Performance Evaluation of Para-virtualization on Modern Mobile Phone Platform

Yang Xu, Felix Bruns, Elizabeth Gonzalez, Shadi Traboulsi, Klaus Mott, Attila Bilgic

Abstract—Emergence of smartphones brings to live the concept of converged devices with the availability of web amenities. Such trend also challenges the mobile devices manufactures and service providers in many aspects, such as security on mobile phones, complex and long time design flow, as well as higher development cost. Among these aspects, security on mobile phones is getting more and more attention. Microkernel based virtualization technology will play a critical role in addressing these challenges and meeting mobile market needs and preferences, since virtualization provides essential isolation for security reasons and it allows multiple operating systems to run on one processor accelerating development and cutting development cost. However, virtualization benefits do not come for free. As an additional software layer, it adds some inevitable virtualization overhead to the system, which may decrease the system performance. In this paper we evaluate and analyze the virtualization performance cost of L4 microkernel based virtualization on a competitive mobile phone by comparing the L4Linux, a para-virtualized Linux on top of L4 microkernel, with the native Linux performance using *lmbench* and a set of typical mobile phone applications.

Keywords—L4 microkernel, virtualization overhead, mobile phone.

I. Introduction

Recent years, as 3G/beyond-3G mobile communication technology becomes mature, more and more mobile phones are connected to Internet. People start using mobile phones to download audio, video files as well as mobile phone applications from Internet. Typical examples are iPhone App-Store and Android Market. Such trend makes modern mobile phones no longer closed systems like the traditional ones. As a consequence, the security of mobile phones faces big challenges. At the meantime, mobile phone users tend to put important personal contents into mobile phones, such as personal information, contacts, emails, credit card number and even passwords. All these put further pressure on the security aspect in modern mobile phones.

Currently in order to address the security challenges in mobile phones, several different methods are used, such as implementing complete open devices (including open OS), separating application domain and cellular domain, using strict API level certification policies and restricting run-time environment (e.g. Java). However, all these solutions have their own limitations and restrictions [1]. To overcome these limitations, virtualization technology is chosen as another alternative to enhance the security in modern mobile phones.

Virtualization technology can date back to IBM's VM/370 system [2][3][4] in 1960's, which is the first commercial virtual machine system on the world. The initial motivation to use a virtual machine system is to support multiple operating systems and multiplex expensive mainframe hardware. A virtual machine (VM) is a duplicate of a real computer system, whose resources are fully controlled by a virtual machine monitor (VMM). The VMM provides users with an efficient, isolated processing environment, which is essential to allow more than one operating system running on one single machine. At present, virtualization technology is primarily applied on servers and workstations to help system administrators reduce management overhead. In the future, virtualization will be a solution for security and software reliability [5], e.g. in mobile phones. The enhanced security is achieved by isolating trusted data and code from malicious ones so that even though one guest operating system running on the virtual machine is compromised, the integrity of the rest system can not be damaged.

Current virtualization technology can be generally divided into two categories:

1) Full virtualization: unmodified guest OS can be run on the virtual machine directly. The guest OS does not realize that it is running in a virtualized environment. Full virtualization is usually realized either by Dynamic Binary Translation (DBT) (e.g. VMware ESX [6] [7]) or by hardware assistant (e.g. Xen [8]).
2) Para-virtualization: unlike full virtualization approach, modifications are needed to de-privilege the guest OS. Usually modifications of system call interface, memory management, and interrupt handling are necessary. The advantage of para-virtutalization approach is its high performance. Examples of this approach are Xen and L4 microkernel based virtualization approach (L4Linux).

The requirements for virtualization on mobile phones are quite different from virtualization on high performance systems. Some suitable virtualization technologies for high performance systems may become unapplicable on mobile phones due to hardware resources and power consumption limitations. For example, in full virtualization by DBT, the executed instructions are intercepted and replaced in real time. This is computationally intensive and unsuitable for mobile systems. And the hardwares dedicated for virtualization, which are usually available for PC and server systems are not yet available at the embedded market. With its high performance

Yang Xu and Klaus Mott are with Infineon Technologies AG, Am Campeon 1-12 85579 Neubiberg, Germany e-mail: {yang.xu;klaus.mott}@infineon.com
Felix Bruns, Elizabeth, GonzalezShadi Traboulsi, and Attila Bilgic are with Institue of Integrated System, Ruhr-Bochum University, ICFO-03-560 Ruhr-Bochum University D-44780 Bochum Germany email: {felix.bruns;elizabeth.gonzalez;shadi.traboulsi;attila.bilgic}@is.ruhr-uni-bochum.de .

virtue, Para-virtualization is currenlty the emerging solution for virtualization in mobile phones. Examples are Trango (current VMWare MVP)[9], VirtualLogix virtualization technology [10] and L4 microkernel virtualization technology [11]. This is substantially determined by the fact that on mobile phones, high performance efficiency is preferred because of limited resources. Among all these available embedded system virtualization solutions, only the L4 microkernel virtualization approach is open-source, which is a big advantage in research work. It gives us the possibility to deeply understand the embedded system virtualization approach and allows us to do detailed analysis and evaluation. Furthermore it provides us more space for optimization in the future.

L4 microkernel was designed and optimized for 486 and Pentium architectures. It has been proved quite efficient on x86 systems [12][13][14]. However, on ARM processors, which are usually used in mobile phones, the efficiency of L4 microkernel, especially used as a VMM, has not been extensively investigated yet. In order to narrow this gap and to get exact performance data of L4 microkernel based VMM on mobile phones, we evaluated the performance of L4 Fiasco microkernel (Re-implementation of L4 microkernel from TU Dresden. In the rest of this paper, we call it L4 for short) as a VMM on a modern mobile phone platform by comparing the performance of L4Linux and native Linux. The experiment results presented not only the virtualization overhead of basic system operations on L4Linux but also the virtualization overhead of a set of typical mobile phone applications above L4Linux. Both of these are usually concerned by mobile phone system developers. Furthermore, with the help of our evaluation results, potential points to be optimized as well as the methods to optimize the performance of applications above L4Linux can be identified.

The rest of this paper is organized as following. We first begin with related works in Section 2. Then in Section 3 we introduce some basics about the Fiasco L4 microkernel virtualization approach, i.e. L4Linux. In Section 4, we describe our evaluation methodology. In Section 5 and Section 6, the evaluation results are presented and analyzed respectively. After the evaluation results, in Section 7, a short discussion is given. Finally, we conclude in Section 8.

II. Related Works

The well known performance evaluations of L4 microkernel, are presented in [12][14]. In [14], several aspects of microkernel are addressed, such as kernel-user switches overhead, address space switches overhead, thread switches and interprocess communication (IPC) overhead, which are usually considered to be expensive on microkernel. It is proved that the significant overheads are usually due to the inefficient implementation, not inherited from the microkernel itself. The work in [12] especially focused on the virtualization overhead of L4Linux. To identify the virtualization overhead, the same set of benchmarks (i.e. *lmbench*, hbench and AIM) are executed on both native Linux and L4Linux on the same hardware platform then the obtained results are compared. The typical virtualization overhead of L4linux was shown to be around 5% to 10% [12]; by comparing the performance of L4Linux with another microkernel based virtualized Linux, MKLinux, it is concluded that the performance of the underlying microkernel indeed affect the virtualization overhead. An inefficient microkernel can make the system several times slower. All the works above are carried out on x86 platforms. None of them addressed the performance of L4 microkernel on embedded systems, for example, ARM processor based systems.

Concerning the performance of VMM for embedded systems, there have been some works [15][16]. The main target of [16] is to describe the optimization techniques used on Pistachio L4 microkernel, a re-implementation of L4 microkernel from University Karlsruhe. Some selected *lmbench* results are given to prove the effects of their optimizations. No real application performance was presented there. In [15], Xen is ported and evaluated as a VMM on ARM processor based system. *lmbench* is also used to investigate the performance of basic system operations. Simple performance comparison between Xen and Pistachio L4 microkernel is performed. A User Interface program is used as a real application benchmark to reflect the performance of Xen on common operations in mobile phones. The evaluation results show that Xen has moderate virtualization overheads on ARM processor.

III. L4Linux Basics

L4 microkernel only provides basic services needed to implement arbitrary systems (including virtual machines), such as address space management, thread management, and IPC. It runs in the most privilleged mode of the hardware so that it can control resources of the whole system. Additionally, it offers good isolation characteristic, which is necessary to run several subsystems concurrently on a single machine. Therefore, L4 microkernel has all the essentials that a VMM should have. By using L4 microkernel as a VMM, several virtual machines can run in parallel. An typical VM on L4 microkernel is the L4Linux.

L4Linux is a para-virtualized Linux on top of L4 microkernel where Linux is explicitly modified according to the virtual machine interfaces. The first port was done in 1996 at TU Dresden [17]. Since then it has been continuously updated to the newest Linux version. The supported architectures also have extended into embedded processors, e.g. ARM processors. Up to now, the latest version is L4Linux2.6.30.

A. Implementation

L4Linux is implemented with server-client approach, which is illustrated in Figure 1: an L4 task is used as a Linux-server that provides all the Linux services to the user processes/clients, which are also implemented as L4 tasks in different address spaces. User processes can only communicate with the Linux-server through IPC, which is one of the most important primitives that are supplied by the L4 kernel. As a result, the Linux-server and the untrusted user processes are isolated from each other. Moreover they are also isolated from the rest of the system. This is the reason why microkernel virtualization approach can enhance the security

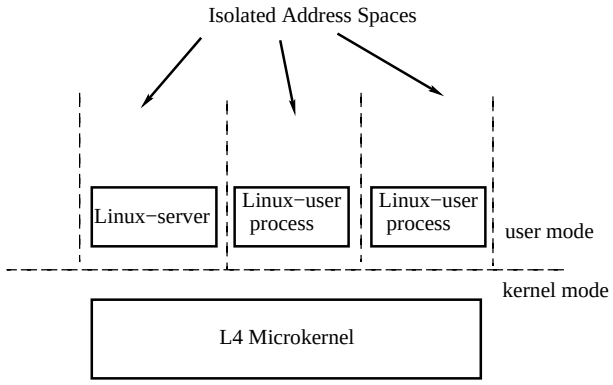

Fig. 1. L4Linux Implementation

aspect. Because the user processes/tasks from virtual machine are separated from the rest of the system, even though the guest OS on the virtual machine is compromised, the damage will not propagate to the rest of the system. Applied to mobile phones, this means even if a downloaded virus took control of the guest OS, telephone call still can be made, because the modem part of the mobile phone is protected from the attack.

B. System Calls

As the Linux-server and user processes are isolated from each other, system calls on L4Linux can not be directly handled by the kernel/Linux-server like in the native Linux. A mechanism called *syscall redirection* is applied to implement system calls on L4Linux. This mechanism is realized by using a user-level exception handler. Since L4 microkernel uses different system call numbers than native Linux, when an user process triggers a Linux system call, the L4 microkernel treats this as an exception, which is redirected to the Linux-server by the user-level exception handler (step 2 in Figure 2). Upon receiving this redirected system call, L4Linux processes this as a normal system call on native Linux and after that it will reply to the user process that triggered the system call by sending an IPC. The whole handling process is shown in Figure 2, from which we can see that each system call on L4Linux costs 2 kernel entry/exit pairs and 2 address space switches. This is much more expensive than in native Linux.

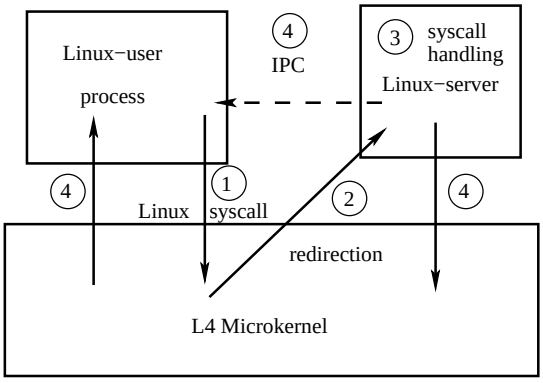

Fig. 2. System Call on L4Linux

C. Signaling

For security reasons, directly manipulating thread's stack, stack pointer, as well as instruction pointer from other threads in the same address space (like in native Linux) is not allowed in L4Linux. Instead, a user level signal-handler is added to every user process to solve this. Once the signal-handler receives a message from the Linux-server, it makes the main thread/Linux-server running in the same address space save its state and enter Linux by manipulating the main thread/Linux-server's stack pointer and instruction pointer.

D. Pagefault Handling

Pagefaults are handled by the Linux-server, which works as a pager for the user processes it creates. When a pagefault occurs, similar with the procedure of a system call, the L4 kernel treats the pagefault as an exception and redirects it to the Linux-server by sending an IPC. After the Linux-server receives the notification, it checks the shadow page tables that are maintained in user-level and assigns a new page from its own address space to the corresponding user process by mapping operations. The real page tables are kept within L4 and can not be directly accessed by user-level processes for security reasons. Maintaining the shadow page tables is quite expensive [18].

E. Interrupt Handling

The interrupt handling scheme on native Linux is emulated on L4Linux. In native Linux, the interrupt handler is implemented with two parts, top-halves interrupt handlers and bottom-havels interrupt handlers. In L4Linux, the top-halves interrupt handlers are replaced by separate dedicated threads that wait for interrupt messages sent by L4 kernel when hardware interrupt is triggered. Each thread corresponds to one hardware interrupt source. The bottom-halves are implemented by one single thread, which has higher priority than the Linux-server so that the interrupt handlers and Linux-server can execute sequentially.

IV. Evaluation Methodology

The evaluation in this paper differentiates itself from the previous evaluation works [12][15][16] in the following point: besides executing microbenchmarks (*lmbench*), a group of typical mobile phone applications are tested as application specific benchmarks to estimate the performance of L4 microkernel based VMM in real use cases. The reason for this approach is: microbenchmarks are usually used to detailedly analyze basic operations of operating system, e.g. system calls. From the discussion in Section III-B we can see that system calls on L4Linux are expensive due to the virtualization approach of L4Linux. Therefore, evaluating the weak points could help us find low-level bottlenecks and potential points that can be further optimized in the future. However, we should keep in mind that the results of *lmbench* do not indicate the overall performance, as it only measures the performance of basic system operations, which do not represent any real applications. To get the impression of the performance in real

use scenarios, we further evaluate with the application specific benchmarks. With this method, we can identify how much the real applications suffer from the low-level bottlenecks and the relations between the performance of low-level operations and high-level applications.

TABLE I
HARDWARE PLATFORM CONFIGURATIONS

Infineon $XMM6180$	
CPU	ARM1176JZS
Level 1 Cache	16K I-Cache, 16K D-Cache
Level 2 Cache	N.A.
CPU Frequency	364Mhz
Memory	64MB DDR
Memory Frequency	180Mhz
FPU	N.A.

From the evaluation we expect to answer the following two questions:

1) How large is the virtualization overhead on basic system operations?
2) Does this overhead matter in real use scenarios?

To evaluate the performance of L4 microkernel based VMM on modern mobile phone platform, we use $XMM^{TM}6180$ platform from Infineon Technologies as our evaluation hardware platform. The core component of $XMM6180$ is Infineon's $X\text{-}GOLD^{TM}618$ single chip basedband processor [19], which integrates wireless communication modem, mixed signal audio, measurement subsystem and power management unit on a single chip. It not only offers the modem functionality such as GSM, UMTS, GPRS, EDGE, HSxPA, but also provides many multimedia extensions, e.g. hardware accelerator for video recording and playback, integrated audio codes etc.. Table I shows the details of the platform configurations[1]. To generate comparable evaluation results, the same hardware platform and configurations are used across all the measurements in this paper.

To measure the virtualization overhead of L4 microkernel based VMM, we setup two benchmark environments: the first one is used to generate performance results of L4Linux, which is done by running all the benchmarks directly over L4Linux; the second one is executed as a comparison experiment where the same set of benchmarks are re-executed on native Linux. Any deviations from these two performance results are caused by the virtualization layer, i.e. virtualization overhead. The L4Linux used in this paper is L4Linux2.6.30. To make the results comparable, the same version of native Linux is chosen. This scheme is shown in Figure 3.

V. MICROBENCHMARKS

lmbench is developed by Larry McVoy and Carl Staelin from 1993 to 1995 [20]. Since then it has been widely used to measure the performance bottlenecks on many machines and operating systems. It is also commonly used for evaluating VMMs [12][15][16]. *lmbench* contains a suite of benchmarks that are designed to measure basic operations, such as system calls, context switches, memory access etc. All these benchmarks basically fall into two categories: latency benchmarks and bandwidth benchmarks. Depending on different benchmarking purposes, *lmbench* can be configured to measure performance of both operating system and hardware platform. As in this paper, the same hardware platform is used for all the measurements, *lmbench* is configured to execute the benchmarks related to operating systems only. Due to limited memory size of the board, all the benchmarks used in the experiment are compiled dynamically linked.

[1]The CPU frequency, memory frequency and cache size in Table I ONLY stand for the configurations we used in this paper.

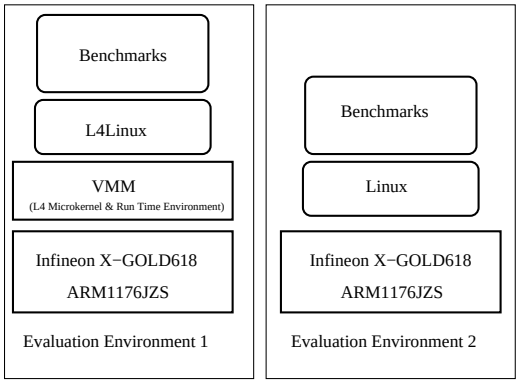

Fig. 3. Evaluation Environments

TABLE II
lmbench LATENCY RESULTS

Test Cases	Native Linux(us)	L4Linux (us)
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 hander 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

1) Latencies: Table II shows selected results of *lmbench* latency measurements. Results of both native Linux and L4Linux are listed in this table so that we can compare the differences. Figure 4 illustrates the corresponding slowdown/overhead of L4Linux, which is normalized to the performance of native Linux. As the figure indicates, executing these basic operations on L4Linux can be quite expensive compared with native Linux, especially the simple syscall benchmark, which is about 30 times slower than the one on native Linux.

The reason for this significant overhead is as follows: on native Linux, user tasks and kernel share the same address space. Each system call costs nothing but a CPU mode change; On L4Linux, as explained in Section III-B, Linux-server and

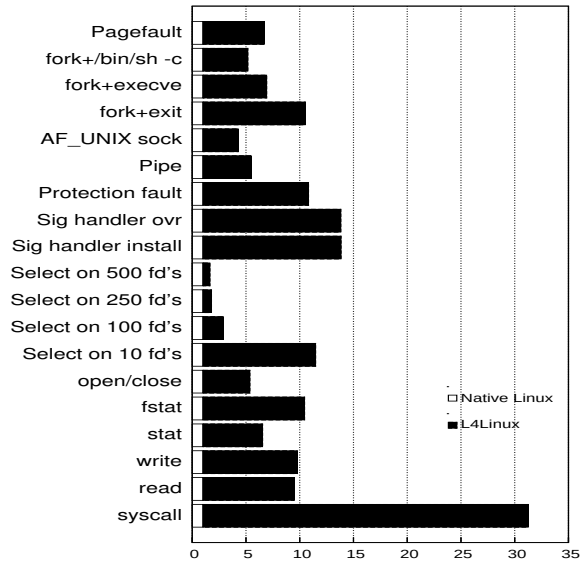

Fig. 4. *lmbench* Latency Results (normalized to native Linux performance)

user processes are isolated in different address spaces. Each system call costs 2 kernel entry/exit pairs plus 2 address space switches, which are time consuming procedures.

The overhead of kernel entry/exit and address space switches can directly attribute to three aspects [14]: pure kernel instructions execution, cache and TLB flushing. Since the processor (ARM1176JZS) uses physically indexed cache, cache flushing is not necessary during address space switches. Therefore, cache flushing is not a main contributor to this overhead. In current version of L4Linux, frequent TLB flushing is also eliminated by applying ASID (Address Space Identifier). As a consequence, the TLB can be retained across all the address spaces. Considering the above facts, we can conclude the significant overhead in system calls comes from executing all the additional kernel instructions when kernel entry/exits and address space switches are executed within system calls.

As additional signal-handler is used to avoid direct inter-thread manipulation in signal handling, the cost of signal handling is more expensive than the one on native Linux. In benchmarks fork, fork+execv and fork+sh in Table II, processes are created and executed. All these need to update page tables, which is realized by maintaining a set of shadow page tables. Manipulating shadow page tables and mapping guest virtual addresses to host physical addresses also add a lot of overhead to these operations.

2) Context Switch: Table III lists some selected results of context switch latencies both on native Linux and L4Linux. Without considering cache footprint (while the process size is 0k bytes), one context switch on L4Linux takes about 3 times longer than on native Linux. This overhead becomes lower when size of the processes increases. This is because when the size of process increases, the context switch overhead is dominated by cache interference.

Figure 5 plots the complete context switch results, from which we can see that the surface that is constructed by the context switch latencies of L4Linux is above the corresponding

TABLE III
SELECTED *lmbench* CONTEXT SWITCH RESULTS

Test Cases	Native Linux (us)	L4Linux(us)
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

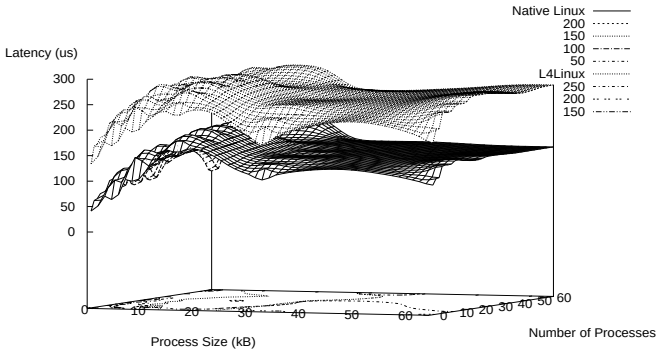

Fig. 5. *lmbench* Context Switch Results

surface of native Linux. This means context switches on L4Linux always take longer time than on native Linux.

3) Memory Access Bandwidth: The performance of memory access is presented in Figure 6 where we can hardly differentiate the curve of native Linux and its counterpart from L4Linux, which means the memory read bandwidth and memory write bandwidth of L4Linux are quite close to the native Linux. This is mainly due to the fact that in this benchmark simply an unrolled loop that sums up a series of integers is executed, which does not request any kernel services. Therefore, nearly no overhead is added to the system. This is the advantage of L4 virtualization approach. Because L4Linux does not try to emulate or intercept any instructions, the performance is nearly the same with the native one when the user process executes computing intensive tasks.

In the memory read bandwidth curve there is a sharp decrease when the block size is 16k bytes. This is because level 1 cache of ARM11 is 16k. The memory read bandwidth decreases tremendously when the accessing memory block size is larger than the cache size. As there is no level 2 cache available on our hardware platform, there is no second sharp decrease in this figure. The curve of memory write bandwidth is relatively flat compared with the memory read bandwidth curve. This is caused by the fact that on ARM1176JZS the cache is read allocate. The data accessed in the write bandwidth benchmark are not cached during execution. Therefore, the write bandwidth is not affected by the cache size. Thus

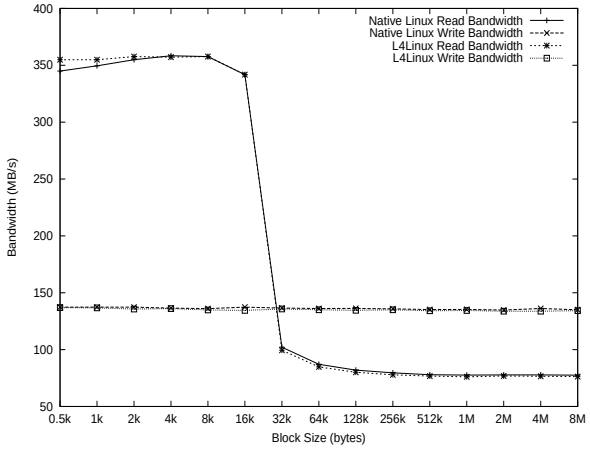

Fig. 6. Memory Read & Write Bandwidth

the write bandwidth does not change when the block size increases.

From the results of *lmbench* we can conclude that the performance of basic system operations on L4Linux is much lower than the one on native Linux. This is due to the virtualization overhead caused by the kernel entry/exits and address space switches during system call processing. Memory access on L4Linux performs just as efficiently as it does on native Linux. This is because no kernel services are need during execution. This also implies that by avoiding kernel services the virtualization overhead can be eliminated.

VI. Application Specific Benchmarks

lmbench only measures the performance of basic system operations. It does not represent any real application scenario. In order to evaluate the performance of L4Linux under more realistic use cases, we executed a series of application specific benchmarks, which will be described in details in this section.

Modern mobile phones are not only used as a simple communication tool but also as an entertainment device as well as personal digital assistant (PDA). Audio and video playback, digital image viewing & processing, web page browsing and word processing applications are commonly found nearly on all of the modern mobile phones. In order to generate more realistic benchmark results, which give us the impression of virtualization overhead under real use cases, we selected a series of application specific benchmarks from MiBench [21] and some other open source application to evaluate the performance of L4Linux. All the selected benchmarks can be divided into the following three groups:

1) Multimedia Applications: we choose lame, mad, mplayer, jpeg, tiff2bw, tiffdither, tiffmedian and typeset to represent the multimedia application benchmark set, which covers MP3 encoding, decoding, video decoding, digital image processing and HTML typesetting.
2) Office Automation: as modern mobile phone users tend to heavily rely on mobile phone as PDA to do word processing work, the office automation functionality becomes more and more important on modern mobile phones. The following typical applications are selected in this category to evaluate the performance in this aspect: ghostscript, stringsearch, ispell and rsynth.
3) Telecommunications: as the original basic functionality of a mobile phone, telecommunicate applications are chosen as the last category. It includes CRC32, FFT/IFFT, GSM encode/decode as well as adpcm encode/decode.

All the benchmarks are executed and timed on both native Linux and L4Linux. The benchmark results are listed in Table IV in terms of absolute execution time.

TABLE IV
Application Specific Benchmark Results

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

Figure 7 and Figure 8 illustrate the virtualization overhead in terms of slow-down. Unlike the situation in *lmbench*, as the figures show us, in most of the benchmarks the performance of L4Linux is very close to the native Linux, e.g. in case of lame and mplayer the overhead is only about 3%. This is much better than the performance of *lmbench*. An explanation for this is that these two applications are all CPU bounded applications where system calls are seldom triggered. Therefore, the huge system call overhead does not dominate anymore. What is more, L4 does not use the traditional “trap and emulate” virtualization approach. It does not try to emulate or intercept any instructions. Thus, most of the instructions can be directly executed on the CPU, which further reduces the virtualization overhead. This is the same explanation for the memory access bandwidth benchmark in *lmbench*.

At the same time, we also notice that some other applications, like jpeg and mad small, that also belong to the group of computing intensive applications, have much higher overhead compared with lame and mplayer. After further analysis of Table IV we can find that all the benchmarks with high overhead have short execution time compared with those with lower overhead, e.g. stringsearch 0.07s, adpcm 0.37s, mad small 0.232s, tiff2bw 0.64s, jpeg 0.84s. Based on this observation we assume that this inconsistent overhead is mainly produced during process creation and destruction. The absolute time for creating process and destroying process turns

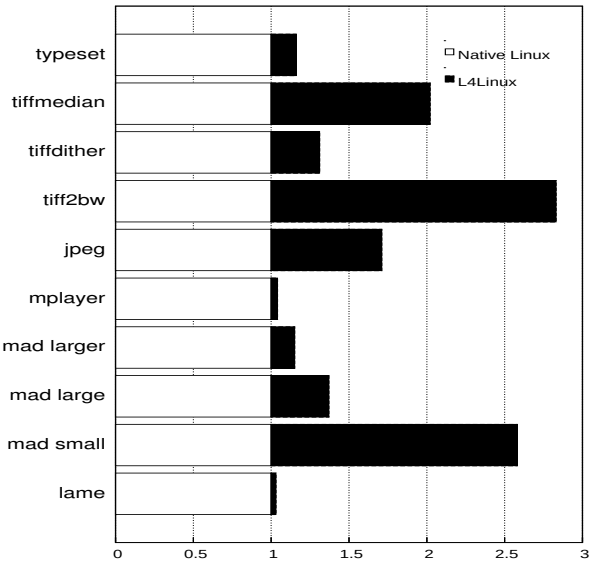

Fig. 7. Virtualization Overhead of Multimedia Applications

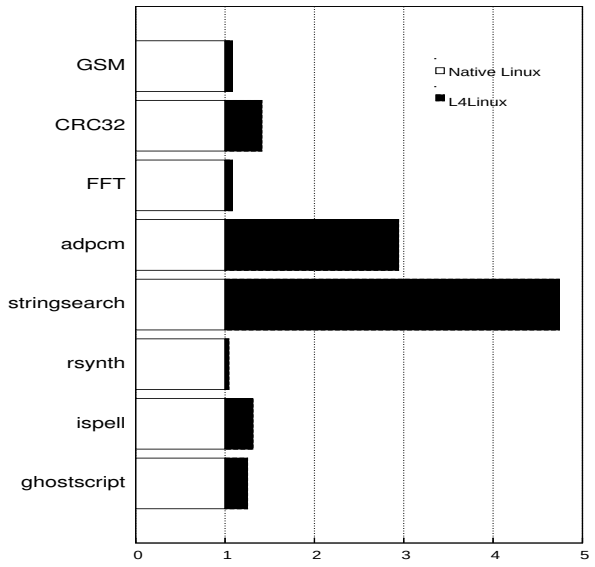

Fig. 8. Virtualization Overhead of Office Automation and Telecommunications Applications

to dominate when the execution time is short. This means if we use larger input file to extend the execution time of the benchmarks, the overhead will decrease accordingly. To prove this assumption, we re-measure the mad benchmark with two other MP3 files with different sizes. As expected, the overhead is reduced from 158% to 37% and 15% respectively. The overhead of stringsearch, adpcm, jpeg, tiff2bw, tiffmedian as well as tiffdither can be reduced in the same way. Considering the real use case that usually the input files are large enough to hide such kind of overhead, we can conclude that with typical mobile phone applications where the significant overhead of basic system operations do not dominate, L4Linux performs nearly as efficiently as native Linux.

VII. Discussion

The evaluation results from Section V and Section VI tell us that the virtualization overhead of L4 microkernel is use-case dependent. The overhead of system calls on L4Linux is significant while for computing intensive applications the overhead can become nearly neglectable. This is mostly caused by the virtualization mechanism applied by L4Linux. The more system calls are triggered, the more L4 microkernel services are involved, the more overhead is added to the system. This means applications with lots of system calls will suffer a lot from the virtualization overhead while the ones with few system calls can perform very efficiently. This rule gives us a guideline for developing and optimizing L4Linux based applications: frequent use of system calls should be avoided in performance critical applications; by avoiding triggering system calls frequently, the performance of applications can be improved.

VIII. Conclusions and Future Work

Based on our evaluation results, we can conclude that L4 microkernel based para-virtualization approach is feasible for modern mobile phones. Considering all the benefits the virtualization approach provides, such as a promising solution to the security challenge on modern mobile phones, accelerating development and cutting development cost, the inevitable virtualization overhead (around 5% for typical mobile phone applications) it adds to the system is acceptable and affordable.

At the same time, our evaluation results also indicate that the virtualization overhead of L4 microkernel based VMM is use case dependent. The overhead of system calls that trigger lots of kernel activities is significant while for those CPU bounded applications the overhead becomes nearly neglectable. This suggests us a guideline for developing and optimizing L4Linux based applications: frequent use of system calls should be avoided in performance critical applications.

The results of our evaluation also point out directions for the future works. First of all, kernel profiling work and detailed analysis need to be done in order to identify the hotspot within the significant system call overhead. After that proper optimizations need to be developed and evaluated. Secondly, rather than performance cost, further impacts of the virtualization on mobile phones need to be evaluated, with respect to power consumption, memory footprint and resources usage.

Acknowledgment

The authors would like to thank eMuCo, Embedded Multi-Core Processing for Mobile Communication Systems, and all the involved eMuCo partners for their support and help in this research.

eMuCo (www.emuco.eu) is a European project supported by the European Union under the Seventh Framework Programme (EP7) for research and technological development.

REFERENCES

[1] J. Brakensiek, A. Dröge, M. Botteck, H. Härtig, and A. Lackorzynski, "Virtualization as an enabler for security in mobile devices," in *IIES '08: Proceedings of the 1st workshop on Isolation and integration in embedded systems*, 2008.

[2] R. Goldberg, "Survey of virtual machine research," *IEEE Computer*, vol. 7, no. 6, pp. 34–45, 1974.

[3] P. Gum, "System/370 extended architecture: facilities for virtual machines," *IBM Journal of Research and Development*, vol. 27, no. 6, pp. 530–544, 1983.

[4] L. Seawright and R. MacKinnon, "Vm/370 - a study of multiplicity and usefulness," *IBM Systems Journal*, vol. 18, no. 1, pp. 4–17, 1979.

[5] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: Current technology and future trends," *Computer*, pp. 39–47, 2005.

[6] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006.

[7] C. Waldspurger, "Memory resource management in vmware esx server," *ACM SIGOPS Operating Systems Review*, vol. 36, pp. 181–194, 2002.

[8] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick, "Xen 3.0 and the art of virtualization," in *Linux Symposium*, 2005.

[9] "http://www.vmware.com/technology/mobile/."

[10] V. Inc, "Virtuallogix vlx," *Online at http://www. virtuallogix. com*.

[11] M. E. Gonzalez, A. Bilgic, A. Lackorzynski, D. Tudor, E. Matus, and I. Badr, "Ict - emuco: An innovative solution for future smart phones." in *IEEE ICME Workshop on Multimedia Signal Processing and Novel Parallel Computing.*, 2009.

[12] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg, "The performance of μ-kernel-based systems," in *Proceedings of the sixteenth ACM symposium on Operating systems principles*, 1997, pp. 66–77.

[13] H. Härtig and M. Roitzsch, "Ten years of research on l4-based real-time systems," in *Proceedings of the 8th Real-Time Linux Workshop*, 2006.

[14] J. Liedtke, "On micro-kernel construction," *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, pp. 237–250, 1995.

[15] J. Hwang, S. Suh, S. Heo, C. Park, J. Ryu, S. Park, and C. Kim, "Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones," in *5th IEEE Consumer Communications and Networking Conference, 2008. CCNC 2008*, 2008, pp. 257–261.

[16] C. van Schaik and G. Heiser, "High-performance microkernel and virtualization on arm segmented architectures data objects," *In Proceedings of the 1st International Workshop on Microkernels for Embedded Systems*, 2007.

[17] M. Hohmuth, "Linux-emulation auf einem mikrokern," Master's thesis, TU-Dresden, 1996.

[18] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," in *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008.

[19] *X-GOLD 61X Production Specification*, Version3.0 ed.

[20] L. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*. Usenix Association, 1996, p. 23.

[21] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *IEEE 4th annual Workshop on Workload Characterization*, vol. 131, 2001, pp. 184–193.