EVALUATION AND ANALYSIS OF OKL4-BASED ANDROID

Rafika Ida Mutia

Department of Electrical and Information Technology
Lund University
Sweden
wx08rm7@student.lth.se

Abstract—**As smartphones and feature phones are increasingly in high demand nowadays, researches on improving the kernel capability to better support and provide extension features on mobile phone becomes interesting. Microkernel technology which widely used in embedded system devices, such as mobile phones, is one of the hot topic area on this research area. Second generation micorkernels like OKL4 provide flexibility, optimization and security while keep high-performance with a small memory footprint.**

On the other hand, the Android mobile phone operating system runs on Linux as its kernel and it offers Java runtime environment using Dalvik Virtual Machine (Dalvik VM) and custom C library called Bionic. Porting the OKL4 into Android might introduce a new efficient and improved technology on mobile phone. With their own characteristic, the OKL4-based Android could provide performance improvements, such as fast IPC mechanism, small memory usage and transparent superpage supports that could improve Java virtual machine performance.

Index Terms—**Microkernel, Android, OKL4**

I. INTRODUCTION

The increased demand on smart phone and feature phone have required the mobile phone to be more powerful, have extended features, have better connection to internet while having optimal battery life. Smartphones run complete operating system software providing a platform for application developers. There are several operating systems for different type mobile phones, such as Symbian, Android, Blackberry OS, iOS and Windows Mobile OS. An open source OS, such as Android and iOS, are equipped with large complex operating systems derived from solutions that originally targeted desktop architecture. These operating systems should solve some issues to embedded systems related, such as low-power processor, limited system memory, small caches and battery power.

On the other hand, operating system kernel is the lowest level of abstraction that is implemented in software. It manages the system resources and connects applications to actual data processing in hardware level. Microkernel with its small size, implement the smallest set of abctractions and operations on its privileged mode and put other features such as drivers, file systems, etc, in de-privileged mode. Second generation microkernel that has been developed in Open Kernel Labs, OKL4, is designed for embedded system to be used for the user environment on mobile phone while having smaller size, smaller memory footprint and increase the performance of mobile phone.

By utilizing the open-sourced Android operating system, replacement of Linux kernel with an OKL4-based system is the aim of OKL4-based Android project in [1]. OKL4 has also been optimised for processors found in mobile phones. It runs on mapped modem which is used to select the correct cache line allowing multiple mappings of the same virtual page to co-exist. The address space identifire (ASID) ensures the processes to access only their own mappings without needing to flush the TLB.

The approach of OKL4-based Android integrated system has been clearly staed and elaborated in [1]. The aim of this report is then to study the feasibility of porting Android components to OKL4-based operating system and analyze the benchmarking done in by comparing the OKL4-based Android system to Linux-based Android with further elaboration and analyses. As an evaluation system introduction, in [1] the Android Developer Phone 1 (ADP1) is used as the basis of the Linux/OKL4 and utilizes ARMv6 processors with 32 message registers.

The organization of this report is as follows. Section II briefly explains the characteristics of Android and OKL4. The components needed for the porting and the procedure on how it is done are described in Section III. The next section, Section IV, explains the implementation of porting Android components to OKL4. The Section V evaluates the OKL4-based Android system and compares it with the existing Linux-based Android. The following section, Section VII, looks at the one-step further of this system, an integration of OKL4-based Android with virtualization by using microvisor, which called OK:Android. The last section, Section VIII provides conclusion of this report.

II. **BACKGROUND**

A. ***Android***

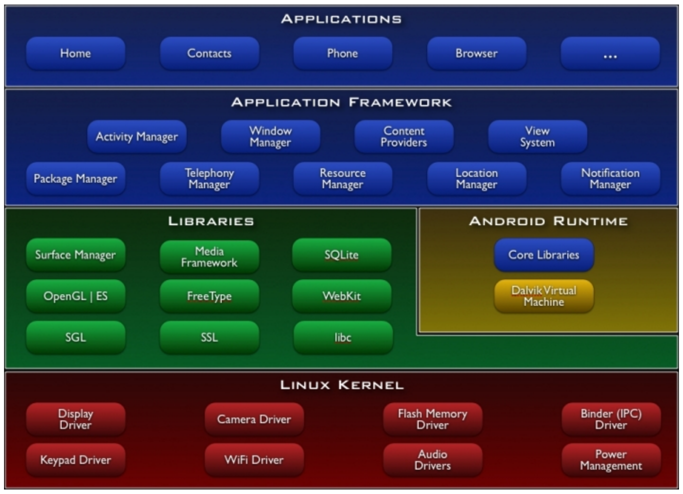

Fig. 1. Android Stack

1) ***Linux****:* The existing Android system uses Linux as its kernel, but in OKL4-based Android system, the Linux kernel is replaced with OKL4 microkernel which minimize the system to only contain required platform to operate. The OKL4 is a minimal kernel and hence the complet user space-support operating system has to be written. With this bottom-up approach, it could impact performance and memory usage.

2) ***Bionic****:* Bionic is a custom C library for Android developed by Google which provides a C system interface to the Linux kernel. It has a minimal *pthreads* implementation to support thread creation and synchronization and also contains some Android-specific additions such as system properties and logging.

To meet the requirement of embedded hardware, Google aims to have a small and fast `libc` implementation. The stripped shared library of Bionic is only 238 KB size which is considerably small compared to 1.11 MB stripped GNU `libc` implementation. A small sized Bionic is an advantage for the mobile phone to have less system memory than desktop machines. Furthermore, Android Developer Phone 1 (ADP1) has only 128 MB of RAM and more importantly has no swap space.

3) ***Dalvik VM****:* The core of user environment in Android is the Dalvik VM. The compiled format of Java code, *Java class files*, is converted into Dalvik dex files through the use of a developed tool. As such, Dalvik VM can execute applications written in Java. However, Dalvik applications are interpreted in OKL4-based Android system and it cause much slower execution than native code due to unavailable just-in-time compiler.

Dalvik, with its dex file format, has several time-saving and space-saving optimization compared to Java *jar* format, a collection of compiled Java class files. An uncompressed dex file is approximately the same size as a compressed jar file. Hence, memory saving as Dalvik executables does not need to be decompressed into system memory, and they can also be executed from a memory-mapped file. Space-saving optimization in Dalvik is when portions of the dex file that are not commonly used can be ejected from memory, and reloaded again on-demand.

Another important optimization is the Zygote process. Zygote process is a Dalvik VM process that has loaded core Java libraries and is ready to begin executing byte-code. It forks with copy-on-write semantic to create new Dalvik VM instances whenever a new application is launched. It improves the process start-up time since new applications have all core libraries pre-loaded and initialized as done by the Zygote and can be executed immediately.

As Dalvik is designed to have each process executed in its own instance, the Zygote optimization is possible. It means that if one Java application crashes, it will not impact other Java applications because of process isolation provided by the OS. This is important when all user applications run on Dalvik, as having the user environment crash due to one faulty application would not affect other application.

4) ***Binder IPC****:* The System Server process provides access to the system services such as Package manager and Power manager. It also delivers events to application for user input notification. The processes are written in Java and runs in Dalvik VM instances. In order to have access to the provided services, user application performs IPC to the System Server. Furthermore, user applications can also provide services to other user applications. These are provided by Binder IPC driver, an additional to the Linux kernel by Google.

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. To perform remote function call, the function arguments first need to be delivered to the remote application. The process of converting data into and back out of a transmissible format is called marshalling. This is achieved through the use of the Android Interface Definition Language (AIDL).

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. Binder can be used to facilitate shared memory between processes. One of the additions to the Linux kernel is an *ashmem* device driver. *Ashmem* allows kernel to reclaim the memory region at any time, as long as the region is marked by users as purgeable. The *ashmem* allocates a region of memory and represented by a file descriptor. The file descriptor can be passed through Binder to other process. The receiver can pass the *ashmem* file descriptor to the *mmap* system call to gain access to the shared memory region.

Binder implements security by delivering the caller's process ID and user ID to the callee. The callee then is able to validate the sender's credentials. In Binder, remote objects are called as Binder References. By having access to a remote object, user could perform RMI on the object. One process can pass a reference through Binder to give another process access rights to a remote object. The Binder driver control and blocks a process from accessing a remote object if it does not have the permissions.

*5) **Role of IPC:*** The IPC is used to access all services in the System Server. The example is such as to launch a new application, Binder is used to request the operation of the Activity Manager. Another examples are accessing Power Manager to keep the screen on, to flush the contents of a window to the screen by notifying the Surface Flinger, etc.

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 to user applications. The Service Manager is identified by integer zero. Furthermore, the most common user of Binder is to notify the Surface Flinger of window updates and to dispatch input events from the System Server to user application.

*B. **OKL4 Microkernel***

OKL4 provides a minimal, but portable set of abstractions to the hardware. Its design and API derives from the L4 family of microkernel. In the following subsections, several aspects of characteristic of OKL4 that is beneficial for the OKL4-based Android system will be described.

*1) **Small Memory Footprint:*** OKL4 targets embedded system with power-saving hardware, limited system memory and operating on battery power. The OKL4 is very small and minimize its presence in the caches which results in less cache misses on its execution and has more room for user applications.

*2) **High Performance IPC:*** Performances of IPC have a large impact on performance of overall microkernel-based system. All primary system services must be implemented in userspace and IPC is used for processes to access these services. The bare-bones nature of OKL4 IPC provides a simple mechanism to transfer data which results in a fast IPC mechanism.

Data is transferred from one thread to another using message registers. With 32 message registers in the platform implementation of this system, it allows IPC operation to transfer at most 32 words data, with 4 bytes per word. The message registers consist of several CPU general-purpose registers, with the rest located in the User-level Thread Control Block (UTCB) of thread 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.

*3) **ARM Support:*** The particular advantage of OKL4 on ARMv5 chipsets is its superior context-switch handling. In ARMv5, TLB entries do not have an address-space identifier (ASID), which means TLB entries of different processes cannot be separated from one another. Hence TLB must be flushed clean to prevent processes from accessing each other's memory.

In addition, the Virtually-Indexed Virtually-Tagged (VIVT) caches in ARMv5 means different address spaces can have same virtual address mapping to different physical addresses, which is common in multiple address space OS such as Linux. Hence, 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. Another way to avoid flushing of TLB caches on every context-switch is by fully utilize the ARM domains and the Fast Context Switch Extension (FCSE).

On the other hand, the ARMv6 features Virtually-Indexed Physically-Tagged (VIPT) caches and TLB entries do not contain an ASID. The translation process outputs the physical address being mapped to, and it is used to select the correct cache line allowing multiple mappings of the same virtual page to co-exist. The use of ASID ensures processes access only their own mappings without needing to flush the TLB.

*4) **Transparent Superpages:*** The userspace OS personality (OS personality will be explained in III-A) is responsible for allocating system memory to user processes and able to implement transparent superpages with the mechanism supplied by OKL4, it can implement transparent superpages. With transparent superpages, user applications allow the OS to decide the page sizes used for memory locations. A large page size means high TLB coverage and therefore less TLB misses, less processing time and internal fragmentation.

However, there is a trade-off between performance and memory usage due to internal fragmentation, but any imaginable policy can be implemented. The policy designed for Android specifically could be constructed to provide a balance between performance and memory usage.

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

III. **Methodology**

In [1], instead of porting complete Android to OKL4, only subset of Android components is ported to OKL4-based operating system. As in Figure III,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.

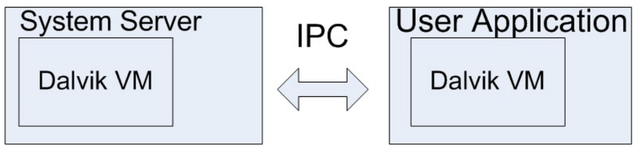

Fig. 2. High-level Requirement of an OKL4-based Android System

Firstly, a form of IPC is needed to facilitate the communication between application process and system process. Secondly, Dalvik VM requires Bionic-compatible C library and operating system support for Linux system calls. Thirdly, the System Server needs access to the drivers, such as video and input, to be able to perform its role. Finally, Dalvik needs to load Java libraries at runtime which requires a file system.

Therefore, the basic requirements to support Android runtime on OKL4 microkernel are OS personality, IPC mechanism, Bionic-compatible C library, Dalvik VM and System Server which will be explained in the sections below.

A. ***OS Personality***

A microkernel does not provide a complete OS and most standard OS services are implemented in userspace, therefore the collection of userspace servers for is called as operating system personality. Only a few things still live in the kernel, the other system policy is implemented in the OS personality, e.g. threads scheduling in the system is dictated by OKL4.

1) ***Component-based Design****:* An OKL4-based Android system are fundamentally different to a Linux-based system. Microkernels encourage the design of a componentised system running in user space. The communication between component then been done by using IPC mechanism.

Native code written for Android expects a POSIX API as provided by their `libc` implementation (Bionic) working with the Linux kernel. However, the OKL4 system call API does not provide POSIX semantics. Therefore, the OKL4-based Android system needs an OS personality running in user space 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 consists of several loop processing Linux system call requests from user space applications. The thread that handles this main server loop will be referenced as the root server.

The design will be monolithic to simplify programming and to maintain the performance. Drivers will conceptually be implemented as separate components with their own thread of control. The reason is to facilitate the componentization of unrelated drivers into separate protection domains at a later date so that driver components can be easily swapped out with different implementations depending on which hardware platform targeted. Furthermore, the OKL4 3.0 distributions provide a library layer on top of the L4 API called *libokl4* to assist the construction of OS personalities.

2) ***Drivers****:* Implementing all drivers to be handled in a single thread of control will destroy code portability since it is affected by the behaviors of each driver. Hence, each driver will be implemented as a separate conceptual component in user space with its own thread of control.

Processing overhead raised on OKL4 since the kernel delivers interrupts using IPC. To cope with it, context-switch is required, which is not the case in monolithic kernel. However, overhead is low and interrupt latency is not an issue with the devices available on a mobile phone.

B. ***IPC Mechanism***

Binder in Android is used for dispatching input events to user applications. Although in [1], Binder is not fully ported, obtaining a pixel buffer shared with the video subsystem does require Binder's shared via file descriptor feature. As such, Binder can be bypassed by replacing dynamic allocation of shared memory buffers with static allocation by the OKL4-based Android at startup. This becomes possible since the port will launch pre-determined applications.

C. ***Bionic-compatible C Library***

There are two possible approaches to develop a Bionic-compatible C library; by porting Bionic to run on OKL4-based system Android , which redirect system calls to the OS personality and by porting any missing features from OKL4's minimal `libc` implementation.

The first approach is used in [1] since OKL4's `libc` proved to be lack of necessary features and would have required large porting effort. Furthermore, using OKL4's minimal `libc` implementation will not guarantee the same semantics offered by Bionic and it could lead to compatibility issues. By porting Bionic to run on OKL4-based system Android, emulation for system calls is provided as needed by the applications running on the system.

D. *Dalvik VM*

The Android runtime is powered by the Dalvik VM. It is portable to different architectures and different OSes. As such it links to very few other libraries, with Bionic libc being the only prominent one.

Dalvik has major porting issue, the core Java libraries stored in the file system is in an unoptimized format. When Dalvik first loads these libraries, it optimizes them and stores a copy into a cache on the file system. When other Dalvik instances start up, the optimized copy from the cache can be used so the optimization process need only be performed once. Dalvik invokes the *dexopt* tool using *fork/exec* to perform this optimization. This is time consuming and requires a write-capable file system.

However, there is another method could be used, that is by uploading the libraries to the Android simulator and invokes the optimization process by launching Dalvik with specific parameters. The pre-optimized libraries can then be downloaded back. It avoids time consumed on implementing and debugging features that might not be used anywhere else.

E. *System Server*

Porting the entire System Server involves parts of the system that are not available on OKL4-based Android system, such as power management. Instead, only partial port containing components required is done such as IPC, video and input.

F. *File System*

A read-only in-memory file system is necessary to be able to load Java libraries at runtime. The OKL4 build system allows placement of files into memory and access them using `libokl4`. The OKL4 environments store 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 have not existed yet.

A real device and a real file system implementation provide more flexibility. Files can be created and written to at runtime. However, this functionality is not required for an initial Android port. Hence, an in-memory file system is implemented in the OKL4-based Android because of low complexity and less time consuming reasons.

IV. **Implementations**

A. *OS Layer*

Building an OS personality on top of OKL4 to support Android requires several subsystems to be implemented. It also have to keep the policy of minimal implementation and be simple, yet still work to optimum level.

*1) **File System:*** A read-only in-memory file system is implemented in this project. In order to let user process access this file system, extra framework is required. Therefore, a file table is allocated to each process so that files in the in-memory file system can be accessed. The entry in the file table contains the state about an open file, which is the address of the file data in memory, file size and current seek position within the file.

*2) **Virtual Memory Subsystem:*** The responsibility of VMS is to manage both virtual and physical memory. It needs to handle the demands of Android applications, which most commonly is the `mmap` system call. The POSIX semantics of `mmap` system call expect the allocation of memory regions to be zero-ed out or contain the specified file.

The required semantics are implemented by making use of the callback support of libokl4 `memsections` which actually in libokl4 is a block memory consist of a base, a range and a variety of other attributes such as access permissions and page size.

By using callback, user can choose when to map a virtual page to a physical frame during creation of `memsections` or in response to a page fault. The physical memory frame can be zero-ed out before allocating memory regions to user. By using an in-memory file system, it meets the requirement of backing memory pages with contents of a file. File contents are already in memory and can be copied directly without requirement of accessing disk device.

As a result, this project supports pre-mapped and lazily mapped `memsections` that match `mmap` semantics. Lazy mapping of pages reduces memory usage which is a benefit since Dalvik allocates several large buffers which are megabytes in size.

Specifying the page size of `memsections` is supported by libokl4 although only explicit super page support is implemented in this project. A bug in the platform code means the physical addresses are only aligned to 4 KiB blocks. The misalignment also varied between different compiles;hence only large pages are successfully mapped by manually offsetting the addresses by the size of the last misalignment. A simple function malloc_64k was made available to userspace, and has the same interface and semantics as the standard `malloc`.

*3) **Process Creation**:* New processes are created by using *fork* system call in Linux. It aims to duplicate the address space of the caller. The *fork* system call is used to launch Dalvik optimization tool, *dexopt* and Zygote optimization.

The creation of new processes is done by parsing the executable located in the in-memory file system with a custom-written ELF loader. File mapped *memsections* could be created for the text and data segments and a zero-mapped *memsections* for the *bss* segments. All system calls in OKL4 microkernel that create new address spaces are handled by *libokl4* which provides the protection domain abstraction for address spaces.

To launch multiple Dalvik VM instances in the system from the same binary, program arguments (`argc` and `argv`) are implemented by copying the arguments data into the new process's address space and pointing the `L4_UserDefinedHandle()` to this piece of memory. When the process starts up, it finds arguments and passes them to the program's main function.

*4) **Thread Creation**:* Android makes use of multiple threads in its application and in system framework. The process of starting up new threads is a combination of OS code and user library code. In OKL4-based Android, the *libokl4* is used to create or start new thread. The *pthreads* library in Bionic is responsible for creating new threads and providing synchronization tools for multi-threaded environment.

During thread creation, a stack location is chosen by *pthreads* library and information of executed function is placed above the given stack pointer. In Linux thread creation, it is handled by clone system call which consists 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*. Different return value from clone is used to force the parent and child to take different code paths out of the system call.

Because of lack implementation of *fork* in OKL4, the process is a little bit different. The startup information 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 pass it to the *pthreads __thread_entry function*.

*5) **Futexes**:* For more sophisticated thread-synchronization tools, such as semaphores and condition variables, Linux provides a fast user space mutex (futex).

Semantics

The semantics are such followed:

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

The semantics involve synchronization using the value and physical address of a single word as pointed to by the variable *ftx* in the above function declarations. The syncronization process of semantics is such followed.

Thread A calls `futex_wait` to sleep until Thread B calls `futex_wake` on the same futex variable. If Thread B do this first, the futex semantics say that Thread A should awake immediately. To do this, `futex_wait` compares 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, but if the Thread B changes the futex value and then calls `futex_wake` before Thread A performs the `futex_wait` call, then the expected value expected by Thread A of the futex word is wrong. Then a call to `futex_wait` will cause Thread A to wake up immediately as intended.

In `futex_wait`, the kernel compares the expected value of the futex word to the real value. The OKL4-based Android OS personality is required to maintain page tables to translate word's virtual address into physical address.

In order for the futex implementation be compatible with future OKL4 release, user space page table is implemented. As value must be accessed on a call to `futex_wait` in semantics, the page containing the *ftx* variable must be mapped in and accessed. The mapping to OKL4 can be done by using system call. To avoid overhead, the page is not necessarily be unmapped so that it can be used for the next call to the 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. A futex is identified by 32-bit physical address of the futex word. The 32-bit address can be used as a key to associative container when accessing the state of a futex.

*6) **Timer Driver**:* There are two timer devices on ADP1 bus; the general-purpose timer (GPT) and the debug timer (DGT). OKL4 already uses GPT to implement timer ticks for context-switching. Hence, DGT is left for timestamps and a sleep functionality implementation.

7) Video Driver: Video has internal buffer storing of last sent frame; hence it can be lazily pushed. It is different to Video Graphic Array (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 needs to provide three functions; access to the frame buffer, permission invoking of Direct Memory Access (DMA) operations to update the screen, and a notification for when the DMA is complete.

*8) **Input Driver**:* The ADP1 has several sources of events, such as 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 pressed or touched generates an interrupt. As all work is interrupt-driven, only a simple IPC wait loop was required.

Each touch event consists of several events in the Linux input framework. The OKL4 touch driver does not operate exactly the same on Linux. Reading events from touch device in /dev/input reveals the Linux driver interleaves events resulting in less data being transferred. The OKL4 touch driver sends all events every time due to time constraints by generate hard-coded touch sub-events per touch interrupts.

There are two approach can be done to move event data out of the driver; to synchronously transfer each event using the OKL4 IPC message registers and to use asynchronous notifications and a fixed-size shared memory region where parties read and write event data from a single-reader and single-writer lock-free queue.

B. *Dalvik Application Framework*

Starting up new Dalvik applications is a process that requires the System Server to provide several services including Package Manager, Window Manager and Binder. In order to avoid full runtime stack, a user application guises the standard API in order to develop a proof-of-concept demonstration that Android applications can run on an OKL4-based system. However, the implementation is used only for user applications and not the System Server.

*1) **UI Layout**:* The UI layout in Android applications is described by XML file. The entries in the layout include platform-supplied classes such as *TextView*, *SurfaceView* or application-specific classes. When an application is packaged for release, the layout file is compiled into binary form for faster parsing during application launch. In OKL4-based Android system, the class name that form the UI layout is extracted and Java reflection is used to construct Java objects based on the classes.

*2) **Image resources** :* For image resources, Android access image files by passing an auto-generated enumeration to the Java framework. Then the object that representing the image is returned.

*3) **Event Loop**:* User applications in Android consist of an event-loop that processes messages from a queue. Events can be generated from input devices or from user applications itself. The event loop waits on OKL4 IPC to receive key and touch events from the System Server and thus replicates the behaviors of native Android application.

C. *System Server*

Many system services that allow access to devices such as input, audio and video run in the System Server. In OKL4-based Android, video and input frameworks are ported to the system and calls to Binder IPC are substituted with OKL4 IPC.

The input framework responsibles of getting events from the OS and deliver them to the appropriate applications. There are two main components threads of control; `InputDeviceReader` and `InputDispatcher`. The `InputDeviceReader` read events from the kernel and pushes them onto a queue, for the `InputDispatcher` to do remaining work whereby the event is eventually dispatched to the user application using Binder IPC. In [1], the event reading from the OS is replaced with OKL4 equivalent solution that is using 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.

*1) **InputDeviceReader**:* The `InpurDeviceReader` responsibles for reading events from the OS and dealing with input events that may affect the system. The OKL4 is implemented by using asynchronous notifications and shared memory, which requires only two system calls to complete touch event; *IPC notify* by the driver and *IPC wait* by the receiver. It significantly improves the time taken to extract event data from the OS since Linux usually require 3 to 6 system calls, one poll and 2 to 5 touch sub-events.

Due to missing some system functionality in the OKL4-based Android project, various portions of code were removed to make the event routing path of the `InputDeviceReader` functional, including the power management, battery statistics and replacing checks to know the phone status.

*2) **Input Dispatcher**:* The `InputDispatcher` responsibles to do routing event for user-applications. The correct application is determined and the event is marshaled and dispatched over IPC. However, there is a replacement for 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 function to place primitives and Java built-in classes into the array container. The Java implementation of *Parcel* mostly enters native code using JNI to perform C++ calls on the C++ parcel class.

Parcel class is used for marshaling, but 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.

However, OKL4 IPC has limitation that the use of shared memory is encouraged for larger transfers. While the modification is not ideal for benchmarking, which is the first goal of this OKL4-based Android, performance of transferring 2 bytes is accepted since it is unlikely to have a noticeable influence on the results.

Similar to `InputDeviceReader`, various portions of `InputDispatcher` had to be removed, such as power management, battery statistics and large section of code that determined the destination of the input event. Furthermore, there is only one client application to send events and also feature that stores event history within `MotionEvent`, the class that represents touch events, is removed. Extra events are stored together to avoid them being dropped, but investigation into this matter revelaed that it is actually rarely occur in Android and sending multiple events together can be ignored.

V. System Evaluation

A. *Improving Performance with OKL4*

There are three segments of performance that has been improved with the OKL4-based Android; input driver framework, IPC and superpages. The input framework is good platform for microkernel-based optimization because data needs to be moved from the driver, to a subsystem component and finally to a user application. A fast IPC mechanism can improve performance in this area. The OKL4 also offers the ability to experiment with different page sizes.

The use of asynchronous notifications and fixed-size shared memory region as method to deliver input events to the System Server reduce the processing time for retrieving touch events compared to the implementation on Linux. It is considered as major advantage since touch events generate a lot of traffic in the input subsystem.

Binder IPC responsibles for dispatching input events from the System Server to user application. Improving IPC performance can reduce the amount of processing done by the dispatch stage of input routing and therefore reduce overall processing time of input event.

Another factor to improve the performance is the usage of superpages and the possibility of having transparent superpage support. Increasing TLB coverage on Android is a major advantage due to sheer size of Dalvik VM and Java runtime framework. With less cycles wasted on TLB misses, overall processing time could be reduced which results in an improved performance across the entire system.

B. *Evaluation Environments*

1) *Android Developer Phone 1 (ADP1):*

- Qualcomm MSM7201A system-on chip which has ARM1136js processor
- ARM9 processor for baseband software
- Separate 4-way 32 Kb and data caches
- Two-level TLB hierarchy with first level of 10-entry fully-associative MicroTLBs implemented in logic for both instruction and data. Second level contains a single MainTLB made up of two memories, an 8-entry dully-associative block and 64-entry 2-way block

2) ***Android:*** Android version 1.5 release 2 as the basis for the port to OKL4.

3) ***OKL4:*** OKL4 was compiled with all performance options switched on and kernel tracing disabled. Dalvik was compiled with profiling disabled. OKL4 `kdebug` interface is used to access the event counters without invoking the OS personality.

4) ***Linux:*** Linux has a built-in profiling framework called *Oprofile* that enables configuration of event counters on ARM11 CPU. It is not compiled into the kernel with default build. However, since several problems encountered, a framework is created and code is placed inside the Binder device's `ioctl` system call.

C. *Benchmarking for Comparison of OKL4-based Android to Linux-based Android*

In [1], comparison to investigate the performance of Linux-based Android and OKL4-based Android is done by using CaffeineMark 3.0 JVM benchmarking suite. By using the same hardware and Dalvik VM implementations, but in different OSes, the performance is expected to be similar. However, to avoid porting problem, the hardware is tested by running series of tests, such as :

- Sieve; Uses the *Sieve of Eratosthenes* algorithm to find primes numbers
- Loop; sorting and sequence genaration to measure compiler loop optimization
- Logic; Tests the speed with which the VM executes decision-making instruction
- String; Meausres 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

The results of the benchmark shows similar performance both in OKL4 and Linux which means all further benchmarks should be carried out using all 5 libraries.

*1) **Superpages:*** As the OKL4-based OS is built from the scratch, superpages gives big advantages. The ADP1 uses ARM11 CPU which has 72 TLB entries in the main TLB. For a system that uses 4 Kb page size has a TLB coverage area of 288 Kb. Hence, by using a larger page size, the TLB coverage area is increased which means TLB misses is reduced and it could improve performance of the system.

On Android, video subsystem uses a compositing window manager which means each application on the display has its own pixel buffer and it consumes memory. Size of background image or framebuffer is 300 Kb which exceeds the TLB coverage area. It means that some of drawings or images effectively flush the TLB at least once.

*2) **IPC Microbenchmark:*** Performance of IPC in OKL4 and Binder Android could be different. The IPC in OKL4 is a minimal implementation of a fast IPC mechanism while Binder IPC provides more functionality. To make fair comparisons, isolated performance of both IPC mechanisms are done. However, IPC performance on native code will be different than on Java code which needs to use Java Native Interface (JNI) framework to access the native IPC functions. Android runs all its application on Java; hence performance on IPC through Java JNI should be compared to native code.

Framework of Binder allows set up of simple Remote Procedure Call (RPC) service. Hence, a simple service is created to measure the round trip-time for an IPC. The same scenario is done on OKL4 by using OKL4 IPC. Binder driver API is not so easy to be measured as kernel schedule any number of arbitrary tasks between send and receive. Android provides a C++ API and Java framework API to make it easier to use Binder. The API allows registering of RPC service with the system Service Manager. The RPC service is a layer abstraction above delivering raw uninterpreted data and these data need to be interpreted in a real environment to make progress. Thus, all IPC benchmarks are based on simplest RPC scenario of executing ping-pong.

The test is carried out by executing a ping-pong loop by client in about thousands of iterations. THe average of round-trip-time is calculated by dividing the total number of cycles by the number of ping-pong iterations. Furthermore, the average Main TLB misses is also calculated with the same method.

*3) **Input Framework:*** The input framework is divided into three parts for fair comparison purposes; obtaining events, Java processing and dispatching the event that will be described as follows.

Obtaining Events The `InputDeviceReader` obtains an event from the touch driver. OKL4 uses a single system call to wake up before accessing the event data from shared memory and wait for a notification from the dirver. On the other hand, Linux uses pair of system calls, poll and read to access the data.

Java Processing After event is obtained, it is passed back to Java code and being processed to determine its purpose. Here, in OKL implementation, some of input routing path in System Server is missing. However, comparable measurements are taken.

Dispatching the Event

After being processed, the event is dispatched over Binder IPC to the user application. Preliminary IPC benchmark showed that OKL4 IPC outperformed the Binder IPC and improve the dispatch performance. The measurement were taken from point right before the touch event is marshalled and dispatched in System Server until point right after event is reconstructed and identified.

VI. **Results and Analysis**

*A. **Superpages***

As expected earlier, number of TLB misses is reduced in OKL4-based system. However, it does not improve the performance much since data cache gives the bottleneck perfromance. In the test, reading and writing from large buffers will be effectively flushing the 32 Kb data cache several times over just to draw the background which requires accessing 600 Kb of data sequentially. Hence cost of data cache miss could not be estimated without particular measurement on it.

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%)

Fig. 3. Benchmark of Application Using a Different Page Size for Large Pixel Buffers

However, rough estimate of TLB miss can be calculated by obtaining the page size of two buffers in the two benchmarks. From Figure VI-A, it can be seen that the standard deviation is low for all measurements but there is more variance on 64 Kb page size. There are 192 less TLB misses and saving of 34,149 cycles in procesing which suggest the cost of 204 cycles. From the standard deviation in the results, calculated TLB miss cost ranges between 192 and 217 cycles. Furthermore, it is confirmed by measuring the cost of uncached memory reads which done by measuring the number of cycles when performing 4 reads to uncached memory in a tight loop for 16384 iterations. The measured cost of a single read to uncached memory was 95 cycles.

A TLB look-up on ARM11 begin by accessing one of the MicroTLB on MainTLB. One second of TLB miss will result in a walk of a two-level page table by hardware which requires two uncached memory reads and total cost of 190 cycles. However, the miss latency of the look-up on the MainTLB is still unaccounted for and from ARM Reference Manual [3], the remaining 14 cycles can be attributed to this process. It confirms the early calculation of complete TLB miss cost of 204 cycles is close to the real cost.

B. ***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

Fig. 4. C/C++ ping-pong micro-benchmark

1) ***Analysis of C/C++ IPC Micro-benchmarks****:* Figure VI-B1 proves that OKL4 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). Binder IPC is a deivece in a file system and the *ioctl* system call for the Binder device is overloaded to handle all calls to Binder. There is no IPC fastpath through the kernel which limits the maximum performance of Binder IPC and make OKL4 IPC be faster than Binder IPC.

System call *ioctl* in Binder is a series of command that is interpreted by Binder driver. The IPC command is BINDER_WRITE_READ (BWR) and signals to the processing driver, that is write, read or write followed by read. Some important things discovered when using BWR to do a write followed by a read could affect performance, such as following:

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

The TC messags in Android appear to be ignored but removing the generation of TC message causes the system fail to boot. It might be part of the design or a bug, or part of original OpenBinder implementation. However, it constributes to poor perfromance of Binder.

The Binder implementation also allows multiple user application to calls into Binder at the same time. A global Binder lock must be acquired to make progress. The threads that are waiting on a read operation release the lock and wait in the driver to be woken up where they reacquire the lock and continue. The location of user process's payload can be anywhere in the user's address space and of unpredictable size. On the other hand, OKL4 IPC implementations have IPC fastpath, not multi-threaded and accesses user data from a predetermined location (the thread's User Thread Control Block, or UTCB).

Lack of IPC fast path which effectively flushing the TLB once per iteration can be the reason of high TLB miss rates on Binder. Whereas on OKL4, use of an IPC fastpath means it minimizes the amount of text and data touched to reduce TLB misses.

2) ***Analysis of Java IPC Micro-benchmarks****:* 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 minimize the number of JNI calls required to perform IPC.

While in C/C++ benchmark change of payload size make Binder took an extra time of 35 microseconds when moving from a 4 byte payload to 124 bytes, on Dalvik VM it is increased by over 1 ms. The change on payload size does not account for the difference on Dalvik. The only change is use of JNI to fill up 31 payload integers. By using a Parcel, a standard Java class is used by Binder to send IPC data, one JNI operation is required to load each integer for delivery. The same applied for integer arrays where each integer is still loaded one-by-one.

	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

Fig. 5. Java ping-pong micro-benchmark

The last column of table VI-B2 measures the cost of all payload of JNI calls when using 124-byte payload. However, this benchmark is only conducted on Linux. At each iteration, two sets of 31 integers are written through JNI, 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 126 JNI calls in total and takes almost a millisecond to perform. The total JNI per call is estimated by dividing the total time to 126, which is around 7.84 microseconds per JNI call. It is longer than the round-trip-time of a 4-byte using OKL4 IPC on C.

The results shows that Java JNI decreased the IPC performance. The number of JNI calls for the OKL4 benchmark was minimized to only one. Its benefit is highlighted with OKL4's 124-byte round-trip-time on Java which outperform Binder on C++ using 4-byte payload.

The TLB miss rate while performing JNI is also incredibly high. Binder in C++ has contributed stress on the MainTLB by effectively flushing it once per iteration. Together with Dalvik and JNI, it appears to be the cause of sharp drop in performance for the 4-byte Binder round-trip-time on Java. It explains the reason of 4-byte round-trip-time on OKL4 does not suffer performance drop as much as Binder does.

C. *Input Framework*

	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)

Fig. 6. Results for Obtaining an Event

*1) **Analysis of Obtaining an Event:*** Processing the event on Linux is done by read the system call whereas OKL4 uses shared memory. Reading from shared memory obviously results in a faster IPC compared to reading from system call. However, a touch event consists of multiple sub-events and it depends on the applied interleaving. It means that the event is read from driver several times before generating touch event in the Java code and it propagates through the system.

However, the benchmark measures the cost of reading a single sub-event and not the whole event. Furthermore, multiplying results from OKL4 to cover the whole touch event can not be compared to the results from Linux due to large variance in Linux measurements. The code measured is native and small C/C++ code. Despite of this fact, another benchmark can be implemented and measurement can be taken for the whole touch event in order to measure the input cost. However, looking at the results and input path from table VI-C1, it can be concluded that the input framework is not a bottleneck of the performance.

*2) **Analysis of Java Processing:*** Variations in results of OKL4 and Linux implementation can be quite interesting. Android normally takes 2 ms to process a touch event, but the OKL4 implementation does it in 417 microseconds. A proper process of Java processing is such following:

	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)

Fig. 7. Results for Java Processing

1) Check the purpose of event,
2) Notify the power management service that a user activity is happening,
3) Log the event with battery statistics service,
4) Place the event onto shared queue and notify another thread,
5) Read the event off from queue and examine its type event,
6) Determine which process to send the event to

In [1], only stages 1,4 and 5 are done, which means the other stages can contribute to slow-down experienced on Linux. The TLB misses is definitely the contributor of long process time. Further profiling is required to determine the true source of the issue, but JNI is likely to play a part. As observed in IPC micro-benchmarks heavy use of JNI destroys the performance.

	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)

Fig. 8. Results for IPC Dispatch Stage

*3) **Analysis of IPC Dispatching:*** The IPC dispatch results in table VI-C3 shows that OKL4 IPC outperforms Binder IPC. In *no marshalling* column, an extra set of OKL4 was taken since the JNI cost has been known beforehand. This is done to be able to differentiate the cost of IPC from cost of JNI. From this result, it is now more convincing that JNI is responsible for the majority of processing time and TLB misses although OKL4 is still faster than Binder. Replacing Binder IPC with OKL4 IPC will improve the overall time by about 5%.

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. Applying it could help reduce the impact of touch procsesing on the system. Android is addressing this issue for future releases by throttling the frequency of touch events to 35 per second.

VII. **OK:Android**

The OK:Android is a virtualized Android components, distributed and integrated as a secure cell under the control of the microvisor. Microvisor is microkernel-based embedded hypervisor which developed by OKL4 and supports mobile virtualization, componentization and security. The OK:Android is an OS support package for Android from OK labs which enables use of a guest OS on OKL4 mobile virtualization platform. The OK:Android provides a standard Android environment without any modification in Android application. However, new application can be developed by using standard Android development tools.

A. Advantages of OK:Android

Benefits creating Virtual Machine (VM) with an Android guest OS are such as followed.

- Android applications can run on the same processor side-by-side with legacy applications and legacy OS. It eliminates the need of multiprocessor hardware or porting the legacy system to Android
- Through Secure HyperCell Technology, OKL4 cells complement the Android VM by providing an execution environment with better real-time properties and stronger security
- OKL4 cells are well-suited to hosting RTOS, ease the implementation of latency-sensitive functions without sacrificing the rich ecosystem support available for Android

The OKL4 cells make it easier to meet the security and certification requirements of key applications with much smaller trusted computing base than is possible for an Android environment. In OKL4 microvisor, Android and its application can run in isolation from other software subsystems which offer higher level security and reliability to the subsystems without providing dedicated hardware execution environment.

B. Towards OK:Android

The OK Labs microvisor combines operating system, embedded hypervisor, virtualization and componentization capabilities in a very small piece of system software. The key capability of the microvisor is the Secure HyperCell Technology. The microvisor has complete and sole control of the underlying hardware and is the only software running in privileged mode; each microvisor cell partitions and multiplexes the hardware between any other software on the target system as required, from high level OS down to individual applications and drivers.

In the following subsections, the process to a complete, rapid and risk-reduced Android migration path will be discussed.

*1) **Integration of OKL4 with the target hardware**:* The OKL4 microvisor supports common embedded mobile processors. It has complete System on Chip Software Development Kit (SoC SDK) and provide a simple method for system engineers to integrate the OK Labs microvisor.

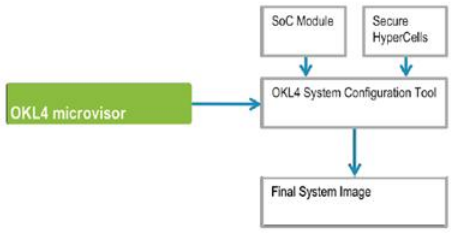

Fig. 9. Integration of OKL4 with targetted hardware

*2) **Mobile phone virtualization and integration of Android**:* The microvisor has an off-the-shelf pre-virtualized OK:Android implementation available. The OK:Android is received and integrated to the system utilizing the OKL4 system configuration tool. All Linux applications, including the rest of the Android platform stack, are completely binary compatible and executed without modification.

When a syscall is executed by an Android application, the microvisor simply encodes the syscall into a message transmitted by its IPC.

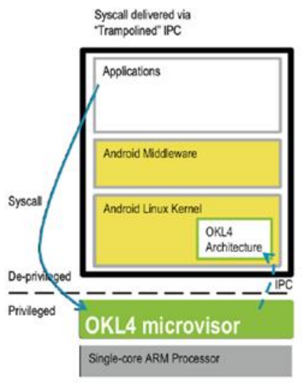

Fig. 10. Mobile phone virtualization with integrated Android

3) ***Mobile phone virtualization and introduction of the communications stack:*** The communication stack component that is most commonly running on an RTOS can be hosted without change in the OKL4 microvisor environment. Furthermore, a number of common communications stacks for mobile devices are already virtualized for the OKL4 microvisor.

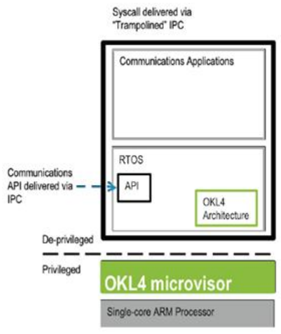
Syscall delivered via "Trampolined" IPC
Communications Applications
RTOS
Communications API delivered via IPC
API
OKL4 Architecture
De-privileged
Privileged
OKL4 microvisor
Single-core ARM Processor

Fig. 11. Communication stack in virtualization

4) ***Migration of legacy components:*** Several selected legacy components are migrated to OKL4 microvisor directly without using OKL4 compatible libraries and integrated into the final system as a cell. One or more secure cells can each contain a single, strongly isolated, highly integrated, legacy component that executes directly on the OKL4 microvisor.

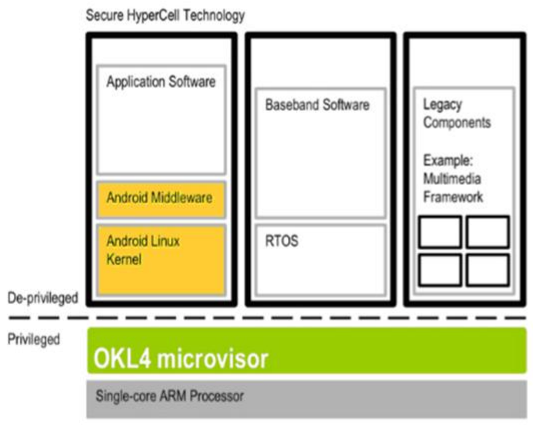

Fig. 12. Full migration of virtualization with Android

VIII. **Conclusion**

The Android is not completely ported to OKL4 microkernel in [1], only some sufficient parts are ported. Analysis on feasibility and worthiness of porting Android to OKL4 is also elaborated. From the system evaluation and benchmark result, it is proved that OKL4 could improve the performance of Android in several aspects, such as memory usage, IPC performance and exploitation of superpages. However, measuring memory-usage of incomplete port provides less information than a complete port. Therefore, it could not be investigated further than speculation about OKL4's minimality principle.

On the other hand, IPC performance is one of the major advantage by porting OKL4 to Android. Android componentizes mean of communication between user applications with another userspace process to access system services and devices, including input, video and audio. The fact that OKL4 IPC is much faster does not improve the performance much since Dalvik has to be interpreted in user environment. Furthermore, interpreted user environment (Dalvik) decrease the performance improvements. In addition to it, the JNI implementation to access native code is slow and size of Dalvik applies a lot of pressure on the TLB.

Although transparent superpages could not be implemented in [1] due to time consumed in TLB misses handling, it could improve the performance with its own characteristic that has been mentioned before. The lacks of transparent superpages in Linux makes the OKL4 as an advantage.

From the CaffeineMark 3.0 benchmark, Dalvik performance on OKL4 is similar to the one on Linux. In [2], it shows that use of 64 KiB page improved Java VM performance by 9%to 48% and in 1 MB pages, the performance is improved from 24% to 48%. [1] concludes that if Android is run under more proper componentization by isolating services, the increase in reliance on IPC would make OKL4 a better candidate for a more trustworthy platform.

Furthermore, OK Labs has introduced OK:Android, an off-the-shelf paravirtualized version of the Android smartphone platform. OK:Android uses Secure Hypercell Technology to enable Android to be used as guest OS running in a secure hypercell on top of the OKL4 mobile phone virtualization platform, OKL4 microvisor. Combination of OK:Android and OKL4 also extends new levels of security and robustness to the increasingly popular smartphone OS from Google and the Open Handset Alliance (OHA).

References

[1] Michael Hills, *Native OKL4 Android Stack*. Bachelor Thesis of University of New South Wales, Australia, November 2, 2009.

[2] Jinzhan Peng, Guei-Yuan Lueh, Xiaogan Gou, Ryan Rakvic, *A comprehensive Study of Hardwaare/Software Approaches to Improve TLB Performance for Java Applications on Embedded Systems*. Proceeing of the 2006 Workshop on Memory System Performance and Correctness, pages 102-111. New York, USA, 2006.

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