufficient parts of Android to OKL4 to be able to make system comparisons to Linux. The outcome of this goal was to determine not only whether a complete port of Android is feasible, but whether it is worthwhile. Continuing to port Android to OKL4 for no gain in performance is a rather pointless exercise.

Throughout the thesis it was presented that OKL4 could improve the performance of Android in several areas. They were the following:

- Memory-usage
- IPC performance
- Exploitation of superpages

Measuring the memory-usage of my incomplete port provides little indication as to the usage of a complete port. Therefore this could not be investigated further than speculation about OKL4's minimality principle.

IPC performance on the other hand was investigated thoroughly due to Android's componentised design with user applications communicating with another userspace process to access system services and devices, including input, video and audio. The cost of IPC on Android for user applications was shown to be incredibly expensive, and OKL4 IPC was shown to be much faster. However, the performance impact of using an interpreted user environment (Dalvik) renders any performance improvements to be minimal. The JNI implementation to access native code is amazingly slow, and the size of Dalvik applies a lot of pressure on the TLB.

Unless Dalvik is improved substantially to be faster and smaller, for example by making use of just-in-time compilation, the performance of the operating system underneath will not have a large impact. The actual IPC process for OKL4 on Dalvik accounts for at most 41% of the processing time, with the rest spent marshalling the payload for delivery.

In Dalvik's current state, another way to improve performance is to exploit superpages. My experiments show that a lot of time is spent handling TLB misses. I was unable to implement transparent superpages and therefore could not measure the impact this would have on all of the presented benchmarks. However, I believe it would definitely have a very noticeable effect on overall system performance. With Linux's lack of transparent superpage support, OKL4 has the advantage.

The CaffeineMark 3.0 benchmark showed that raw Dalvik performance on OKL4 is on par with that on Linux. IPC performance of OKL4 was shown to be much faster, but Dalvik performance mitigates this benefit. Through further work, the benefits of superpages on overall Android performance can be measured. Previous work by Peng, Lueh, Wu, Gou and Rakvic [15], showed the use of 64KiB page and 1MiB pages improved Java virtual machine performance by 9 to 48% and 24 to 48% respectively for a SPECjvm98 workload. If similar results could be reproduced for standard system workloads on Dalvik, an OKL4-based Android would quite definitely be worth pursuing.

Finally, it should be noted that if Android were to undergo proper componentisation, by isolating unrelated services from one another, the increase in reliance on IPC would make OKL4 the better candidate for a more trustworthy platform.

Chapter 6

Future Work

6.1 Limitations

My OKL4-based system has several limitations that need to be overcome to further facilitate the porting process.

6.1.1 File System

Programming a boot image onto the ADP1 is done over a USB connection. The use of an in-memory file system means that all files have to be transferred on every development cycle. Initially the size of the file system was small, and the transfer completed quickly. However, the in-memory file system peaked at approximately 60MiB in size by the end of the thesis. This not only increased the compile time to package these files into the boot image, but it also increased the transfer time. As a result, the compiling and booting process now takes over a minute and hampers development efforts. This problem will only become worse as the number of files increase. There is also an issue when the file system size exceeds the ADP1's memory capacity. Therefore it would be desirable to make use of the SD card device to relieve these problems.

6.1.2 Dynamic Linker

Currently the system is compiled using static linking only. Most Android libraries are configured to produce shared libraries. One particular issue is when Dalvik VM loads a shared library at runtime, it searches for and executes a special function to register native functions with JNI. I had to work around this by manually calling this special function for each library in use. This was an acceptable solution for the short-term, but dynamic linking is a necessity in the long-term.

6.1.3 C++ Support

C++ support in my build system and ELF loader is non-existent, and C++ static initialisers are not being executed. This is a serious issue with a large amount of Android libraries being written in C++ and results in obscure page faults at runtime. In my thesis I consistently had to work around this by manually inserting code to perform such initialisations when necessary. It wasted a lot of my time figuring out that this was the problem behind many mystery bugs.

In the following example the variable *foo* is not initialised in my system because it's constructor is never being called. However, *foo* is meant to be constructed with the default constructor.

```
class C {
public:
   int val;
   C(void) {
      val = 5;
   }
};

static C foo;

int main(int argc, char** argv) {
   printf("%d\n", foo.val); // foo.val is not 5
   return 0;
}
```

6.2 Additional Features

A complete port of Android to OKL4 offers many possibilities for future work. Some of them are discussed below.

6.2.1 Transparent Superpages

As discussed in section 2.2.4, the trade-off for the performance of larger page sizes is that internal fragmentation is likely to be larger and consume more memory. My experiments have shown there is a lot of pressure on the TLB in Android. Moving from a 4K page size to a 64K page size could improve overall performance significantly, and reduce power usage as a result of having to process an order of magnitude less TLB refills. The ADP1 also has roughly 96MiB of memory available to the user environment. By inspection of the default build running on the ADP1, almost half of that is being used to support the file cache in Linux. This is a lot of spare memory, and there is a trade-off to be investigated between memory-usage, and performance and power. Further work in this area would yield valuable data on how to improve the performance of Android.

6.2.2 Binder

Binder is used in many of Androids subsystems. Both video and input require it, and user applications use it to communicate to the System Server. A complete port without Binder would not be feasible.

An implementation of Binder on OKL4 would require a monolithic *Binder Server* component in the system. Binder's duties involve:

- registering and allowing access to RPC services
- reference counting of Binder objects (capability to an object provided by a service) that are passed through Binder
- sharing memory between processes by passing around file descriptors

Registering RPC services and requesting access to them can be handled by the Binder Server, by keeping a global list of registered services and returning unique handles to users. In the general case, communication between a client and service can be direct. However, if a reference-counted Binder object is being distributed to another process, the message must pass through the Binder Server to be able to process the reference-counting. This incurs extra overhead of having to double the number of IPCs to transfer a Binder reference. I have observed that distributing access to Binder references is not a critical operation that occurs frequently. Getting access to a service usually happens once. Using the service is more common and in the proposed implementation this communication is done directly.

Finally, the Binder Server in a component-system would not have the necessary capabilities to create and distribute shared memory. An appropriate interface is needed with another component that does have such capabilities, such as the rootserver in my current OKL4 system.

6.2.3 Zygote

With a Binder implementation, porting the System Server should be a much smoother process. The next step is to facilitate launching new Dalvik VM instances. This is achieved with the Zygote process as discussed is discussed in section 2.1.3, and requires a fork implementation. OKL4 can support fork through the use of the *ExchangeRegisters* system call, which allows you to read the register values of a thread.

In Android, the Zygote listens on a socket and forks new Dalvik VM instances on request. An OKL4 implementation could simplify this by replacing the use of sockets with IPC, and provide the System Server with an IPC capability to the Zygote.

Appendix A

System Calls

The following function prototypes are the Linux system calls that have been implemented in the Android on OKL4 port.

```
int __futex_wait(volatile void* ftx, int val, const struct timespec* timeout);
int __futex_wake(volatile void* ftx, int count);
void* __brk(void* addr);
pid_t gettid(void);
pid_t getpid(void);
void* __get_tls(void);
int __set_tls(void* addr);
int __open(const char* filename, int flags, mode_t mode);
int close(int fd);
ssize_t read(int fd, void* buf, size_t length);
ssize_t write(int fd, const char* buf, size_t n);
int stat(const char* filename, struct stat* buf);
int fstat(int fd, struct stat* buf);
off_t lseek(int fd, off_t offset, int mode);
int flock(int fd, int operation);
void* __mmap2(void* addr, size_t len, int prot, int flags, int fd, long offset);
int munmap(void* addr, size_t size);
int mprotect(const void* addr, size_t len, int prot);
int __ioctl(int fd, int op, void* data);
int uname(struct utsname* buf);
int clock_gettime(clockid_t clk_id, struct timespec* tp);
int __getcwd (char* buf, size_t size);
int gettimeofday(struct timeval* tv, struct timezone* tz);
int sched_yield(void);
int __getpriority(int which, int who);
int setpriority(int which, int who, int prio);
int __pthread_clone(int(*fn)(void*), void* child_stack, int flags, void* arg);
int prctl(int option, unsigned long arg2, unsigned long arg3, unsigned long arg4,
   unsigned long arg5);
int __fcntl64(int fd, int cmd, void* arg);
int nanosleep(const struct timespec* req, struct timespec* rem);
```

Bibliography

[1] Patrick Brady. Anatomy & Physiology of an Android. `http://sites.google.com/site/io/anatomy--physiology-of-an-android`, 2008.

[2] Matt Richtel. Google: Expect 18 Android Phones by Year's End. `http://bits.blogs.nytimes.com/2009/05/27/google-expect-18-android-phones-by-years-end/`, 2009.

[3] David A. Wheeler. Linux Kernel 2.6: It's Worth More! `http://www.dwheeler.com/essays/linux-kernel-cost.html`, 2007.

[4] Gernot Heiser. Advanced Operating Systems Lecture 1. `http://www.cse.unsw.edu.au/~cs9242/08/lectures/01-intro.pdf`, 2008.

[5] Dan Bornstein. Dalvik VM Internals. `http://sites.google.com/site/io/dalvik-vm-internals`, 2008.

[6] Jesse Burns. Developing Secure Mobile Applications for Android: An Introduction to Making Secure Android Applications. `http://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf`, 2009.

[7] Linux Online - GNU General Public License. `http://www.linux.org/info/gnu.html`.

[8] Carl van Schaik and Gernot Heiser. High-Performance Microkernels and Virtualisation on ARM and Segmented Architectures. *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems*, 2007.

[9] ARM. ARM1136JF-S and ARM1136J-S Technical Reference Manual. `http://infocenter.arm.com/help/topic/com.arm.doc.set.arm11/index.html`, 2009.

[10] LXR linux/Documentation/vm/hugetlbpage.txt. `http://lxr.linux.no/#linux+v2.6.31/Documentation/vm/hugetlbpage.txt`.

[11] Ulrich Drepper. Futexes Are Tricky. `http://people.redhat.com/drepper/futex.pdf`, 2009.

[12] Developing on a Device — Android Developers. `http://developer.android.com/guide/developing/device.html`, 2009.

[13] Binder — Android Developers. `http://developer.android.com/reference/android/os/Binder.html`, 2009.

[14] Touch *extremely* expensive performance-wise - Android Developers — Google Groups. `http://groups.google.com/group/android-developers/browse_thread/thread/39eea4d7f6e6dfca`.

[15] Jinzhan Peng, Guei-Yuan Lueh, Gansha Wu, Xiaogang Gou, and Ryan Rakvic. A comprehensive study of hardware/software approaches to improve tlb performance for java applications on embedded systems. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 102–111, New York, NY, USA, 2006. ACM.