An Overview of Microkernel, Hypervisor and Microvisor Virtualization Approaches for Embedded Systems

Asif Iqbal, Nayeema Sadeque and Rafika Ida Mutia

Department of Electrical and Information Technology
Lund University
Sweden

Abstract—**This paper addresses an essential application of microkernels; its role in virtualization for embedded systems. Virtualization in embedded systems and microkernel-based virtualization are topics of intensive research today. As embedded systems specifically mobile phones are evolving to do everything that a PC does, employing virtualization in this case is another step to make this vision a reality. Hence, recently, much time and research effort have been employed to validate ways to host virtualization on embedded system processors i.e., the ARM processors. This paper reviews the research work that have had significant impact on the implementation approaches of virtualization in embedded systems and how these approaches additionally provide security features that are beneficial to equipment manufacturers, carrier service providers and end users.**

Index Terms—**microkernel, hypervisor, microvisor, virtualization, embedded systems**

I. Introduction

Virtualization has bought overwhelming changes in IT sector, since it allows multiple operating systems to run on a single computer there by consolidating servers, reducing captial costs, increasing energy efficiency and offering other significant advantages. The next big step is to implement virtualization on embedded systems, especially on mobile phones where the same benefits can translate to saving $5 to $10 per phone as estimated by Open Kernel Labs. [1] At present, programmers need to develop OS-specific applications, rewriting the same applications for different OSes. This can take months. With virtualization, features on a mobile phone can be added regardless of the operating system, making applications more accessible, reducing the number of processor chips required thereby cutting costs. Furthermore, virtualization on mobile phones can help carriers respond to security changes since operators will then be able to allocate access and bandwidth variably between untrusted and trusted applications.

Virtualization on embedded processors such as ARM processors are generally tested on top of a microkernel or a hypervisor; the L4 microkernel and Xen are examples of each. However, Open Kernel Labs, a world leading provider of mobile phone virtualization solutions have implemented and marketed the first virtualized mobile phone in April 2009 called The Motorola Evoke QA4. The software architecture for Evoke uses two virtual machines, one running Linux and the other running BREW(Binary Runtime Environment for Wireless) on top of the OKL4 microvisor which hosts features from both the traditional microkernel and hypervisor. [2]

However, the market for virtualization for embedded systems including mobile phones are still in its early stages, as the technology is not widespread. This paper focuses on the microkernel-based virtualization approaches for embedded systems. Comparisons with the hypervisor approach are discussed, with performance evaluation of the different approaches. The paper also addresses the issue of security of microkernel-based virtualization.

This paper is organized as follows. Section II starts off with a brief introduction on virtualization and the two kinds of virtualization related to this paper. It includes comparisons between microkernels and hypervisors, two competing platforms for virtualization in embedded systems. A detailed description of the three basic approaches to implementing virtualization in embedded systems is presented in Section III and Section IV. Section IV provides detailed description of a sucessful implementation of the OKL4 microvisor on the Motorola Evoke QA4 mobile phone. In Section V practical implementations of the different approaches i.e., microkernel and hypervisor and their performance comparisons are reviewed. Before concluding, Section VI looks into the security advantage provided by virtualization and outlines how both the microkernel and the hypervisor achieves this objective.

II. Background

This section provides an overview of virtualization and the types of virtualization that are related to this paper. Virtualization in embedded systems is introduced

and the difference between the two approaches that is used to carry out virtualization, the microkernel approach and the hypervisor approach, are investigated.

A. Virtualization

Virtualization refers to providing a software environment on which programs and complete operating systems can run as if they were running directly on the hardware. The software layer providing this environment is called Virtual Machine Monitor (VMM). [3]

In order to provide this illusion, VMM has three important characteristics: [4]

1) The VMM provides an environment to the software that is essentially identical with the original machine.
2) Programs running in this environment show minor decrease in speed.
3) VMM has the complete control of the system's resources.

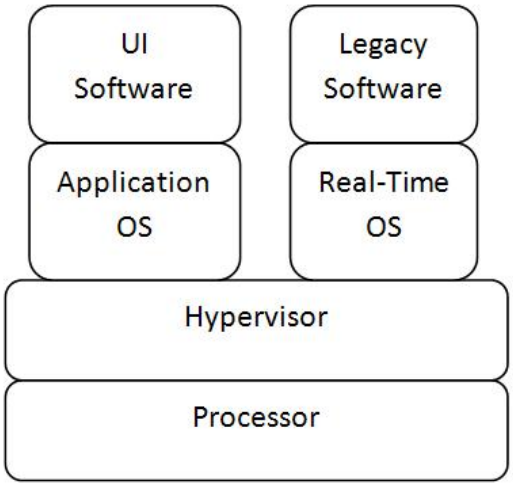

Fig. 1. Virtualization Architecture in Embedded Systems

All the three characteristics are important and make virtualization highly useful in practice. The first point ensures that software makes sure that software that runs on a real machine will also run on the VM and vice versa. The second guarantees that virtualization is implemented while keeping high performance in mind, and the third ensures that only hypervisor or VMM is running on the hardware and no other software can control any of the system's resources without authorization from the VMM.

Virtualization is provided mainly in two ways, Para-Virtualization and Native/Full Virtualization [5]. Both will be explained such as followed.

- *Para-virtualization*
  This technique requires modifications into the kernel of the guest operating system so that it calls the underlying virtualization software instead of relying on the complete emulation of system's hardware. This kind of virtualization is provided by Xen and L4 with modified OS running on top of the microkernel.[5]
- *Native Virtualization*
  This technique requires no modifications to the guest operating systems, it relies on the hypervisor to emulate the low level features of the hardware as expected by the kernel of the guest OS. As a result, full operating systems like Linux, UNIX and windows can run inside the virtual machines with no changes. Native Virtualization is usually realized either by Dynamic Binary Translation (DBT) (e.g. VMware ESX) or by hardware assistance (e.g. Xen). [5]

B. Virtualization on Embedded Systems

Modern embedded systems are increasingly moving towards general purpose systems. Their functionality is growing rapidly and as a result, the complexity and the size of the software needed is also growing. The software stack on a Smartphone is already 5-7Mloc(Million Lines Of Code) and growing. Also embedded systems are running applications which were developed for the PC, as well as the applications written by programmers without embedded system expertise. As a result, the demand for high-level application-oriented operating systems with commodity APIs like Linux, Windows, Mac OS, has increased. [6]

Still, embedded systems have some differences to general-purpose systems. Embedded systems are still real-time systems. They are still resource constrained like the power availability in the form of a battery, resulting in tight energy budgets. Also, the memory is still a cost factor in addition to being a consumer of energy.

The relevance of virtualization in embedded systems comes from their ability to address some of the new challenges posed by them [6], such as explained in the next paragraphs.

Mainstream operating systems lack the support for true real-time responsiveness required from a RTOS and they are also unsuitable for supporting the legacy firmware running on the current devices. Virtualization can help here by providing Heterogeneous operating system environments with a RTOS and an application OS (Linux, Windows, Symbian, etc) running on the same processor, as shown in Figure 1, given that the underlying hypervisor can deliver interrupts reliably fast to the RTOS. As a result, the legacy stack can continue providing device's real-time functionality and the application OS can provide the required commodity API and high level functionality suitable for application programming.

The above problem can also be addressed by using a multi-core processor with application OS running on one

and the RTOS running on another core provided that there is some hardware support for securely partitioning memory. The virtualization supports architectural abstraction, as the same software architecture can be migrated essentially unchanged between a multicore and a virtualized single core processor.

The strongest motivation for virtualization is the security. With the trend towards open systems, the probability of an application operating system getting compromised increases significantly, resulting in a system wide security breach. This can be minimized by running such an OS in a virtual machine, thus limiting access to the rest of the system. Also the downloaded code may be restricted to run in a virtual machine environment, or services which can be accessed remotely could be placed in a VM. This use, however, is only valid if:

- The underlying hypervisor/VMM is considerably more secured than the guest OS, meaning that the hypervisor must be smaller to keep TCB smaller.
- Critical functionalities can be isolated into VMs different from the exposed user- or network-facing ones. [6]

If these conditions are not met, the hypervisor will increase the TCB size which is not good in security point of view. Now in order to tackle the challenges posed by modern embedded systems, virtualization software should meet the following criteria:

- capability to provide strongly encapsulated VMs/components, which is required for security isolation and fault containment as well as it should be able to run isolated guest OS instances and their application stacks (unmodified). This must be done while preserving real-time responsiveness for the time critical subsystems.
- capability to provide controlled low-latency, high-bandwidth communication between components, including shared memory, controlled by a system-wide security policy, imposed by a small and trustworthy TCB. The components should be lightweight enough so that the encapsulation of individual threads is possible in order to enforce a global scheduling policy.

These requirements exceed the capabilities of a hypervisor and require more general-purpose OS mechanisms. High performance microkernels seem to provide the right capabilities required for this job. Various members of L4 microkernel family feature extremely low overhead message passing (IPC), provide encapsulation by having lightweight address spaces, memory mapping mechanisms and high performance user level device drivers.

The efficient and fast IPC mechanism is the key for a high performance microkernel based system including virtual machines. Virtualization traps are caught by the microkernel and converted into corresponding exception messages sent to a user-level VMM. It, together with lightweight address spaces, is key enabler of encapsulated lightweight components. [7]

C. Microkernels vs. Hypervisors

There are many literature arguing the ease of practicality of microkernels and hypervisors. While all claim that the two kinds of systems are common in their architecture, the differences lie in their technical implementations. [8] and [9] are two papers that vies for the feasibility of adopting virtualization by implementing these two architectures. One of the advantageous characteristics of hypervisors proposed by [8] is that although VMM acts as an additional layer between the hardware and the user, it does not degrade the performance noticeably while keeping the overhead small.

In addition, since VMM are out-of-the-applications, the code can simply be run on regular desktop machines without requiring modifications to existing applications. [8] further suggests that the external pagers used by microkernels for memory management can have significant impact on performance, as the kernel may need to wait for a long time for a pager to evict before other executions can proceed.

VMMs on the other hand strictly partitions memory between VMs and does not employ pagers. Consequently, failure of a VM is isolated, thus it does not compromise the stability of the system. Moreover, the developers of hypervisors need not deal with improving IPC performance, which is an issue for microkernels.

Finally, another important difference lies in the granularity of compartmentalization of Xen(a hypervisor) and a microkernel. Since microkernels changes the API visible to applications, considerable effort must be spent to implement emulation interface layers on existing OSes. A characteristic of Xen including other VMMs is that existing OSes can be run on top of it, allowing VMMs to be easily adaptable to feature sets of existing OSes while providing a familiar development environment and using existing tools extensively such as, networking routing, disk management, etc.

[9]refutes the claims put forth by [8]. First it lists the three core primitives of the IPC in microkernels followed by a subset of the primitives in VMMs. Combining the three primitives of the IPC into a single primitive results in reduced number of security mechanisms, code complexity and code size. A reduced code size in turn means reduced number of errors in the privileged kernel including reduced footprint.

In contrast, the rich variety of primitives for VMMs requires a dedicated set of mechanisms, resources and kernel code. While [8] suggests that employing VMMs to carry out virtualization requires minimal modifications to existing operating systems, [9] insists that the recent redirection of VMMs to adopt para-virtualization instead

of pure virtualization means that this advantage is uncountable. VMMs has a diversity of interfaces that leads to structural compromises. While super-VMs can be used to combine and co-locate critical system functionality, it introduces a large number of software bugs. A VMM interface is structured to be close to the underlying architecture thus making it inherently unportable across architectures. The same is not true for microkernels since it uses a common set of abstractions to hide peculiarities of the platform. With microkernels, it is possible to reuse system components across a wide variety of platforms, minimizing porting and maintenance overhead as a result.

In addressing the issue of external pagers, [9] states that Parallax, which is a cluster volume manager for virtual disks, also uses external pager to provide file service. Like in the case of L4 microkernel, a failure of the Parallax server only effects its clients. Lastly, the use of a separate virtual machine Dom_0 including Xen means that VMs have to communicate with Dom_0 via simple asynchronous unidirectional event mechanism. This is analogous to a form of asynchronous IPC. An examination of CPU overhead of Dom_0 drivers under high load shows that the CPU load generated by Dom_0 accounts for approximately all of the CPU load. Hence under high load, the IPC dominates the driver overhead in Xen. [9] also cites another paper that concludes that the IPC operations carried out by an Xen-based system and a comparable microkernel are the same.

Before [10] presents the OKL4 microvisor, a system employing specific characteristics of both VMMs and microkernels, it discusses the main differences between the two approaches, which are listed below:

- *Abstractions*, While developing a VMM, the goal is to abstract resources that look much like the real ones without considering the implementation size. For a microkernel, the size is significant, however, the gap between the physical and the abstract should not be so large that it violates the minimality principle.
- *Memory*, A VMM virtualizes the Memory Management Unit (MMUs) software interface and page tables thereby providing virtual MMUs (VMMUs). Microkernels on the other hand uses network-like communication mechanisms to manage its address spaces via operations of grant, map and flush.
- *Execution*, The VMM provides each VM with a virtual CPU (vCPU) and multiplexes between them. Multiple privilege levels are retained and communication includes priorities and time slices. The second generation microkernels uses threads (as in L4) or scheduler activations (as in K42) to abstract execution time. This minimal abstractions allows for scheduling activities and also enforces priorities and time slices. Microkernels treats all user-level activities as peers.
- *I/O*, The hypervisor uses multiplexing and virtualizes devices by exporting a device interface to the guest OS while the actual device drivers resides in the hypervisor itself. In contrast, since device drivers are implemented at user level by microkernels, the drivers are run as user-level processes and communicates with the rest of the system via IPC.
- *Communication*, Virtual Machines (VMs) communicate like individual machines would via networks. This mechanism is provided by the hypervisor implementing virtual networks running on standard networking stacks. Microkernels use highly-optimized message-based IPC to communicate between VMs. This mechanism minimizes overhead by utilizing synchronous IPC.

III. Basic Implementation of Virtualization

This section provides details of the three approaches to carry out virtualization. The first is the microkernel approach where the main focus is on porting Linux in para-virtualized form on top of the L4 microkernel, also known as L4Linux. The second is the hypervisor approach, where the focus is on Xen and its performance evaluation for embedded systems. Finally, the microvisor approach is introduced which is said to be the convergence point of microkernels and hypervisors, employing features of both approaches.

A. The Microkernel Approach

Security in embedded systems has received renewed interest with the increase of personal computing devices like Personal Digital Assistants (PDAs), Smartphones, and the likes. Embedded systems vendors apart from balancing the ”apparent” power, features and flexibility the software platform provides to mobile device users, also have to take steps against the possibility of mobile devices being compromised either by users themselves or other third-party software. In 2004, a Symbian ”cabir” worm showed that the embedded system environment is not safe from the viruses and worms which are currently present in the desktop PC world.[11]

One of the major approaches to address the security and other problems discussed above is the use of small kernel, small components and small interfaces approach named Microkernel approach. This approach uses the ”Principle of least privilege” and the ”Principle of economy of mechanism” [12]. The current main stream OS kernels have full privilege on the system hardware it supervises. The above mentioned principles are applied in the microkernel by keeping it as small as possible still providing the basic mechanisms needed to implement an OS on it. These mechanisms include low-level address space management, thread management and interprocess communication.

The other important software like device drivers, protocol stacks, file-systems and user interface code etc are provided in the user space with no direct access to the

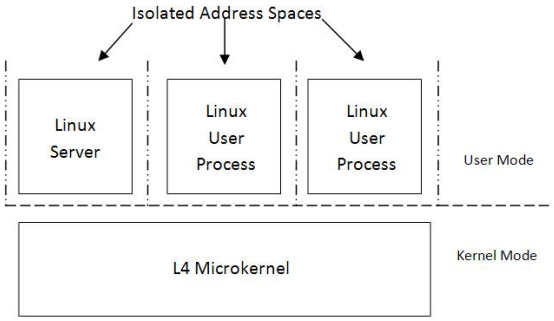

Fig. 2. L4Linux Architecture [15]

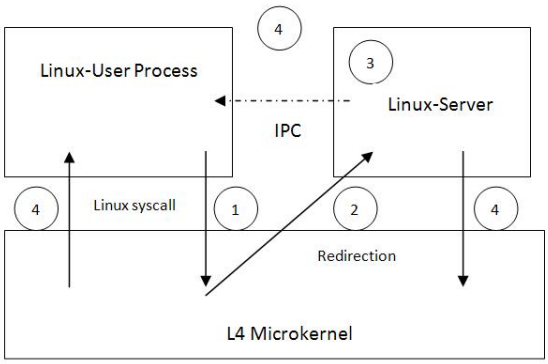

Fig. 3. System Call on L4Linux

hardware. So a small kernel with proper mechanisms enabling security-policy enforcement can provide a system basis that guarantees a high degree of confidence in operation.

A small kernel doesn't necessarily provide a secure system, but security can be provided by isolating the system services, provided above the microkernel, into cells. The security policy could then be enforced by the microkernel regarding the communication between these cells. So the security critical software placed in a cell could be assured by not allowing any other cell to gain access to the address space of the secured cell resulting in a secured system.

In order to place an existing OS on top of a microkernel, many modifications of the kernel of the guest OS are required to make it de-privileged. These changes include modification of system call interface, memory management, and interrupt handling. The examples of this virtualization approach are Xen and L4 microkernel based virtualization of Linux (L4Linux). [13]

L4Linux is a para-virtualized Linux on top of L4 microkernel with Linux completely modified according to the virtual machine interfaces given by L4. Linux was first ported to L4 in 1996 at TU Dresden [14]. Since then, it has been continuously updated to the latest Linux version. Current versions also support embedded processor architectures, e.g. ARM processors. The latest L4Linux version, at the time of writing this report, is L4Linux 2.6.34.

Architecture and Model

L4Linux is implemented with client-server approach (as in data communication) shown in Figure 2.

From the figure, all the servers and user applications are placed inside the isolated cells with isolated address spaces. User applications and processes can only communicate with Linux servers via IPC, one of the most important solutions provided by the L4 microkernel. As a consequence to this, L4 servers are isolated from the untrusted user processes and the rest of the system. Due to this virtualization approach, security of the system is enhanced. Due to the isolation of user processes from the rest of the system, even when guest OS in a VM is compromised, the rest of the system is protected. This is an important aspect of isolation, e.g. even if a downloaded virus takes control of the guest OS, telephone calls still can be made as the modem software is protected from the attack.

System Calls & Signaling

As the Linux-server is isolated from the user processes, they can't communicate directly as they could do in native Linux. The communication is supported by a mechanism called syscall redirection. Whenever a user process triggers a Linux system call, L4 treats this as an exception and redirects the call to the Linux-server. Upon receiving the redirected system call, performs the required job and replies to the user process by IPC. Page faults and interrupts are also handled in the same way i.e. by passing them to the user-level Linux server via IPC. The complete process is shown in Figure 3.

B. The Hypervisor Approach

System virtualization means creating virtual machine by virtualizing a full set of hardware resources, including a processor, memory, storage resources and peripheral devices [16]. And virtualization by its own increase security of the system by running isolated multiple virtual machines under hypervisor, so that any compromised guest OS cannot be propagated to other guest OS domains. It also provides interfaces, called hypercalls, to isolate guest OSes in a secure way.

The approach of using a hypervisor to provide virtualized environment in embedded system that will be analyzed in this paper is by using Xen hypervisor that is embedded to ARM CPU which is called Xen on ARM. Xen has an implementation of architecture independent common components of VMM such as hypercall interface, VM scheduling, VM controls and memory management. Xen makes a privileged VM, called domain0 or Dom_0 in order to manage other guest VMs, called domainU or DomU [17].

Xen provisions VCPUs per domain, including information of scheduling and event channel. Xen also utilizes hardware protection mechanisms to isolate guest OS kernels and its applications. Figure 4 shows the Xen on ARM implementation between underlying physical layer and VM interface in Xen architecure.

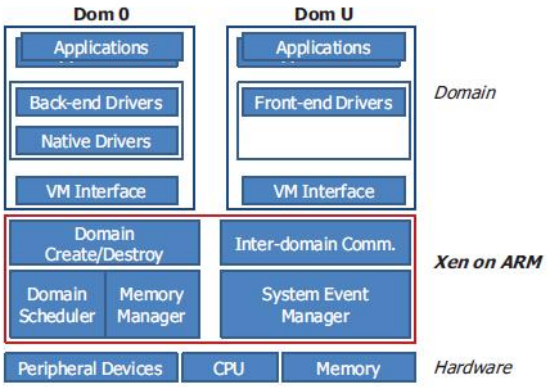

Fig. 4. Xen architecture with Xen On ARM [17]

Porting Xen to ARM

Xen could not be directly ported to ARM because of limitation in ARM's virtualization support compared to x86. The x86 has rich function designed for desktop and servers. And compared to x86's virtualization capability [18], ARM CPU has only one unprivileged mode, i.e., user mode and six privileged mode, i.e., FIQ, IRQ, Supervisor, Abort, Undef, System. However, [19] solves this problem with several steps to port Xen hypervisor and associated Linux guest OS to ARM processors. First, repeatably install Linux distribution on QEMU ARM emulator which compiles Linux kernel to the applied Xen patches. It follows by booting Xen VMM under QEMU. And last step is load the Linux guest OS under Xen VMM on QEMU.

Virtualization Approach by Xen

To allow full resource control by VMM and deprivilege guest OS, VMM runs in supervisor mode while guest OS runs in user mode, together with user applications. It causes difficulty to protect guest OS kernel from user applications since they run in the same mode. Hence both of them should be isolated in the kernel's CPU context and kernel memory context.

- *CPU virtualization*
  Since paravirtualization requires OS modification, providing an abstract supervisor mode to guest OS kernel can be done to minimize the modification. User mode is split to two logical modes, i.e., user process mode and kernel mode, and virtual banked register for those virtual privilege modes. Xen on ARM do the switching between user process mode and kernel modes. The transition is called VCPU mode transition. Figure 5 illustrate the VCPU mode transition diagram and Figure 6 depicts the process of exception handling.

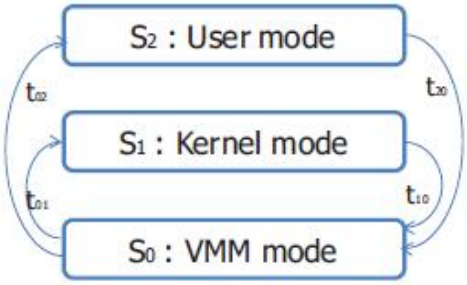

Fig. 5. VCPU mode transitions. t10: on interrupts/faults/aborts/hypercalls, t20: on interrupts/faults/aborts/systen calls, t01: upcall or return from exception, t02: return from exception [17]

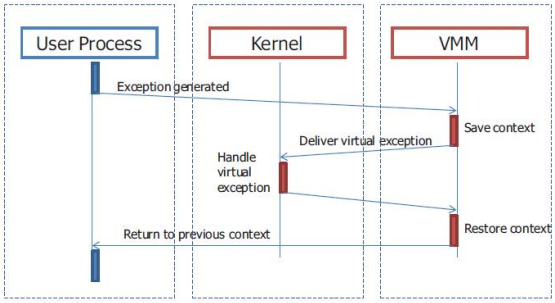

Fig. 6. Exception Handling [17]

VMM mode of the system kernel is activated when exceptions such as interrupt, fault, abort and software interrupt occurs. On entry to VMM mode, ARM CPU's SPSR (Saved Program Status Register) is saved in its virtual system, VSPSR (Virtual SPSR) which will be restored later when the guest OS returns from the exception. The stack pointer register is also saved in the VSP register for later restoration. Xen invokes an upcall to deliver exceptions to kernel mode as virtual exception events. On upcall, the VSPSR information is put on kernel's stack in order for the kernel to perceive the last running virtual processor mode. The upcall mechanism corresponds to hardware level exception handling that makes CPU to jump into exception vector table. The Xen on ARM provides virtual FAR (Fault Address Register) and virtual FSR (Fault Status Register) to guest OS due to limitation of sensitive register access to those files in the kernel. All sensitive instructions contained in guest OS will be replaced with proper hypercalls. Guest OS then invokes hypercall to return from the exception and

Xen restores the saved context.

- *Memory virtualization*
  Xen on ARM ensures isolation between guest domains by creating guest domain's memory mapping and is only updated by Xen. To modify memory mapping, guest OS should invoke hypercall to update page table. Any access to memory area that is not granted for the guest OS will be denied.
  Guest OS should run in user mode and does not have privilege to manipulate MMU. To do this, guest OS invoke hypercalls to control MMU which is the validated by Xen on ARM. Any attempt to access other domain's memory page is prohibited by VMM. Though a domain is compromised by malicious software, it cannot attack other domain as far as VMM is not compromised.

C. The Microvisor Approach

An explanation of the microvisor design is provided in the section below. It also includes a detailed look at a successful implementation of the OKL4 microvisor in the Motorola Evoke QA4 mobile phone which has been in the market since 2009.

IV. The Microvisor

In [10], an approach that implements features from both the hypervisor and the microkernel approaches is proposed. One of its important features is that it is designed to be portable across a range of architectures including ARM processors for effective use in embedded systems. Keeping the trend to employ virtualization which can support legacy device drivers and legacy OS environments while providing security, the goal of the microvisor is to be a single kernel that can provide platform virtualization with the benefits of both worlds; the efficiency of the best hypervisor and the generality and minimality of the microkernel.

Developed by Open Kernel Labs, the aim of the OKL4 microvisor is to serve as the hypervisor as well as the microkernel of performance sensitive and memory-constrained embedded systems. Inspired by the seL4, it also allows for access control of all resources and a mechanism for formal verification.[10] Unlike the Xen, the OKL4 has a much smaller trusted computing base (TCB) and accounts for nearly $1/10^{th}$ of the size of the Xen hypervisor in terms of LOC. [20]

The microvisor hosts a number of advantages some of which include the following[21]:

- A small microkernel base provides easier manageability.
- A verifiable correct implementation of the L4 concepts. The OKL4 microvisor's code is based on seL4, which has been verified in June 2010. Having seL4 as its base, the OKL4 is impenetrable.
- User-space virtualization of guest OSes, network stacks, etc.
- Secured implementation of device drivers, stacks and other privileged code.
- Implementation of fast and secured IPC threads for communication among Guest OSes and other partitions.

A look at the resulting architecture of the OKL4 microvisor is presented by the following model. [10]

- *Execution*, The execution abstraction of the OKL4 microvisor is that of a VMM. It employs one or more virtual CPUs (vCPUs) with the guest scheduling activities discussed earlier.
- *Memory*, Like that of a VMM, the OKL4 microvisor implements virtual MMUs(vMMUs) with a virtual TLB(vTLB) that is larger than the real TLB. The TLB is implemented as a page table and is traversed by the microkernel incase of a page fault. The guest uses the vMMUs to map virtual to physical memory.
- *I/O*, The I/O abstraction consists of memory mapped virtual device registers and virtual interrupts(vIRQs).
- *Communication*, The vIRQs (for synchronization) and channels are used. The OKL4 microvisor implements the asynchronous IPC model since it adopts better to large, real-world embedded systems compared to the traditional L4's synchronous IPC. This is because embedded devices hosts multi-MLOC modern stacks and mobile device application environments.

The OKL4 microvisor has been succesfully implemented in the Motorola Evoke QA4 mobile phone which has been in the market since 1 April 2009. It uses the same OKL4 microvisor that has been described above and hosts unique aspects and designs that would not be supported by traditional hypervisors. The Evoke runs Linux as the operating system(OS) supporting the user interface(UI). The baseband stack runs outside of the Linux and components from the BREW [Qualcomm] UI framework that were needed to be reused were not ported on to the Linux. Virtualization was used to make all of these features possible. The Linux runs on a separate VM from the baseband stack and BREW, on top of the OKL4 microvisor. Interaction between the two virtual machines takes place via the OKL4 using message-passing IPC and shared memory. Both the complete Linux system and the AMSS/BREW baseband stack and OS are de-privileged and runs in user mode. The architecture used for the Evoke is depicted in Figure 7.

The virtualization overhead is insignificant due to the high-performance of the OKL4 Microvisor and is better than that of the Native Linux[2]. This is due to the fast-context-switching technology provided by OK labs. In contrast to the previous reservation that the lack of

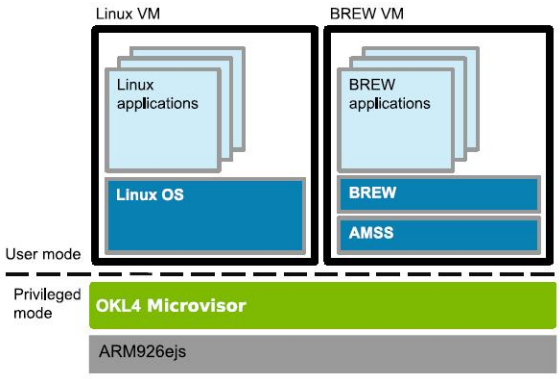

Fig. 7. Software architecture of Evoke [2]

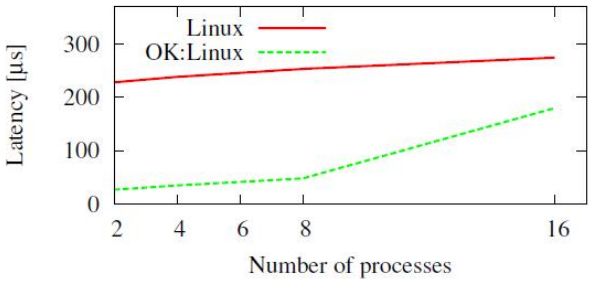

Fig. 8. Context-switch latency as measured by LMbench for native and virtualized Linux [2]

address-space ID (ASID) tag in the TLB of ARM9 processors means that the ARM9 flushes TLBs and caches on each context-switch leading to poor performance, the smart management of the address space and the utilization of the ARM9 MMUs avoids this without sacrificing performance. This can further be validated from Figure 8. As can be seen, the context-switch latencies in OK:Linux are greatly reduced compared to the native Linux. This difference is more pronounced when the active processors are small which is a common scenario for mobile phones. This is an advantage for the Evoke's user-interface. All the user-interface functionalities including the touch screen is owned by the Linux apps while video rendering uses a rendering engine running on BREW. When a user requests a BREW app, Linux communciates with BREW in the other VM to start up the app. The BREW obtains access to the screen by using a frame buffer from a shared-memory mapping as shown in Figure 9. This integration of the two subsystems is seamless and is possible because of the two critical features of the OKL4; quickly establishing and sharing memory between the VMs and the fast message-passing based IPC. All these amenities were made possible only because of the integration of the OKL4 microvisor.

The table in Figure 10 compares the main features of microkernels, hypervisors and microvisors. This comparison represents a summary of the discussion above and detailed explanations of the security aspects and the features of the Trusted Computing Base(TCB) are described in Section VI.

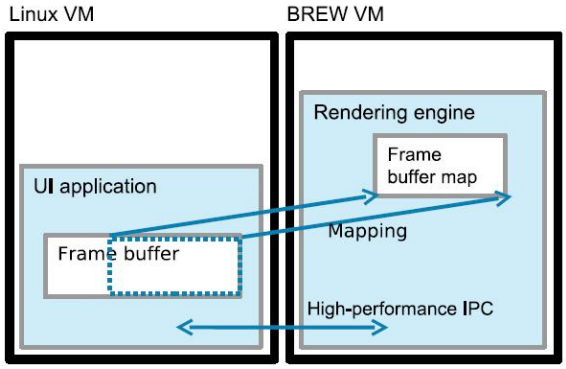

Fig. 9. Memory sharing and IPC across virtual machines[2]

	MICROKERNEL	HYPERVISOR	MICROVISOR
MEMORY	OS-like concept of address space	Uses vMMUs and virtual Page Tables	Similar to the hypervisor uses vMMUs which contains vTLBs
EXECUTION	Uses threads or scheduler activations to abstraction execution time	Each VM is assigned a vCPU and the VMM multiplexes between them	Similar to the hypervisor uses vCPUs with guest scheduling
I/O	Run device drivers as user level processes, communicate with rest of system via IPC	Uses virtual devices by exporting device interface to guest OS, with actual device driver resides inside hypervisor	Similar to the microkernel. Uses memory-mapped virtual device registers and vIRQs
COMMUNICATION	Uses synchronous IPC to communicate between VMs.	Virtual networks based on existing virtual-device abstraction	Uses IPC like the microkernel but implements asynchronous IPC unlike the microkernel
ABSTRACTION API	L4 API	Abstract hardware machine	OKL4 API
DEVICE DRIVERS	Outside kernel, in user space	Separate VM (D0M0)	Outside kernel, in user space
SECURITY	Capability-based Access Control	Mandatory Access Control	Capability-based Access Control with Secure HyperCell Technology
TRUSTED COMPUTING BASE	seL4	None	seL4

Fig. 10. Table of Comparison between Microkernels, Hypervisor and Microvisor

V. Performance Comparison

Benchmark defines as the test performance of certain parameters on a computer program or a set of programs by running a number of standard tests and trials against it. There are two categories of benchmarks; micro-benchmarks and macro-benchmarks. Micro-benchmarks measure very small and specific units of work, e.g. latencies of executing specific Linux system calls or how a system is able to switch between two processes. On the other hand, macro-benchmarks measure more

Test Cases	Native Linux (in microsecond)	L4Linux (in microsecond)
Simple syscall	0.7158	22.3718
Simple read	3.6399	34.3318
Simple Write	3.1738	30.9641
Simple stat	16.2348	106.0476
Simple fstat	4.2553	44.4267
Simple open/close	39.0611	208.7772
Select on 10 fd's	4.3945	50.335
Select on 100 fd's	28.5861	82.4257
Select on 250 fd's	68.8872	122.4306
Select on 500 fd's	136.3443	221.5062
Signal handler install	2.619	36.2203
Signal handler overhead	9.8543	136.0835
Protection fault	4.0656	43.9095
Pipe	127.8178	698.4668
AU_UNIX sock stream	205.9367	876.9344
Process fork+exit	5213.2701	54736.8421
Process fork+execve	15915.493	110000
Process fork+bin/sh-c	45000	232000
pagefaults	46.0907	307.4131

Fig. 11. System Latencies Comparison of Native Linux and Paravirtualized Linux on L4Linux [15]

Test Cases	Native Linux (in microsec)	L4Linux (in microsec)
2p 0k	40.02	132.9
8p 0k	69.53	180.04
2p 4k	63.29	169.1
8p 4k	121.35	236.16
2p 8k	100.10	196.84
8p 8k	158.13	263.18
2p 16k	158.94	231.11
8p 16k	189.60	276
2p 32k	107.83	176.37
8p 32k	136.71	225.43
2p 64k	104.52	186.83
8p 64k	140.77	244.75

Fig. 12. Context Switch Latencies Comparison of Native Linux and Paravirtualized Linux on L4Linux [15]

substantial and high-level units of work that are more closely aligned with real-world workloads.

In the subsections below, both benchmarking types will be analyzed, for native Linux and paravirtualized Linux with different virtualization approach, i.e., Xen, L4Linux and OKL4.

A. Micro Benchmarks

Comparison of Native Linux and Paravirtualized Linux on L4(L4Linux)[15]

The test is run by using LMbench analysis tool. From the Figure 11, there is significant overhead noticeable in simple syscall benchmark on L4Linux compared to native Linux, i.e., about 30 times slower than the one in native Linux. The reason for the significant overhead is as follows. On native Linux, both user tasks and kernel share same address pace. Each system calls costs nothing but a CPU mode change. On the other hand, on L4Linux, Linux server and user processes are isolated from each other in different address space where each system call costs 2 kernel entry/exit pairs plus 2 address space switches which are time consuming procedures.

The significantly large overhead of the address space switches and the kernel entry/exit are a direct contribution from three mechanisms: pure kernel instruction executions, cache and TLB flushing. The ARM1176JZS processor where this evaluation is carried out, the processor uses physically indexed cache and hence flushing is not necessary during address space switches. In addition, frequent TLB flushing is not necessary on the current version of L4Linux, since the Address Space Identifier (ASID) is used. Consequently, the TLB can be retained across all the address spaces. Therefore the significant overhead results because of executing all the additional kernel instructions when kernel entry/exit and address spaces switches are executed with the system calls.

In L4Linux a signal-handler is implemented to avoid direct inter-thread manipulation in signal handling, making it more expensive than the signal handling implemented on native Linux. In the table in Figure 11 the benchmarks, fork, fork+execv and fork+sh are related to process creation and execution. These processes need to update the shadow page tables in L4Linux. Implementing shadow page tables and mapping guest virtual addresses to host physical addresses, adds a lot of overhead to these operations.

Furthermore, [15] analyzes the context switch latencies both on native Linux and L4Linux which are displayed on Figure 12. The values on the table shows that without cache footprint, one context switch on L4Linux takes approximately 3 times longer than on native Linux. For longer processes, the context switch overhead is dominated by the cache interference, hence the overhead becomes lower as the size of the processes increases. It can be also noted that the context switch on L4Linux will always be longer than on native Linux.

[15] also investigates performance of memory access which is shown in Figure 13 where there is only slightly difference between native Linux curve with L4Linux curve. This is because for this evaluation, a loop that adds a set of integers is executed. This process does not require kernel service, hence no overhead is added to the system. L4Linux does not emulate or intercept any instructions, the performance is close to the native Linux when the user process executes computing intensive tasks. This is an advantage of the L4 virtualization approach.

The cache on the ARM1176JZS processor is read allocate. This is the reason why the memory write band-

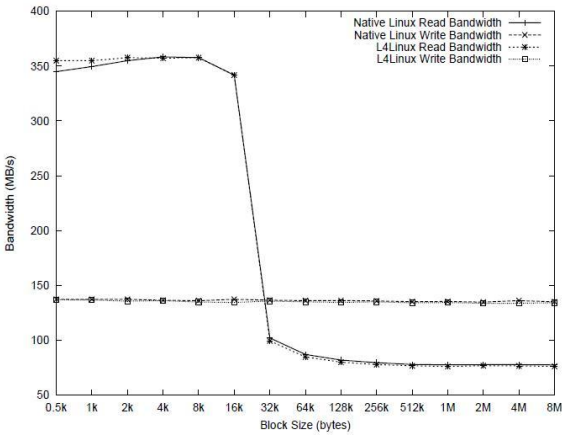

Fig. 13. Memory Access Latencies Comparison of Native Linux and Paravirtualized Linux on L4Linux [15]

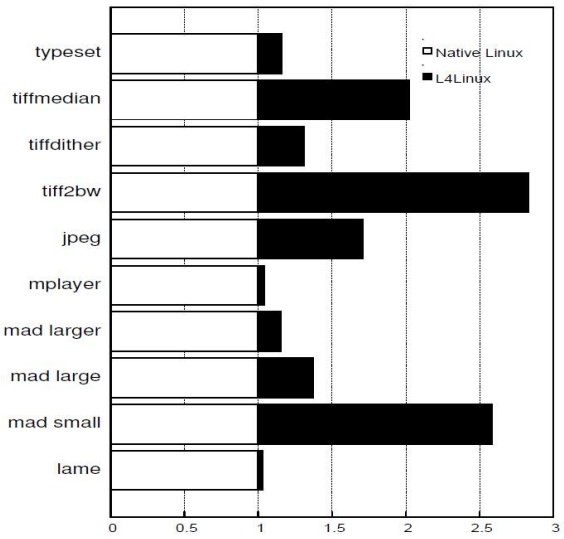

Fig. 14. Virtualization Overhead of Multimedia Applications [15]

width curve for the L4Linux and native Linux are flat, as data accessed in the write bandwidth benchmark are not cached during execution; hence it is not affected by cache size. In contrast, a sharp decrease can be seen for the memory write bandwidth for the block size of 16k bytes. Since the level 1 cache for ARM11 is 16k bytes, the read bandwidth plummets when the accessing memory block is larger than 16k bytes.

Since LMbench only measures the performance of basic system operations and does not provide measures for real application scenarios, it is necessary to carry out a similar evaluation under more realistic environments. Hence, a series of application specific benchmarks from MiBench have been investigated in [15] to evaluate L4Linux. These benchmarks cater to three different groups of applications: multimedia applications, office automation and telecommunications. The benchmarks are executed and timed both on L4Linux and native Linux. The virtualization overhead of specific applications are shown in Figure 14 and Figure 15. In contrast to the performance from LMbench, the performance here are very close to the Native Linux for most of the benchmarks here. The overhead for lame and mplayer is only about 3%, since these two applications are CPU bounded where systems calls are not often triggered. Hence, large overhead due to system calls are not an issue here.

Furthermore, like in the LMbench performance for memory access bandwidth, the CPU does not emulate or intercept instructions and hence most of the instructions are executed on the CPU reducing the virtualization overhead even more. Other computing intensive applications such as jpeg and mad small have much larger overhead than lame and mplayer. It is also seen from Figure 16 that benchmarks with high overhead has short execution times compared with benchmarks with low overhead. This is because the absolute time for creating

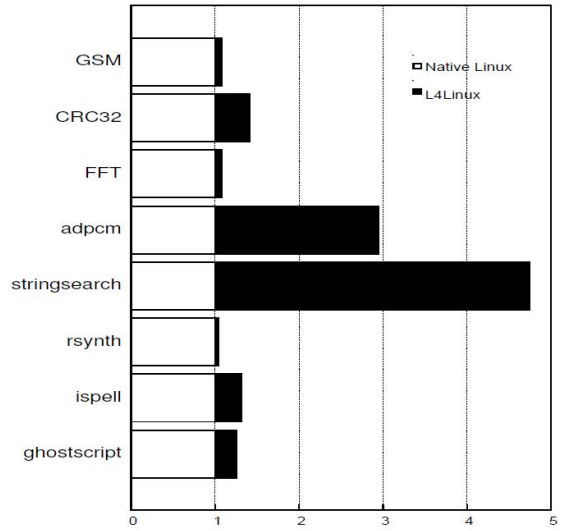

Fig. 15. Virtualization Overhead of Office Automation and Telecommunications Applications [15]

process and destroying process dominates when execution time is short. This results in decreased overhead when a larger input file is used to extend the execution time of the benchmarks. This trend is clearly illustrated in Figure 14 for the three mad benchmarks.

From all the observations put forth by [15], it is concluded that the virtualization overhead of L4 microkernel varies from system calls to real application scenarios. While the virtualization overhead for system calls are large, they are insignificant for computing intensive applications. This is a result of the virtualization approach followed by L4Linux. Applications that implement a lot of system calls incur large overhead while applications with few system calls

Benchmarks	Native Linux (s)	L4Linux (s)
Multimedia Applications		
lame	78.69	81.1
mad (small MP3 file)	0.232	0.598
mad (large MP3 file)	1.828	2.506
mad (larger MP3 file)	7.842	9.012
mplayer	5.512	5.712
jpeg	0.84	1.44
tiff2bw	0.64	1.814
tiffdither	1.44	1.892
tiffmedian	1.742	3.52
typeset	7.86	9.118
Office Automation		
ghostscript	6.188	7.7354
ispell	6.658	8.71
stringsearch	0.07	0.332
rsynth	24.75	25.642
Telecommunications		
CRC32	0.7	1.088
FFT	10.794	11.63
GSM	9.8	10.61
adpcm	0.37	1.088

Fig. 16. Application Specific Benchmark Results [15]

will perform more efficiently.

Comparison of Native Linux and Paravirtualized Linux on Xen Hypervisor[16]

Figure 17 shows the result of test comparison for bandwidth measure and latencies measure for native Linux and paravirtualized Linux. The paravirtualized Linux is compared with native Linux running on bare metal hardware without VMM. Latencies of most OS services are actually not higher than twice of the performance in native Linux. The reason process creation takes longer times than native Linux is the large number of page table updates occur when creating a new process and hyper call is invoked for every page table entry update.

Another test on context switching latency is shown in the Figure 18 where there are four cases, i.e., native Linux, paravirtualized Linux with the TLB lockdown, paravirtualized Linux without TLB lockdown and page table separation scheme. From the observation on the figure, paravirtualized Linux context switch time has about 50 microseconds additional overhead which is appended by virtualization and considered to be moderate compared to separate page table scheme. Furthermore, TLB lockdown optimization shows slight contribution when context size is larger than 8 Kbytes.

Comparison of Native Linux and Paravirtualized Linux on OKL4 and Relative Performance of OKL4 and Xen[22][10]

Along with introducing the OKL4 with a description of its model, [10] also provide performance evaluation

Bandwidth measured in Mbps			
Tests	**Native Linux**	**Paravirtualized Linux**	**Ratio**
bw_pipe	42.79	38.21	0.893
bw_unix	44.52	39.18	0.880
bw_mem 512 rd	1033.42	1075.79	1.041
bw_mem 512 wr	1034.67	1019.27	0.985
bw_mem 512 rdwr	1034.67	1019.27	0.985
Latencies measured in microseconds			
Test	**Native Linux**	**Paravirtualized Linux**	**Ratio**
Lat_pipe	135.13	234.42	1.735
Fork+exit	2891.75	10021.0	3.465
Fork+execve	3109.25	10524	3.385
SysV semaphore	45.974	81.42	1.77
Lat_unix	251.41	431.85	1.70
Signal handler	11.23	20.43	1.82
Null syscall	1.13	2.83	2.50
Read syscall	2.60	4.94	1.90
Write syscall	2.25	4.16	1.85

Fig. 17. Bandwidth and Latencies Measure for Comparison of Native Linux and Paravirtualized Linux on Xen Hypervisor [16]

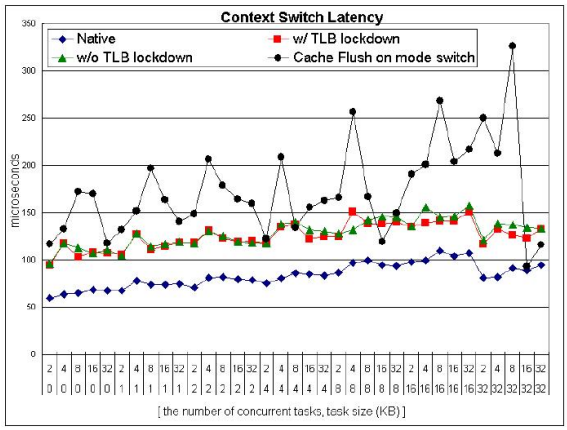

Fig. 18. Context Switch Latency in Xen [16]

of an implementation of the OKL4 microkernel on a processor based on ARM v7 architecture. The LmBench results are presented in Figure 19.

The majority of the benchmarks require a single Linux system call, which needs a single OKL4 hypercall to virtualize. The basic overhead is around 0.3μs per hypercall. On the other hand, since the IPC and process creation benchmarks are complex, they require multiple hypercalls and hence incur larger overhead. [10] compares the LmBench results of Xen on a ARM v5 processor from [16] and the OKL4 microkernel on the same processor. An overhead of 35% and 27% was observed for fork and fork+execv. The corresponding values on Xen were around 250%). The negative overhead values for open/close and signal handling and high values for protection fault in Figure 19 are as a change in the layout of the virtualized system and are thus of no significant concern.

Figure 20 shows the result of running Netperf on both

Benchmark	Native	Virtualized	Overhead	
null syscall	0.6 μs	0.96 μs	0.36 μs	60 %
read	1.14 μs	1.31 μs	0.17 μs	15 %
write	0.98 μs	1.22 μs	0.24 μs	24 %
stat	4.73 μs	5.05 μs	0.32 μs	7 %
fstat	1.58 μs	2.24 μs	0.66 μs	42 %
open/close	9.12 μs	8.23 μs	-0.89 μs	-10 %
select(10)	2.62 μs	2.98 μs	0.36 μs	14 %
select(100)	16.24 μs	16.44 μs	0.20 μs	1 %
sig. install	1.77 μs	2.05 μs	0.28 μs	16 %
sig. handler	6.81 μs	5.83 μs	-0.98 μs	-14 %
prot. fault	1.27 μs	2.15 μs	0.88 μs	67 %
pipe latency	41.56 μs	54.45 μs	12.89 μs	31 %
UNIX socket	52.76 μs	80.90 μs	28.14 μs	53 %
fork	1,106 μs	1,190 μs	84 μs	8 %
fork+execve	4,710 μs	4,933 μs	223 μs	5 %
system	7,583 μs	7,796 μs	213 μs	3 %

Fig. 19. LmBench results for OKL4 [10]

Type	Benchmark	Native	Virt.	O/H
TCP	Xput [Mib/s]	651	630	3 %
	Load [%]	99	99	0 %
	Cost [μs/KiB]	12.5	12.9	3 %
UDP	Xput [Mib/s]	537	516	4 %
	Load [%]	99 %	99 %	0 %
	Cost [μs/KiB]	15.2	15.8	4 %

Fig. 20. Netperf results for OKL4 [10]

the Native Linux and Virtualized Linux on OKL4 for an ARM v7 processor. Both systems show a fully-loaded CPU and the throughput degradation of the virtualized was only 3% and 4%. The paper claims that these values are the lowest overhead on a virtualized Linux system, even when compared to figures available of virtualized Linux on other architectures.

Another test on the comparison of native Linux and paravirtualized Linux on OKL4 is shown in Figure 21. The test is run with OKL4 3.0 release and OK Linux 2.6.23 on Xscale PXA255 platform at 255 MHz. Meanwhile, Xen used 266 MHz ARM926 processor and has different memory architecture. In the latencies performance, the smaller values shows better performance while in relative performance, the higher value shows better performance.

B. Macro Benchmarks

Comparison of Native Linux and Paravirtualized Linux on Xen Hypervisor[16]

The result for comparison test for latencies on native and paravirtualized Linux on Xen hypervisor is shown in Figure 22. The UI loading test is run on Qtopia PDA Edition and binaries are installed at NOR flash memory. For image file saving test, 200 files are distributed evenly from 20 kB to 90 kB, and the time taken to save all those files from NFS server to NAND flash memory

	Latencies (in microseconds)		Relative Performance	
Benchmark	Native	Paravirtualized Linux (on OKL4)	OKL4	Xen
Pipe	756.59	84.84	8.92	0.58
Fork	6469	8742	0.74	0.29
Fork+exec	59715	75515	0.79	0.30
Semaphore	261.6	21.08	12.41	0.56
Unix	1292.2	115.01	11.24	0.59
Signal handler	14.26	54.76	0.26	0.55
Null syscall	1.14	5.40	0.21	0.40
Read syscall	3.34	8.45	0.40	0.52

Fig. 21. Benchmark Performance of OKL4 and Xen [22]

Benchmark	Native Linux	Paravirtualized Linux	Ratio
UI Loading Time (in sec)	14.48	14.54	1.004
Image Saving Time (in sec)	52.71	53.81	1.020
Encoding Rate (fps)	5.71	5.63	0.986
Decoding Rate (fps)	24.54	24.41	0.995

Fig. 22. Macro Benchmark Performance Difference for XEN [16]

is measured. And for the codec test, Xvid MPEG4 stream encoder/decoder is used.

Comparison of Native Linux and Paravirtualized Linux on OKL4[22]

The result for comparison test on native and paravirtualized Linux on OKL4 is shown in Figure 23. The benchmark used is ReAIM. The test is run on PXA22 at 400 MHz. The ratio shows slight difference in performance. On ARM processor, the IPC performance for a one-way message passing operation is less than 200 cycles. This high performance provides efficient mechanism for setting up shared memory regions.

VI. Security Performance

A. Security provided by Virtualization and Isolation

The most important aspect of virtualization is security, which is provided by isolating different applications and sub-systems from each other i.e., by placing them inside secured cells. As a result of this isolation, the security policy can be implemented in the microkernel and then enforced globally. The communication between cells will then take place only when the security policy of the microkernel allows it.

In order to ensure security further, minimum software should run in the privileged mode of the system, as

Benchmark	Native Linux	Paravirtualized Linux (on OKL4)	Ratio
1 Task	45.2	43.6	0.96
2 Task	23.6	22.6	0.95
3 Task	15.8	15.3	0.96

Fig. 23. The Performance Comparison Between Native Linux and Linux on OKL4 [22]

in privilege mode, this software has complete control over the application's resources. So apart from being small, this software should also be the most trusted and most secured component of the entire system because its security places a limit on the overall security of the complete system. By reducing its size, one can assure its security and reliability to a much higher level. So by using a secured microkernel, one can build the rest of the system on top of it according to the established security principles like principle of least authority which states:

"Any Piece of code should not have any more access rights capabilities/authority than it's absolutely required to do its job"

This principle means as the access rights to other subsystem are implemented via "capabilities", so if a service running inside a secured cell doesn't need access to some other subsystem (e.g., sound driver) to complete its assigned job, then it shouldn't be given access rights capability for that subsystem.

All the subsystems can then be encapsulated in their own hardware address space and can be allowed to communicate with other parts of the systems in accordance with the implemented system wide security policy described in a security module and enforced by the microkernel.

Currently, the most secure microkernel present in the market is Open Kernel Lab's SeL4 which has been mathematically verified to be completely fault-free and impossible to crack [23]. OK labs have used this microkernel as the building block of its well known industrial strength OKL4 Microvisor.

OKL4 microvisor is a clean, from-scratch design and implementation. It shares code modules with the presently available OKL4 microkernel and SeL4. It's designed to support a mixture of real-time and non-real-time software running on top. Current implementation consists of about 8.6 kLOC of ANSI C and about 1.2 kLOC of assembler compiling to about 35 KiB of text [10].

B. Security on OKL4 Microkernel

The Open Kernel Labs has a solution named OK Labs Secure Hypercell Technology. This technology enables systems engineers to design-in software to ease development, maintenance, deployment, security and reliability of mobile devices by fulfilling the requirement of infrastructure to build complex software systems from multiple simpler subsystems with fine-grained control over resource allocation, communication and security.

Secure HyperCell Technology supports flexibility of security policy implementations that offer highly secure environment for subsystems requiring one. Fault isolation created by dividing complex software across multiple cells could also increase reliability of the system, while integrity is assured by limiting privileged-mode code to OKL4 Microvisor. It reduces the fault chances since operating system kernels and device drivers are moved to user level where any faults or security breaches are better contained.

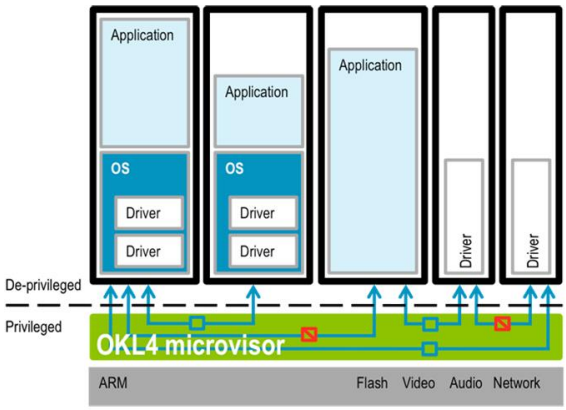

Fig. 24. OKL4 Architecture with capabilities [OK-Labs]

OKL4 microvisor provides encapsulation of subsystems into separate cells having their own address spaces which are not accessible to any other cell. Cells are able to host multiple types of subsystems, including complete OS and application environments (i.e., traditional virtual machines), individual functional subsystems (e.g., authentication code, mobile payment services or media processing), as well as device drivers. OKL4 provides resource control by kernel-protected capabilities. A capability is a kind of key or token of authorization for access between different subsystems residing in separate cells. These capabilities are subject to system-wide security and resource-management policies defined by the system designer. Also, capabilities are static and can't be changed on the run-time. OKL4 doesn't provide any services or policies and operate in the privileged mode of the system. [24]

OKL4 servers, residing in the user level cells, provide services and policies. They are responsible for system libraries, device drivers, resource management (memory, threads & address spaces) and operate in de-privileged or user mode. The OKL4 microvisor architecture is shown in the Figure 24.

The figure shows a typical use of OKL4 microvisor with two application OSes, running in separate protected domains/cells, one application running directly on top of microvisor and drivers placed in different cells. Now capabilities being keys to access other cells, a system designer can use them to implement security or access rights policies in the system. In the above figure, the application can only access the device driver that it has capability for and no other cell can communicate with it.

Furthermore, OKL4 provides:

- *User-level cells containing system components com-*

pletely isolated, as a result of hardware-enforced address space isolation. Components may include complete OSes, lightweight execution environments, device drivers, applications, etc.
- *Complete control over communication between cells*, which is required for mandatory access control.
- *Capability based protection mechanism*, i.e. giving applications fine-grained control over access to data and code, where OKL4 operates as a security monitor controlling access to memory and controls any communication between the subsystems and ensures that they can only be in accordance to the system-wide security policy established by the developer.
- *Bullet-Proof Firewalls* between subsystems due to the correctness of the underlying virtualization software, proven to be fault-free and impossible to crack. OKL4, being unpenetrable and as reliable as hardware itself, allows separation of user level components into fire-walled cells/boxes.
- *Separation into cells* allows independent verification of components and fault containment thus helping in debugging the component with pinpointing the problem area.
- *Fast context switching & high performance IPC.*
- *Trustworthy Computing Base* being small due to the componentization of system, leads to higher robustness against failures as a small TCB is a requirement for devices where failure is not an option.

The performance of OKL4 is often close to the hardware limit as it features OK Fast Address Space Switching Technology (FASS) taking advantages of ARM domains and Fast-Context-Switch Extension (FCSE) providing "Faster than native" virtualized Linux context switching performance on ARMv5 [25]. OKL4 implements IPC mechanisms in assembly "fastpaths" to achieve zero-copy overhead, thus providing low latency interrupt handling. The high performance inter-cell communication is provided by communication channels built over shared memory and lightweight OKL4 IPC allowing highly cooperative yet modular systems without sacrificing performance. OKL4 microvisor design also provides strong real-time guarantees by providing lightweight primitives backed by short kernel execution paths which enables consolidation of real-time and non-real-time components on a single processor core.

C. Security on Xen

Xen, as the hypervisor, has potential to become solution for trusted computing by its own properties, such as open source VMM, small size in terms of code size (about 40KLoc in 2.0 version and reduced to 20KLoC in 3.0 version), high performance implementation, and its ability to run existing application software and support isolated security services.

In order to enhance its security, Xen has launched XenSE (Xen Security Enchanced). XenSE aims to supports wide range of uses such as firewall / IDS domains, Virtual Private Machines (VPMs), conventional MLS systems. The work areas of XenSE are explained as followed.

- *Mandatory Access Control*, such as adding MAC to Xen subjects or objects and IBM sHype patch great start
- *TPM support*, such as trusted or secured boot and TPM virtualization
- *Minimizing the TCB*, includes reviewing the Domain (dom0 kernel, dom0 root), adding fine-grained access control, and refactoring domain0.
- *Devices performance and devices' security*, this is done since the device drivers are a major cause of instability in OSes.
- *Other issues with user-interface*, such as dilemma between user convenience and security.

In [26], secure Xen on ARM for beyond 3G mobile phones is explained, in which Xen implements drivers separation to guarantee its security. The goal is to produce light-weight secure virtualization technology for 3G mobile phone by several approaches. approach security design by secure boot, secure software installation, multi-layer fine-grained access control. By implementing Xen on ARM architecture, the performance ia improved and applications are separated into separate domains. From security point of view, the improvement is achieved by having 5 access control modules and visualization supported, as well as access control mechanism for applications to prevent the phishing attacks.

Furthermore, the architecture is taken into one step extension, i.e., the device driver separation. Device driver domain is separated from Dom0(security applications running on Dom_0 in secure Xen on ARM) kernel. Modifications for this architecture includes: RAMFS used for driver domain during booting, Xenbus,Xenstore and Xen tools modifications and modification on booting procedure, i.e., booting Dom0, followed by creating Device Driver Domain and initializing split device driver.

VII. Conclusion

In this paper we elaborate three different approach to implements virtualization, i.e., by using microkernels, hypervisors and the latest technology from OK labs, microvisor. By general requirement, microkernel supports implementation of a hypervisor. However, hypervisor and micorkernels by their functions have different purposes, structures and APIs. The OKL4 microvisor solves the problem by combining both of them, i.e., fulfill the hypervisor objective of minimal overhead for virtualization and microkernel objective of minimal size.

Furthermore, as virtualization is introduced into embedded system application, security assurance will get

more complex. But virtualization properties itself actually provides security requirement for mobile devices such as help secure mobile platforms, applications and services by carefully chosen additional components and strict isolation from one VM to another and hence reduce the risk from potential threats. Microkernels, on the other hand, assure security by isolation and keeping trusted software to a bare minimum. With this combination, OKL4 provides Secure Hypercell technology which has performance close to the hardware limit as it features OK FASS taking advantage of the ARM domains and FCSE provides speed faster than native virtualized Linux context switching.

And with its benefits and high performance, OKL4 microvisor technology could be a promising technology to be implemented in embedded system in order to fulfill the rapid growth of demand for secure mobile phone.

ACKNOWLEDGEMENTS

We would like to thank our supervisor Martin Hell for his guidance, support and suggestions and also to Christopher Jämthagen for overseeing our work.

REFERENCES

[1] O. Kharif, "Virtualization goes mobile," Bloomberg Businessweek: Technology, 2008.
[2] G. Heiser, "The motorola evoke QA4: A case study in mobile virtualization," Technology White Paper: Open Kernel Labs, 2009.
[3] G.Heiser, "Virtualization for embedded systems," 2007.
[4] G. J. Popek and R. p. Goldberg, "Formal requirements to virtualize third generation architectures," *Communications of the ACM, 17(7)*, pp. 413–421, 1974.
[5] F. Armand and M. Gien, "A practical look at microkernels and virtual machine monitors," 2009.
[6] G. Heiser, "The role of virtualization in embedded systems," *First workshop on Isolation and Integration in Embedded Systems*, 2008.
[7] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, "Camkes: A component model for secure microkernel-based embedded systems," *Journal of Systems and Software Special Edition on Component-Based Software Engineering of Trustworthy Embedded Systems 80(5)*, pp. 687–699, 2007.
[8] S. Hand, A. Warfeld, K. Fraser, E. Kotsovinos, and D. Magenheimery, "Are virtual machine monitors microkernels done right?" 2005.
[9] G. Heiser, V. Uhlig, and J. LeVasseur, "Are virtual machine monitors microkernels done right?" *NICTA Technical Report PA005103*, 2005.
[10] G. Heiser and B. Leslie, "The OKL4 microvisor: Convergence point of microkernels and hypervisors," *2010 ACM 978-1-4503-0195-4/10/08*, 2010.
[11] Website, http://en.wikipedia.org/wiki/Caribe_(computer_worm).
[12] J.Saltzer and M.Schroeder, "The protection in information in computer systems," *Proceedings of the IEEE, 63(9)*, 1975.
[13] J. Brakensiek, A. Dröge, M. Botteck, H. Härtig, and A. Lackorzynski, "Virtualization as an enabler for security in mobile devices," *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, 2008.
[14] M.Hohmuth, "Linux-emulation auf einem mikrokern," Master's thesis, TU-Dresden, 1996.
[15] Y.Xu, F.Bruns, E.Gonzalez, S.Traboulsi, and A. B. K.Mott, "Performance evaluation of para-virtualization on modern mobile phone platform," *Proceedings of International Conference on Computer, Electrical, and Systems Science and Engineering, Penang Malaysia*, Feb 2010.
[16] e. Young Hwang, "Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones," *5th IEEE Consumer Communications and Networking Conference and Networking Conference*, pp. 257–261, 2008.
[17] S. Seo, "Research on system virtualization using xen hypervisor for arm based secure mobile phones," *Seminar 'Security in Telecommunications' of Berlin University of Technology*, January 14 2010.
[18] J.S.Robin and C. Irvine, "Analysis of the intel pentium's ability to support a secure virtual machine monitor," *Proc.9th USENIX Security Symposium, Denver, Colorado, USA*, August 2000.
[19] M. Lemay, D. Jin, S.Reddy, and B. Schoudel, "Porting the xen hypervisor to arm," *Technical Report in UIUC*, 2009.
[20] G. Heiser, "Microkernels vs hypervisors."
[21] "The OKL4 microvisor advantage," Website: Open Kernel Labs, http://www.ok-labs.com/solutions/the-okl4-microkernel-advantage.
[22] M. Bylund, "Evaluation of OKL4," *Bachelor Thesis in Computer Science, Mlardalens University*, April 2009.
[23] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," *Communications of the ACM*, pp. 107–115, 2010.
[24] O. K. Labs, *OKL4 Library Reference Manual*, 2008.
[25] C. van Schaik and G. Heiser, "High-performance microkernels and virtualization on arm and segmented architectures," 2008.
[26] S. B. Suh, "Secure xen on arm : Status and domain driver separation," Xen Summit Autumn 2007, Sun Microsystems, 2007.