The Role of Virtualization in Embedded Systems

Gernot Heiser

Open Kernel Labs and NICTA[*] and University of New South Wales
Sydney, Australia

ABSTRACT

System virtualization, which enjoys immense popularity in the enterprise and personal computing spaces, is recently gaining significant interest in the embedded domain. Starting from a comparison of key characteristics of enterprise systems and embedded systems, we will examine the difference in motivation for the use of system virtual machines, and the resulting differences in the requirements for the technology. We find that these differences are quite substantial, and that virtualization is unable to meet the special requirements of embedded systems. Instead, more general operating-systems technologies are required, which support virtualization as a special case. We argue that high-performance microkernels, specifically L4, are a technology that provides a good match for the requirements of next-generation embedded systems.

1. INTRODUCTION

System virtualization has become a mainstream tool in the computing industry, as indicated by billion-dollar IPOs and sales of startup companies for hundreds of millions. The decoupling of virtual and physical computing platforms via system virtual machines (VMs) supports a variety of uses, of which the most popular ones are:

- consolidating services that were using individual computers into individual virtual machines on the same computer. This utilises the strong resource isolation provided by virtual machines in order to achieve quality-of-service (QoS) isolation between servers;
- load-balancing across clusters, by creating new virtual machines on demand on a lightly-used host, or even migrating live VMs. This utilises the platform abstraction provided by virtualization;
- power management in clusters, by moving VMs off lightly-loaded machines, which can then be shut down (this is effectively load-balancing in reverse);
- firewalling services which have a high risk of being compromised in order to protect the rest of the system. This also utilises resource isolation;
- running different operating systems (OSes) on the same physical machine (e.g. Windows, Linux and MacOS), typically in order to run applications, which are specific to a particular OS. This use is mostly relevant for personal machines (desktops or laptops) and is also enabled by resource isolation.

A main characteristic of such usage cases is that typically all VMs run the same OS (or, in the last scenario listed above, "similar" OSes in the sense that they provide roughly the same kinds of capabilities and similar abstraction levels). Also characteristic of those scenarios is that VMs communicate just like physical machines—via (virtual) network interfaces (including network file systems). This is consistent with the VM view, which is, by definition, like that of a physical machine.

Clearly, most of the above use cases have no equivalent in present-day embedded systems (although some will become relevant with the advent of manycore chips). In order to understand why system virtual machines are recently receiving a lot of interest from embedded-systems developers, we need to have a look at the characteristics of modern embedded systems, and identify commonalities as well as differences to enterprise computing systems.

2. EMBEDDED SYSTEMS PROPERTIES

Embedded systems used to be relatively simple, single-purpose devices. They were dominated by hardware constraints (memory, processing power, battery charge). Their functionality was also mostly determined by hardware, with software consisting largely of device drivers, scheduler and a bit of control logic. As a result, they exhibited low to moderate software complexity. They were subject to real-time constraints, which poses demands on operating systems that are unusual in the general-purpose computing arena.

Traditional embedded systems are also closed: the complete software stack is provided by the device vendor, loaded pre-sale, and does not change (except for rare firmware upgrades).

Modern embedded systems, however, are increasingly taking on characteristics of general-purpose systems. Their functionality is growing, and so is the amount and complexity of their software. The software stack running on contemporary smartphones is already 5–7Mloc, and growing. Top-of-the-line cars contain literally gigabytes of software (and rumour has it that it takes longer to load the software than to build the physical vehicle). Increasingly, embedded systems run applications originally developed for the PC world (such as the Safari web browser running on the iPhone) and new applications (e.g. games) are increasingly written by program-

[*] NICTA is funded by the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

First Workshop on Isolation and Integration in Embedded Systems (IIES'08) April 1, 2008, Glasgow, UK
Copyright © 2008 ACM 978-1-60558-126-2 ...$5.00.

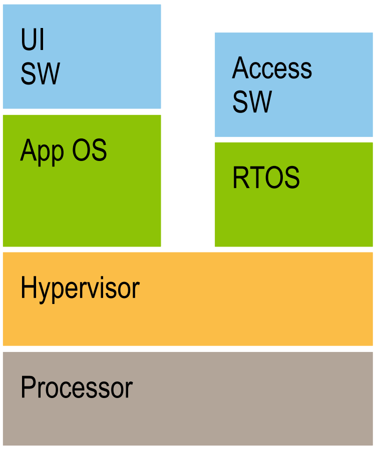

Figure 1: The primary use case for virtualization in embedded systems is the co-existence of two completely different OS environments, real-time OS and high-level application OS, on the same processor.

mers without embedded-systems expertise. This creates a demand for high-level application-oriented operating systems with commodity APIs (Linux, Windows, Mac OS).

Furthermore, there is a strong trend towards openness [4, 17]. Device owners want to load their own applications on the systems and run them there. This requires open APIs (and introduces all the security challenges known from the PC world, including viruses and worms).

Yet some of the old differences to general-purpose systems remain. Embedded devices are still real-time systems (or at least part of the software is real-time). They are also frequently still resource constrained: battery capacity increases only slowly over time, hence mobile devices have tight energy budgets. Also, as many embedded systems are sold for just a few dollars, memory is frequently still a cost factor (besides being a consumer of energy).

At the same time, embedded systems, already ubiquitous, are becoming more and more part of everyday life, to the degree that it is becoming hard to imagine living without them. They are increasingly used in mission- and life-critical scenarios. Correspondingly, there are high and increasing requirements on safety, reliability and security.

3. VIRTUALIZATION USE CASES

The relevance of virtualization in embedded systems stems from the ability to address some of the new challenges posed by them.

One is support for **heterogeneous operating-system environments**, as a way to address the conflicting requirements of high-level APIs for application programming, real-time performance and legacy support. Mainstream application OSes lack the support for true real-time responsiveness (efforts to address this notwithstanding) and they are unsuitable for supporting the large amount of legacy firmware running on present devices (e.g. mobile-phone baseband stacks alone can measure several Mloc).

Virtualization can help here, by enabling the concurrent execution of an application OS (Linux, Windows, Symbian, ...) and a real-time OS (RTOS) on the same processor (see Figure 1). Provided that the underlying hypervisor is able to deliver interrupts reliably fast to the RTOS, the latter can then continue to run the (legacy) stack providing the device's real-time functionality. The application OS can provide the required commodity API and high-level functionality suitable for application programming.

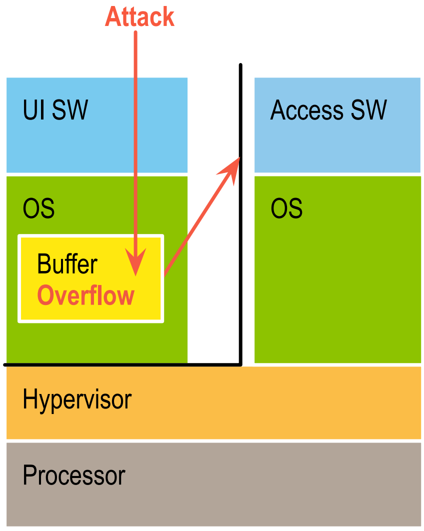

Figure 2: Standard security use case: A (user- or network-facing) OS is compromised, but encapsulated in a VM, which protects the rest of the system from the exploit.

Note that the same can be achieved by using multiple processor cores, each running their own OS, provided there is some hardware support for securely partitioning memory. Multicore chips are proliferating and the incremental cost of a core is dropping dramatically. Furthermore, two lower-performance cores (which can be put to sleep individually) are likely to have lower average power consumption than a single higher-performance one, owing to strong non-linearities in power management [22, 23]. Hence, on its own, this usage scenario is likely only of relevance for a few years. We note, however, that virtualization supports **architectural abstraction**, as the same software architecture can be migrated essentially unchanged between a multicore and a (virtualized) single core.

Forthcoming **manycore chips** provide a different, more longer-term motivation for using virtualization. With a large number of processors, it is likely that embedded systems will be facing issues not unlike the reasons behind today's use of virtualization in the enterprise space. A scalable hypervisor could form the basis for deploying poorly-scaling legacy operating systems on a large number of cores, by partitioning the chip into several smaller multiprocessor domains. A hypervisor can dynamically add cores to an application domain which requires extra processor power, or can manage power consumption by removing processors from domains and shutting down idle cores. (Note that this requires that the application OS can deal with a changing number of CPUs.) The hypervisor can also be used for setting up redundant domains for fault tolerance in a hot-failover configuration.

Probably the strongest motivation for virtualization is **security**. With the trend towards open systems, the likelihood that an application OS is compromised increases dramatically. The resulting damage can be minimised by running such an OS in a virtual machine which limits access to the rest of the system, as shown in Figure 2. Specifically, downloaded code may be restricted to run in a VM environment, or services, which are accessible remotely, could be encapsulated in a VM. This security use case is only valid if:

- the underlying hypervisor is significantly more secure than the guest OS (which means first off that the hypervisor must be **much** smaller), and

- critical functionality can be segregated into VMs different from the exposed user- or network-facing ones.

If those prerequisites are not met, the hypervisor will simply *increase* the size of the *trusted computing base* (TCB), which is counter-productive for security.

Finally, wide (and standardised) support for virtualization enables a new model for **distribution of application software**—shipping the program together with its own OS image. This provides the application developer with a well-defined OS environment, and thus reduces the likelihood of failure of the deployed software due to configuration mismatch. While such a scheme has significant resource implications (especially on memory use), this can be reduced by automatic elimination of replicated page contents [15, 26].

4. LIMITS OF VIRTUALIZATION

The above use cases show that virtualization can provide some attractive benefits to embedded systems. However, there are significant limitations on the use of system VMs in embedded systems—in fact, these limitations are a direct consequence of what makes virtualization popular.

Virtualization is all about isolation—by definition, a virtual machine runs on its virtual hardware as if it had exclusive use of physical hardware. (Note that in an attempt to derive marketing value from a popular buzzword, some vendors are not above advertising (highly insecure) *OS co-location* as virtualization. But this kind of pseudo-virtualization solves nothing but the simplest cases of the heterogeneous-OS use case, and will not be considered any further.)

In contrast to the server space, the model of strongly-isolated virtual machines does not fit the requirements of embedded systems. By their very nature, embedded systems are highly integrated, all their subsystems need to cooperate in order to contribute to the overall function of the system. Isolating them from each other interferes with the functional requirements of the system.

Cooperation between the various subsystems of an embedded system requires **efficient sharing**. This is needed for bulk data transfer—a mobile phone may receive a video file via the cellular network (i.e. via the baseband OS), which is then displayed on the screen (by a media player running on the application OS). Transferring this data “virtual-machine style” via a virtual network interface between the real-time and the application VM implies at least one extra copy operation, a significant waste of processor cycles (and hence battery charge). Clearly, a shared buffer together with low-latency synchronisation/messaging primitives is the way to go. However, such operations do not fit the virtual-machine model.

Another mismatch between embedded-systems requirements and the virtual-machine model is evident in **scheduling**. Virtual machines, by their nature, are scheduled by the hypervisor as black boxes, with the guest OS responsible for scheduling activities within the VM. However, this is not suitable for embedded systems; their integrated nature requires an integrated (global) approach to scheduling, as shown in Figure 3. While real-time activities (running in an RTOS) generally must have the highest scheduling priority, the RTOS domain will also have low-priority background activities. These should not be able to preempt the user interface running under the application OS. Similarly, the application OS may be running some (soft) real-time activities, e.g. the media player, that may take precedence over some other real-time activates in the RTOS environment. The characteristics of embedded systems obviously cannot be addressed with the de-centralised, hierarchical scheduling model that is inherent in virtualization.

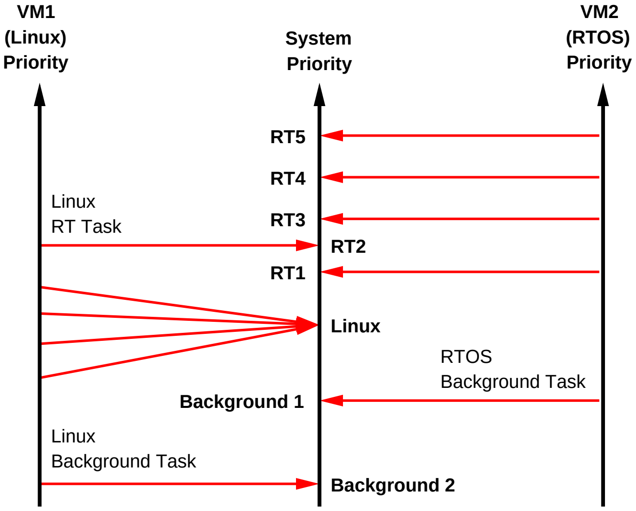

Figure 3: The integrated nature of embedded systems requires that scheduling priorities of different subsystems must be interleaved. This is at odds with the concept of virtual machines.

Energy management is similar to real-time scheduling, in that it is a whole-of-system issue that cannot be done locally. Energy management in embedded systems frequently involves trading off performance (i.e. time) against energy, under the constraint of meeting deadlines. However, energy is a global *physical* resource that cannot be traded off against *virtual* time. This is in contrast to the server world, where the goal is to minimise power consumption (or temperature) rather than energy, and a hierarchical approach can work [24].

Furthermore, virtualization does little to address the possibly biggest issue facing embedded systems: the mushrooming **software complexity**, which threatens to undermine device robustness and safety. The accepted software-engineering approach for addressing complexity challenges is to use encapsulated components [25] for fault containment. While virtualization provides encapsulation, its granularity is too coarse to make a big difference. After all, a virtual machine emulates hardware and is designed to run an operating system supporting its software. This means that virtual machines are fairly heavyweight, and embedded systems cannot reasonably run more than a few of them (remember, memory costs energy and increases the bill of materials).

Addressing the software-complexity challenge requires a component framework that is lightweight in terms of memory overhead as well as with respect to inter-component communication cost, and does not add to the overall complexity of the system.

Last but not least, the issue of **information-flow control** is becoming increasingly important. Surprising as it may sound at first, embedded systems are no-longer single-user devices. A mobile phone handset, for example, has at least three *classes* of stake holders, which are essentially “users” in that they have different access rights:

1. The *owner*, i.e. the human who has, in some way or other, bought the device.

2. The *wireless service provider*, who grants access to a wireless network. In the future we might see several concurrent service providers for the same physical device, along

the lines of the DoCoMo/Intel OSTI proposal [17], which argues for a separation of enterprise and private service when the same physical device is used for private as well as business use.

3. *Third-party service providers* who use the wireless connectivity to provide services independent of the wireless service provider. This includes providers of multimedia content for entertainment or information. Increasingly it includes financial transactions such as payments for arbitrary goods and services.

These users are, in general, mutually distrusting, and each has assets on the device they wish to protect. The owner has address books, emails and documents in which the other users have no legitimate interest, and which the owner wishes to keep private, and thus out of reach of the other users. In the OSTI scenario, these would be further separated into private and enterprise data, leading to multiple logical owner-type users (or "roles").

The service provider needs to ensure the integrity of the network, and needs to ensure that all network access is properly authenticated (to ensure correct billing as well as for complying with legal requirements) and does not interfere with others' legitimate use. It therefore needs confidence that the device will adhere to protocols.

The content providers need assurance that their data is only used according to the owner's license (e.g. only displayed for a limited number of times and not copied to other devices). The owner as well as the providers need assurance that financial transactions with other third-party service providers are secure, including strong protection of access tokens.

We therefore have an access-control problem not dissimilar from what is found in traditional (time-shared) multi-user systems. There are several agents with legitimate rights to access certain data, but no rights to other data. Information must flow between the agents, but a security monitor must be able to enforce access policies. Importantly, many of the subsystems will handle sensible data belonging to any of the users. If these subsystems are inside a virtual machine, then the guest OS must be trusted to enforce the information-flow policies.

From the security point of view, this would obviously be worse than just trusting a single operating system, it would be a (potentially massive) increase in the size of the system's trusted computing base. It would negate many of the potential security benefits that motivate the use of virtualization as discussed above. For security reasons, the TCB should be minimised, and not depend on large application OSes (that have a high risk of compromise).

5. SUITABLE TECHNOLOGY

Having looked at the challenges posed by modern embedded systems, we can now attempt to translate those into requirements for a suitable OS technology:

- The ideal technology will provide strongly encapsulated components, suitable for fault containment and security isolation. Yet it needs to support more traditional system virtualization, specifically the ability to run isolated guest OS instances and their (unmodified) application stacks. This must be done while maintaining real-time responsiveness for time-critical subsystems.

- At the same time, *controlled* low-latency, high-bandwidth communication must be available between components, including shared memory, subject to a system-wide security policy, enforced by a small and *trustworthy* TCB. Components must be lightweight enough to make them suitable for encapsulating individual threads, in order to impose a global scheduling policy.

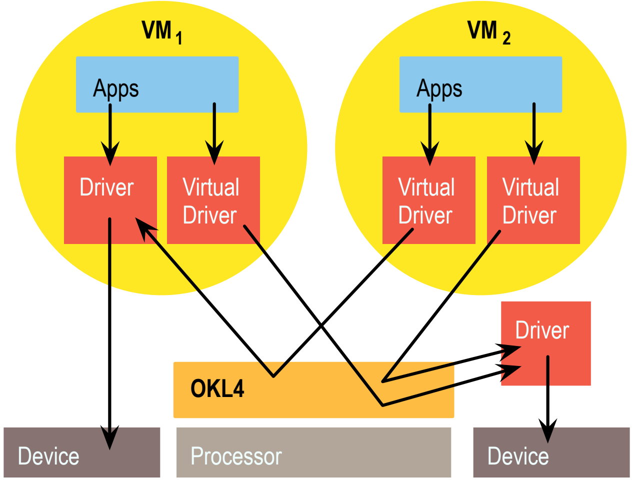

Figure 4: Microkernels run device drivers at user level, ideally each confined to its own address space. They can be accessed from virtual machines using stub drivers, which simply convert I/O requests into IPC messages to the real driver. The same approach can be used for legacy drivers hosted inside their native OS, which eases migration to the microkernel.

This shopping list exceeds the capabilities of a hypervisor, and requires more general-purpose OS mechanisms. High-performance microkernels seem to provide the right capabilities: They have a proven track record as the basis for hypervisors [6, 14] as well as real-time OSes [5, 7, 18]. Specifically the various members of the L4 microkernel family [16] feature extremely low-overhead message passing, light-weight address spaces that form the basis of encapsulation, mechanisms for setting up shared-memory regions, and high-performance user-level device drivers [12, 13].

The efficient message-passing (IPC) mechanism is the key enabler for any microkernel-based system, and this includes virtual machines. Virtualization traps are caught by the kernel and converted into exception messages sent to a user-level virtual-machine monitor. It is also, together with lightweight address spaces, a key enabler of encapsulated lightweight components [11]. Finally it makes user-level device drivers feasible, as interrupts are also converted to IPC messages sent to the driver.

Shared-memory regions are the basis of efficient sharing of buffers (for zero-copy operations). This is also important for sharing devices between virtual machines and other components. Figure 4 shows that driver sharing can also be applied for drivers left inside their original host OS (now running in a virtual machine). This is important for legacy support: A driver can (initially) be left inside its original OS, yet be made available to other components in the system. This, of course, means that the guest OS hosting the driver must be trusted to operate the particular device correctly. As an interim measure this may be acceptable—it constitutes the first step in a migration to a more structured system, where drivers are encapsulated in their own address spaces. Of course, for DMA-capable devices this encapsulation is only complete if the hardware provides an IOMMU [13], such as provided by recent Intel and AMD processors.

Legacy support and a soft upgrade path is also provided by the combination of virtualization with lightweight components. A soft-

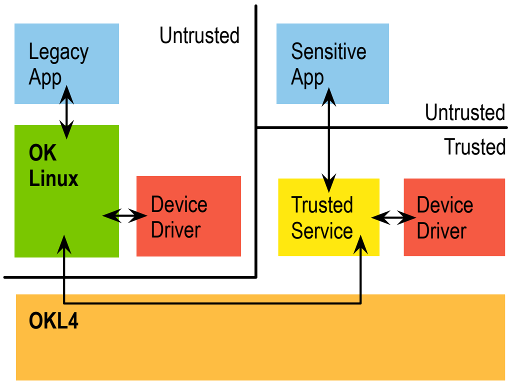

Figure 5: Virtualization allows porting a complete software stack to the microkernel with low engineering effort. Critical components can be extracted to run native on the microkernel with a minimal trusted computing base.

ware stack can easily be ported to a microkernel by encapsulating the complete stack inside a VM. Critical components, such as encryption and key management, can be moved into separate components running directly on top of the microkernel with a minimal TCB [9, 21], as shown in Figure 5. Such components can be active (i.e. contain their own schedulable thread of execution) or passive (invoked similar to a function-call and executing in the caller's scheduling context). The ability to assign an individual scheduling priority to an active component overcomes the limitations of VM scheduling indicated in Figure 3.

Over time, more parts can be extracted out of a VM into separate components, leading to a highly structured system (see Figure 6) with increased robustness. The system designer is able to decide on the most appropriate tradeoff between security, robustness and engineering cost.

This approach is already being actively applied in the embedded-systems industry. The OKL4 microkernel platform from Open Kernel Labs is presently deployed on an estimated 100 million mobile wireless devices. In most cases, the initial deployment looks very much like the one shown in Figure 5, and a number of users are now componentising their software stacks further, moving towards the hybrid model of Figure 6.

Work is ongoing on creating a TCB that is not only small (less than 20kloc, of which only about 10kloc are kernel code), but truly *trustworthy*. NICTA's new high-security version of L4, called seL4, has been proved to satisfy strict isolation and information-flow control requirements [2] without sacrificing the high performance L4 has long been known for [1]. The API of the OKL4 microkernel is presently being evolved into an equivalent of seL4. At the same time, work at NICTA aims at a formal proof that seL4 fully satisfies the isolation requirements of the Common Criteria Separation Kernel Protection Profile [10].

However, the most exciting aspect of the small TCB enabled by microkernel technology is the potential to *establish its trustworthiness* beyond any doubt. A well-designed microkernel is small enough to be completely formally verified—by constructing a mathematical proof that the implementation adheres to the specification (and thus is in a sense guaranteed to be "bug-free").

Such a complete correctness proof of the seL4 microkernel [3] is only a few months away [8], and constitutes the first (and hardest) step towards verifying the complete TCB of some security-critical components.

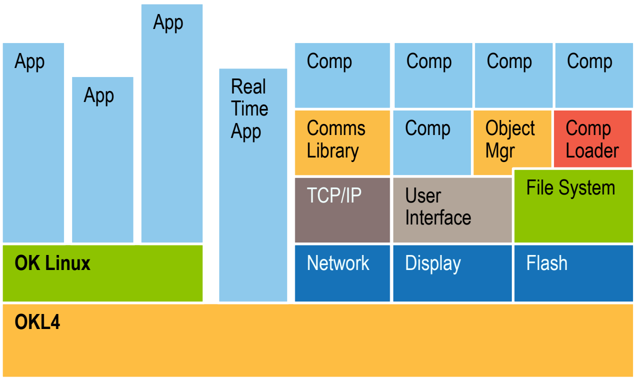

Figure 6: A hybrid system structure supports a monolithic legacy system side-by-side with a highly structured, fault-resilient system based on encapsulated components.

This represents a quantum leap from established OS technology. A formally-verified kernel can be treated very much like hardware: it only changes every few years (and never post-deployment). But it allows the software on top to be upgraded. It also supports new models of secure introspection in systems.

Take a virus scanner, for example. It normally runs inside the operating system, which it is examining for infection. This obviously creates the risk that the virus scanner itself is being infected, or the OS mechanisms it uses for the inspection are compromised in a way that defies detection by the scanner using them. Installing the virus scanner inside a virtual-machine monitor means changing the hypervisor and thus risks infecting the hypervisor itself.

With the hardware-like trusted microkernel, a virus scanner can run on top of the microkernel, subject to its protection mechanisms, but outside any VM. It can be enabled to scan the VM by mapping all of the VM's memory to the virus scanner read-only, so the scanner can itself not damage anything (in observance of the principle of least privilege [20]). It can also be encapsulated so it cannot leak any of the data it sees inside the VM, while at the same time being protected from any user- or network-facing system components (which are at high risk of being compromised).

This use case is an indication of the potential on-going relevance of virtual machines in embedded systems. But it is also a further indication that virtual machines alone are not sufficient.

6. CONCLUSIONS

Virtualization has many aspects attractive to the embedded world, but on its own is a poor match for modern embedded systems. More general OS technology is required, that supports fine-grained encapsulation, integrated scheduling and information-flow control. High-performance microkernels are able to provide all the required features, and enable a migration from monolithic to componentised (fault-resilient) software stacks. The prospect of formal verification of the microkernel's implementation provides an exciting avenue towards systems of unprecedented reliability.

There seems to be a trend in hypervisors to become more microkernel-like [19], including adding microkernel-like primitives to overcome some of the shortcomings of virtualization. Microkernels, however, have the inherent advantage of enabling a smaller TCB [9]. Moreover, the benefits of formal verification can, for the foreseeable future, only be achieved with a real microkernel, due to the poor scalability of formal verification techniques.

We conclude that there are good reasons for deploying some forms of virtualization in future embedded systems. However, the role of system VMs will remain limited and their benefits will only be fully achieved in the context of an overall change of operating-system technology to one based on high-performance microkernels.

7. REFERENCES

[1] D. Elkaduwe, P. Derrin, and K. Elphinstone. Kernel design for isolation and assurance of physical memory. In *1st Workshop on Isolation and Integration in Embedded Systems*, Glasgow, UK, Apr. 2008. ACM SIGOPS.

[2] D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the seL4 microkernel. *Submitted for publication*, Oct. 2007.

[3] K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a practical, verified kernel. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, San Diego, CA, USA, May 2007.

[4] Google. What is Android? http://code.google.com/android/what-is-android.html, Nov. 2007.

[5] Green Hills Software. INTEGRITY real-time operating system. http://www.ghs.com/products/rtos/integrity.html.

[6] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 66–77, St. Malo, France, Oct. 1997.

[7] H. Härtig and M. Roitzsch. Ten years of research on L4-based real-time systems. In *Proceedings of the 8th Real-Time Linux Workshop*, Lanzhou, China, 2006.

[8] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S. M. Petters. Towards trustworthy computing systems: Taking microkernels to the next level. *ACM Operating Systems Review*, 41(3), July 2007.

[9] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro. Reducing TCB size by using untrusted components — small kernels versus virtual-machine monitors. In *Proceedings of the 11th SIGOPS European Workshop*, Leuven, Belgium, Sept. 2004.

[10] Information Assurance Directorate. *U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness*, June 2007. Version 1.03. http://www.niap-ccevs.org/cc-scheme/pp/pp.cfm/id/pp_skpp_hr_v1.03/.

[11] 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):687–699, May 2007.

[12] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. R. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, Sept. 2005.

[13] B. Leslie, N. FitzRoy-Dale, and G. Heiser. Encapsulated user-level device drivers in the Mungi operating system. In *Proceedings of the Workshop on Object Systems and Software Architectures 2004*, Victor Harbor, South Australia, Australia, Jan. 2004. http://www.cs.adelaide.edu.au/~wossa2004/HTML/.

[14] B. Leslie, C. van Schaik, and G. Heiser. Wombat: A portable user-mode Linux for embedded systems. In *Proceedings of the 6th Linux.Conf.Au*, Canberra, Apr. 2005.

[15] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, pages 17–30, San Francisco, CA, USA, Dec. 2004.

[16] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, USA, Dec. 1995.

[17] NTT DoCoMo and Intel Corp. Open and secure terminal initiative (OSTI) architecture specification. http://www.nttdocomo.co.jp/english/corporate/technology/osti/, Oct. 2006.

[18] QNX Software Systems. Neutrino realtime operating system. http://www.qnx.com/products/neutrino_rtos/.

[19] M. Rosenblum. The impact of virtualization on computing systems. Keynote address at USENIX Technical Conference, June 2007.

[20] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63:1278–1308, 1975.

[21] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Proceedings of the EuroSys Conference*, pages 161–174, Leuven, Belgium, Apr. 2006.

[22] D. C. Snowdon, S. M. Petters, and G. Heiser. Accurate On-line Prediction of Processor and Memory Energy Usage Under Voltage Scaling. In *Proceedings of the 7th International Conference on Embedded Software*, Salzburg, Austria, Oct. 2007.

[23] D. C. Snowdon, S. Ruocco, and G. Heiser. Power Management and Dynamic Voltage Scaling: Myths and Facts. In *Proceedings of the 2005 Workshop on Power Aware Real-time Computing*, New Jersey, USA, Sept. 2005.

[24] J. Stoess, C. Lang, and F. Bellosa. Energy management for hypervisor-based virtual machines. In *Proceedings of the 2007 Annual USENIX Technical Conference*, Santa Clara, CA, USA, June 2007.

[25] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley/ACM Press, Essex, England, 1997.

[26] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, USA, 2002.