Wombat
A Portable User-Mode Linux for Embedded Systems

Ben Leslie[†], Carl van Schaik[†] and Gernot Heiser[†‡]

[†] National ICT Australia, Sydney, Australia
[‡] University of New South Wales, Sydney, Australia
⟨firstname.lastname⟩@nicta.com.au

Abstract

Embedded systems are the biggest potential market for Linux, much bigger (in terms of number as well as total value) than either the desktop or the server market. While Linux is making excellent inroads into (high-end) embedded systems, a number of challenges particular to embedded systems threaten to limit its impact. These include the requirements for hard real-time capability, extreme robustness, and, in particular, a minimal trusted computing base. The viral nature of the GPL is also frequently causing problems.

We argue that a portable user-mode Linux which runs on a truly minimal kernel is the answer, and will open up application domains which would otherwise be hard to penetrate. We present such a system, called Wombat, which is a port of Linux kernel to the L4 microkernel. Wombat is readily portable between architectures (presently runs on x86, ARM and MIPS), and initial performance evaluations look promising.

1 Embedded Systems: The Next Frontier

Embedded systems are characterised as devices which are not primarily computers but contain one or more processors, operating "behind the scenes", in order to provide part of the device's functionality. The embedded market is huge — well over 99% of all processors are embedded — and growing strongly (while the PC and server markets are comparatively flat).

Presently, the vast majority of embedded systems are based on rather primitive processors, 8- or 16-bit micro-controllers without memory protection, performing relatively simple control operations. Such devices tend to have little or no operating system (OS), their software comprises essentially a control loop which executes task according to a fixed schedule, and some minimal "kernel',' consisting of some device drivers and simple libraries.

However, there is a strong trend from those "classical" embedded systems towards more powerful platforms, 32-bit (sometimes even 64-bit) general-purpose processors which provide memory protection via a *memory-management unit* (MMU). The reasons for this development include the market demand for more sophisticated embedded systems with a lot of complex functionality. A typical example are personal communication and entertainment devices, where there is an increasing convergence of what used to be dissimilar devices, into a single system offering a wide variety of functions.

Such devices have requirements that are quite different from those of classical (closed) embedded systems: high demand on processing capability, and a much more open architecture, which features internet connectivity, field upgradability via remote access, standardised and well-known application programming interface (API) and the ability to process downloaded data and even execute downloaded code. These are requirements that are well supported by contemporary desktop and server operating systems, and it is therefore not surprising that there is a strong trend towards an increasing use of standard operating systems, such as Linux, in embedded systems. In fact, surveys show that Linux is the leading OS for *new* embedded systems work, with (various versions of) Windows taking second place [Lin04].

While we suspect that such surveys are strongly biased towards 32-bit systems and therefore somewhat misleading, there is little doubt that the tendency towards "standard" OSes in embedded systems is real, and presents the strongest growth potential for Linux.

The main reasons that are typically given for the popularity of Linux in embedded systems are source-code availability and the royalty-free status. Surveys show that many embedded systems developers are willing to pay for development environments, training and other support, but are unwilling to share the income from their products (in the form of per-unit royalties) [EDC03].

2 Embedded Systems Challenges

While modern embedded systems have requirements that are well supported by Linux, they provide a number of other challenges, which make Linux a less-than-ideal choice. These include:

hard real-time: Embedded systems are mostly real-time systems, meaning that they have to respond to external events in a timely fashion. In many cases (eg. multi-media systems) this real-time requirement is "soft", meaning that such systems can tolerate missing a deadline occasionally. Other systems have "hard" real-time requirements: missing a deadline is considered a complete system failure, and may result in mission failure or even death.

In spite of very significant progress in Linux's real-time responsiveness, normal Linux is not suitable for hard real-time systems, and there are signs that the situation has recently worsened [SM04]. Special real-time versions, such as RTLinux [FSM] and RTAI [RTA] address the problem by adding a real-time layer below the kernel proper, in order to have full control over interrupt handling. This leads to an architecture which is, in principle, capable of meeting real-time requirements, although at a cost of running the real-time components in the kernel (with corresponding loss of protection).

However, the resulting system is still too complex to be fully analysed with respect to its real-time performance, with a resulting uncertainty about its ability to really meet the real-time goals. In fact, it has been shown that a heavily-loaded RTLinux system fails to honour its real-time guarantees [MHH02]. Ideally, the system's real-time performance should be established either by mathematical proof, or by a complete empirical execution-time analysis of all its possible execution paths. This is only practical if the kernel and other real-time components are *very* small.

highly robust: Embedded systems are often employed in life-critical or mission-critical scenarios. While the reliability of Linux on desktops and servers is very high, this typically applies to systems which are at least close to widely-deployed configurations. Massive changes to system configuration, as it is typically necessary for an embedded system, will inherently reduce stability and require a significant maturation process. In the meantime, the critical parts of the system should not be affected by other components which may only be required for supporting a user interface or some non-critical entertainment function. Furthermore, the critical part of the system must be protected from attacks by malicious programs which the user downloaded from the internet.

A related issue is that of upgrading the system without downtime. While it is possible, in theory, to upgrade Linux kernel modules without rebooting the whole system, in practice this is very limited, as many modules are tied closely to a specific kernel version, making it impossible to load a newer version of the model into an old kernel. Other components of the kernel are impossible to upgrade without a reboot;

small trusted computing base: A system's *trusted computing base* (TCB) is the set of components of a system which must be assumed to operate correctly in order to ascertain the reliability of *any* part of the system, and the confidentiality and integrity of its data. Note that this does not necessarily mean that all components of the TCB are actually *trustworthy*, but clearly the system can only be trusted to perform its core functionality if the TCB can be trusted too.

As embedded systems often operate in life- or mission-critical situations, they are trusted to a high degree, and one would hope that they are actually trustworthy. In many cases a certification of trustworthiness is required by governments or industry bodies. The methods used to establish such trust either have to be used from the beginning of the system's design and implementation (strict standards for systematic design, implementation and code review) or do not scale beyond systems of a few thousand lines of code (mathematical proof or comprehensive code inspection). They therefore require a minimal TCB size.

As all kernel code executes with system privilege, it is inherently all part of the TCB. The Linux kernel is too big to establish the degree of trust that is required for many embedded applications.

There is one further roadblock for the deployment of Linux in the embedded systems world, **the GPL**. Embedded systems are produced and sold in order to produce income to their developers and producers. In many cases the hardware and its interfaces are the distinguishing properties on which the market success of a device is based. The firmware is often also part of the producer's critical intellectual property, or needs protection as access to its source code would expose the company's trade secrets to a degree that would destroy its business.

It is the nature of embedded systems that such firmware must generally be tightly integrated with the OS. In the case of Linux, this means in most cases that it will become subject to the GPL and thus open source, something that is unacceptable to a company whose business is based on keeping its intellectual property (IP) secret.

Motorola's much-touted "Linux Smartphone" [Mot04] is a case in point. It runs Linux on a general-purpose processor, which is separate from the chips that implement the basic phone functionality. The Linux system is essentially used to run the user interface. The physical separation protects the core system functionality from a misbehaving Linux side, and also the firmware from the GPL.

3 Linux on a Microkernel

An approach that addresses all the above challenges is to run Linux in user mode on top of a very small and fast microkernel. A microkernel is a minimal kernel which provides no services, only fundamental mechanisms that can be used to implement services via user-mode servers. The microkernel is the only part of such a system that runs in privilege mode.

When running Linux on a microkernel, the Linux kernel becomes an unprivileged (user-mode) server process. This setup is somewhat similar to *User-Mode Linux* (UML) [Dik00]. However, UML needs to be hosted on Linux and only runs on x86. Our aim is a user-mode Linux system that requires a minimal infrastructure (and a minimal TCB), and runs on a wide range of architectures relevant for embedded systems (mostly ARM and MIPS).

In our implementation of this approach, the Linux server is called *Wombat*. The microkernel we use is the L4Ka::Pistachio [L4K] implementation of the L4 microkernel API [L4K01]. L4 has demonstrated performance that is an order of magnitude better than that of earlier microkernels [Lie93,LES+97] (which was poor and has brought the whole microkernel idea into disrepute). L4 is designed to support hard real-time applications.

The resulting system, which can be viewed as a variant of Sarma and McKenny's third approach to real-time computing with Linux [SM04], is shown in Figure 1. This approach has been pioneered by Dresden University of Technology (TUD) by their L^{4}Linux system [HHL+97]. They experienced a moderate performance degradation compared to native Linux 2.0.x on a Pentium machine. The TUD researchers have also demonstrated an RTLinux-compatible environment for user-mode real-time tasks, and have shown that its performance is not greatly degraded compared to RTLinux [MHH02].

The Iguana layer in Figure 1 provides some basic OS services, such as resource management and protection. This is necessary as L4 only provides generic *mechanisms*, no services. Iguana is discussed further in Section 4.1.

This system architecture addresses the challenges listed in Section 2 as follows:

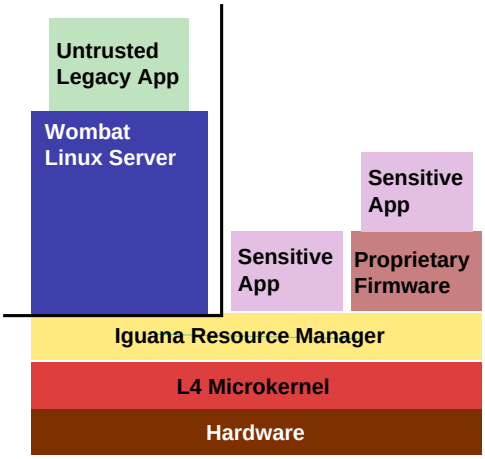

Figure 1: Running a user-mode Linux server (Wombat) and other applications on a microkernel.

- the microkernel is responsible for dealing with interrupts. L4 is designed to satisfy the needs of hard real-time systems, and ensures that real-time applications are immediately activated if a relevant interrupt occurs. The Linux system is not involved in any handling of interrupts destined for real-time tasks;
- the sensitive real-time part of the system is totally isolated from the Linux side by hardware protection (mediated by the microkernel) and by the above-mentioned way of dealing with interrupts. Even if the whole Linux world crashes or is compromised by an attacker, it cannot compromise the real-time side. Similarly, a new version of the Linux server (based on the same or a different version of the Linux kernel) can be loaded by simply restarting it, without requiring any downtime of the sensitive part of the system;
- the size of the TCB is much reduced. Instead of containing, the whole Linux kernel (hundreds of thousands of lines of code at least), it contains the L4 microkernel (about 10,000 loc) and the Iguana resource manager (about 20,000 loc). Obviously, the firmware and other "sensitive apps" are part of the TCB, but that is independent on whether it runs on L4 or on Linux. The edge of the TCB is indicated by a thick line in Figure 1. We will return to the issue of the TCB in Section 7;
- the L4 microkernel, as well as Iguana, are open-sourced under a BSD-style license, allowing it to be used with few restrictions in a commercial environment. As Linux runs as an application on top of L4 (and besides the proprietary firmware), neither L4 nor any other firmware needs to be GPL-ed.

4 System Architecture

Figure 2 provides a more detailed look at the structure of a Wombat system, and the Iguana environment that supports it.

4.1 Iguana

Iguana provides basic services, such as allocating and sharing memory, a memory protection model and its enforcement, and general resource management.[1] Moreover, Iguana supports an address-space layout that leads to a dramatic reduction of context-switching overheads on processors with virtually-addressed

[1]Resource management in Iguana is very rudimentary at this time, but the framework exists for very general and flexible management of time and memory resources.

caches, such as ARM7 (e.g. StrongARM) and ARM9 (e.g. XScale). ARM cores are among the most popular processors for 32-bit embedded systems, and efficient support for ARM is therefore essential to support widespread use of the system.

The issue with virtually-addressed caches is that data in the cache is only identified by virtual address. The virtual address is tied to a particular addressing context (i.e., the address space of a particular process), and different processes tend to use the same virtual address for different information (e.g. the start of the text segment and the top of the stack is normally the same for all processes in a Linux system, even though the code/data at those addresses are process-specific). Consequently, cache must normally be flushed on a context-switch on such a processor.

Cache flushes can be avoided when switching between processes with non-overlapping address-space layout [WH00, WTUH03].[2] Therefore, Iguana is designed to avoid address-space overlap. It does this by utilising techniques developed for so-called single-address-space operating systems (SASOS), such as Mungi [HEV+98]. In fact, the Iguana implementation shares the core part of the Mungi code base.

The core idea of a SASOS is that, rather than having all processes execute in their own (independent) address space, have one address space that is shared by all processes. Security is supported in a SASOS by distinguishing between *address translation* (which is tied to the address space) and *memory protection*. Rather than its own address space, a process gets its own *protection domain*, which is a representation of its access rights to memory. Since in a SASOS all data resides in the same address space, all data can be addressed with a pointer; however, addressability does not imply *accessibility*. Data can only be accessed if it is contained in the protection domain of the process attempting the access.

Unlike Mungi, Iguana does not *force* a single address-space view. Processes can be created in the shared address space (but in their own protection domain) or, alternatively, in an address space of their own. This is important, as on a 32-bit processor the available 4GB address space may not be sufficient to accommodate all processes. (However, as embedded systems rarely have more than 4GB of RAM, a shared 4GB *virtual* address space tends to be sufficient.)

The main advantage of running processes in the shared address space is that they processes automatically get the benefits of fast context switching on processors like the ARM. As we have shown earlier [WTUH03], this can make a 50-fold difference in the context-switching costs, and is therefore very important for embedded systems which use protected components in order to improve robustness. Another advantage (independent of the processor architecture) is that sharing of data is simplified: all data in the shared address space can be shared easily (provided that the participating processes have the right to access it) as it's guaranteed to be visible at the same address for each process, so no pointer conversions are required.

The main drawback of running in the shared address space (besides the 4GB limitation) is that this is inconsistent with the traditional address-space layout (fixed addresses for text and stack). This creates problems for some programs which make the assumption of a known address-space layout (although that is definitely bad practice). Furthermore, strict `fork()` behaviour is impossible to achieve without creating a new address space. μClinux [Arc] has to deal with the same issues, and has demonstrated that in practice this does not cause insurmountable problems for embedded systems.

4.2 Wombat

Figure 2 shows how address spaces are used in an L4-based system running Wombat. The light-coloured polygon represents the shared Iguana address space. Boxes inside represent separate protection domains (each corresponding to a process). In reality, some of these protection domains will overlap (processes sharing text or data) but the diagram does not attempt to represent this. The box sitting on its own in the top left-hand corner represents a separate address space.

Wombat (the Linux server) runs in its own protection domain inside the shared address space, thus is able to benefit from fast context switching and simplified sharing this offers. The protection domains

[2] The so-called PID-relocation of the StrongARM provides support for removing such overlap [WTUH03], but this is limited to 32 processes and works on a large, 32MB granularity, which leads to significant memory fragmentation.

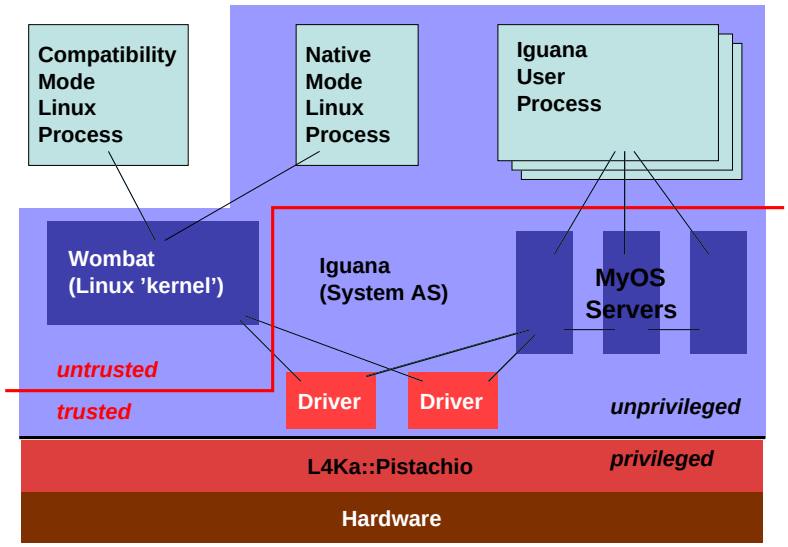

Figure 2: Wombat and Iguana.

jointly labelled *MyOS Servers* are the set of processes that provide the basic OS services, they comprise the Iguana server as well as other (possibly domain-specific) servers. Together with the L4 kernel and device drivers they make up the system's minimal TCB. Device drivers are discussed in more detail below (Sect. 5.4).

The boxes labelled *Iguana User Processes* represent application code that only depends on the system's basic OS services, but are independent of Linux (i.e., they are able to run even if the Linux server is not up, possibly because it is presently being upgraded). This would include the hard real-time components, proprietary firmware, and other "sensitive apps" of Figure 1. The intention is that all such processes execute in the shared address space, but this is not enforced. Programs can execute in their own address space, but they would lose access to most Iguana services, which are only available within the shared address space.

4.3 Linux applications

Linux applications can be run in one of two ways: either in *native mode* or in *compatibility mode*. Native mode Linux processes run inside the shared address space, which gives applications full access to Iguana services, allows them to communicate freely with other Iguana applications (including easy sharing via shared memory), and on ARM provides fast context switches between them (and any other processes running inside the shared address space, including the Linux server). Native Linux processes therefore can use the combined API of Linux and Iguana, and can even offer services to the rest of the Iguana system, so they could act as MyOS servers.

The main restriction on native mode Linux applications are the one imposed by the single-address-space model (and are familiar from μClinux): limited address space size, unusual address-space layout and no support for `fork()` (although a `fork()-exec()` sequence is available as in μClinux, which is sufficient for most applications). While these restrictions are not a problem in most cases, there are some Linux applications which will not run in native mode without significant porting effort.

Compatibility mode is provided to accommodate such applications. As the name implies, full binary Linux compatibility is supported for applications running in this mode. The drawback of compatibility mode is that such applications are much less well integrated into the rest of the system: They can only communicate directly with Wombat (using Linux system calls), not with any other processes executing in the shared address space, and communication with other separate address spaces is only possible via the standard Linux mechanisms (files, pipes, sockets, `mmap()`, SysV IPC, ...).

5 Wombat Implementation

The current implementation is capable of running small embedded style system images containing, for example, the busybox shell. As of time of writing only compatability mode support is available. Wombat is, however an active project, and we aim to run a full Debian-based system on top of it in the near future.

5.1 Structure

Wombat is designed for portability, between processor architectures, hardware platforms, and Linux kernel versions. In particular we want to minimise the cost of supporting additional architectures and the maintenance cost resulting from changes in the Linux kernel.

In order to achieve these goals we introduced a new processor architecture, L4, into the Linux source tree, by adding new directories `arch/l4/`, and `include/asm-l4/`. In order to keep this as architecture-neutral as possible, we resisted the temptation of hacking one of the existing architectures, but instead implemented the `l4` architecture from scratch. Any architecture-specific code (of which there is very little) lives in a subdirectory, e.g. `arch/l4/sys-arm/` for ARM. The initial development was done mostly concurrently for MIPS and ARM and later ported to x86. Ports to PowerPC and Alpha are in progress (as low-priority background activities).

Only three files in the existing Linux source tree were touched: two one-line bug fixes, plus some additional early debugging output in `printk`. Hence, the Wombat implementation easily drops into the existing source tree. The implementation was started in late 2003 using the 2.5 series kernel, and has since been ported to the 2.6 series kernels, the most recently supported version is 2.6.5.

5.2 System calls and exceptions

A Linux process is implemented as a single-threaded L4 process. Like any other process, it can perform L4 system calls, such as L4 message-passing IPC to communicate with other processes in the system. (Note that only native-mode Linux apps are expected to do so, as compatibility-mode apps are built for a native Linux environment and do know nothing about L4. While there is no reason why a compatibility-mode app cannot use L4 system calls, we use L4's security mechanisms to prevent compatibility-mode apps from sending L4 IPC to any process other than Wombat.)

L4 uses different syscall numbers than Linux. Hence, when a Linux app performs a Linux system call, the L4 kernel treats this as an exception, which is mirrored back to Wombat using L4's user-level exception-handling mechanism, a process called *syscall redirection*, or also *trampoline*. To Wombat this looks like a normal IPC message received from the user process. It will process the Linux system call normally (by invoking the standard Linux kernel services) and will return directly to the application process. This is shown in Figure 3.

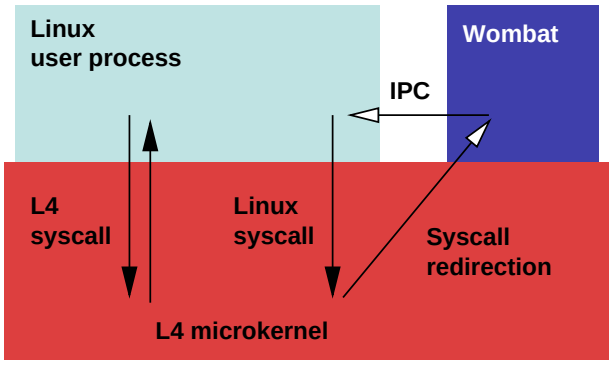

Figure 3: Linux system call redirection.

On the ARM, rather than using different syscall numbers, we actually use a different mechanism for L4 and Linux system calls. Linux on ARM uses the `swi` (software interrupt) instruction to trap into the system call handler. L4, in contrast, uses a jump to an invalid address, which leads to a *prefetch abort* exception. This simplifies decoding of system calls in L4, and eases the distinction between L4 and Linux system calls.

Other exceptions are handled similarly, by L4 converting them into IPC messages to Wombat. Invocation of a signal handler is done in an architecture-dependent fashion, emulating the standard Linux way of handling signals.

Page-fault handling is done by expanding the macros that access the page table and TLB by invoking the appropriate L4 mapping operations.

5.3 Scheduling

The basic idea is that scheduling decisions for Linux processes are done by the normal Linux scheduler. This requires a little bit of work, as L4 has its own scheduler, based on hard priorities (using round-robin within priority levels).

Normal L4 scheduling is used for other Iguana processes, including Wombat itself. The L4 priorities are thus used to schedule the complete Linux part of the system with respect to the other (real-time) processes, but not for scheduling Linux processes with respect to each other. The Iguana real-time processes are normally given higher priority than Wombat, which ensures that runnable real-time processes will always preempt the Linux side of the system. Linux applications will only run if no (higher-priority) real-time process is runnable.

In order to implement proper Linux scheduling for Linux processes, Wombat ensures that only a single Linux user process (per CPU) is runnable at any time. This way the L4 scheduler is forced to schedule Linux apps in a way that is under the control of Wombat. A separate *timer thread* (with an L4 priority higher than all Linux processes) exists inside the Wombat protection domain in order to maintain the Linux time slice. This thread normally waits for a timeout (corresponding to the Linux timer tick). When that timeout occurs, the timer thread is woken, which then calls the Linux scheduler in order to determine the next process to run (according to normal Linux scheduling policy).

In order to run the Linux user process determined by the Linux scheduler, the presently executing one must be preempted and blocked. At present, this is deferred until that process next traps into the kernel (to perform a system call or due to an exception). In the worst case (CPU-bound process), this may not happen until another timer tick (where we force an immediate preemption). In such a case, the the user process runs too long (up to twice its proper time slice). As such, Linux scheduling policy is only approximately observed in the present implementation.

This will be fixed in the near future. The reason it has not been done yet is that there are several ways to achieve the desired result in L4, and we need to implement them all and analyse the cost.

The main advantage of this approach is that the scheduling decision is made by the *unmodified* Linux scheduler, thus avoiding changes to the architecture-independent code of Linux. Furthermore, this approach requires no locks other than what is already in the Linux kernel (and will therefore automatically benefit from all improvements made to locking in Linux).

The Linux idle task calls the architecture-specific `cpu_idle()` function. In Wombat this is implemented as an L4 IPC message to the Wombat “kernel” thread, which simply sleeps until the next timer tick or interrupt. This effectively hands control to the L4 idle thread, unless there is an Iguana thread with a L4 priority lower than that of the Linux processes. Such a thread could be used to put the system into a low-power mode.

5.4 Device drivers

Each device in the system must be controlled by a single driver, either a Linux driver or an Iguana driver. Iguana drivers run as user-mode processes inside their own Iguana protection domain. Standard Linux

Benchmark	Linux	Wombat	UML
syscall	0.392	1.38	8.99
pipe	5.05	8.83	66.8

Table 1: Latency benchmark performance

drivers can be used in Wombat unmodified. Such a simple setup does not support sharing of device access between the real-time and Linux parts of the system.

Shared device access requires that one side contains the proper driver, and the other side contains a stub (or proxy) driver, which, when invoked, forwards the request to the other driver. In most cases, this means that the proper driver is an Iguana driver, in order to ensure that competing accesses by real-time and non-real-time components are correctly arbitrated. Also, the encapsulation of the Linux side would be undermined if Linux drivers were allowed to perform DMA.

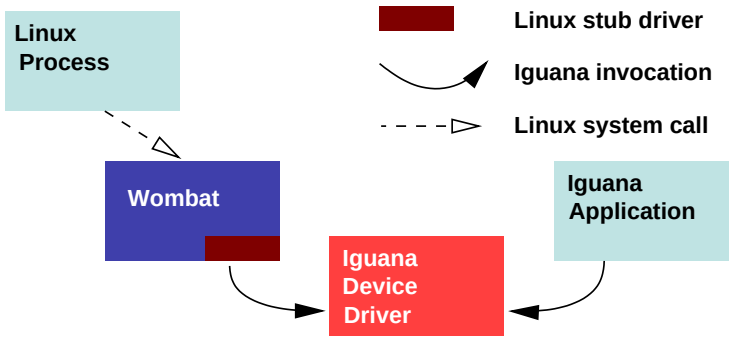

Figure 4: Shared device drivers in a Wombat system.

Figure 4 shows such a setup. An Iguana process can directly invoke the driver. A Linux app invokes it via a Linux system call, in response to which Wombat invokes a (statically or dynamically loaded) stub driver, which in turn invokes the proper driver using L4 IPC.

Iguana's device driver model supports user-mode drivers with performance close to that of in-kernel drivers. The drivers themselves are portable not only across architectures and platforms (as long as they support the same devices) but also between Iguana and Linux — the user-mode driver work presented by Chubb at OLS'04 [Chu04] uses Iguana drivers. Presently we have drivers for several PCI chipsets, Ethernet (10/100/1000Mb/s), IDE disk, serial, LCD screen, and a number of proprietary devices.

6 Performance

We compared the performance of Linux, Wombat and User Mode Linux (UML), on a set of microbenchmarks from the lmbench [MS96] suite. The main impact of running Linux as a server will be an increase in the cost of system calls. The first benchmark used is the "null" system-call latency, which shows the overhead on individual system calls, and represents a worst-case example, as it measures solely the cost of communicating with Linux. The second benchmark, pipe latency, shows the extra cost incurred when communicating between different processes. Finally we examine the extra overhead incurred when switching between running processes.

Measurements were performed on a 2.8GHz Pentium 4 machine. The Wombat server was based on the 2.6.5 kernel. The Linux results were based on 2.6.8.1. The UML results were from 2.6.9-rc3 kernel, with skas enabled.

Table 1 shows the system call and pipe latency on each of the three systems. All figures are in microseconds and represent the average of 100 repetitions.

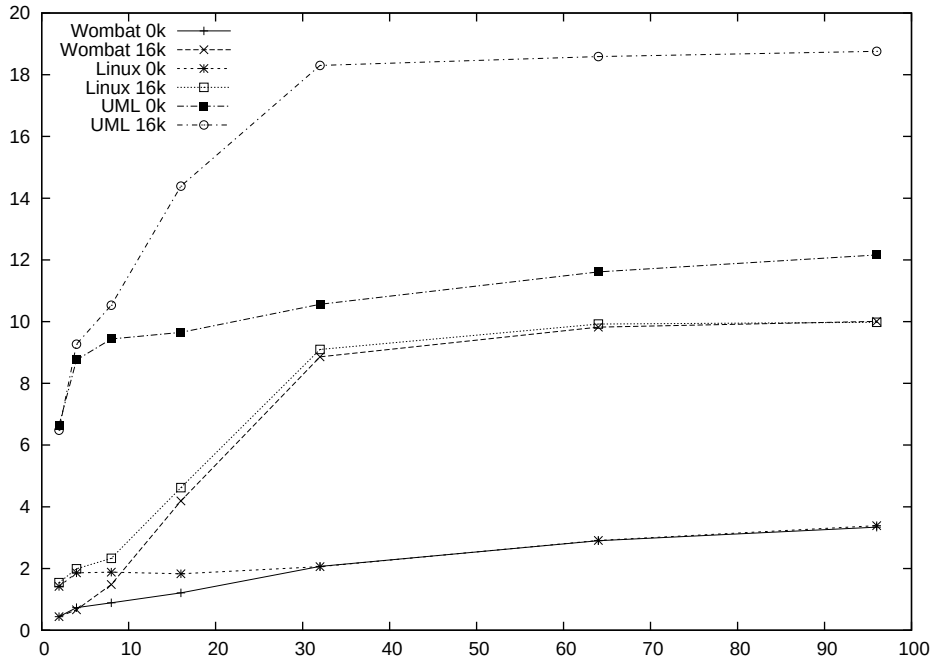

Figure 5: Context switch overhead in each system

As the table shows, the worst-case slowdown represented by the null system call is a factor of 3.5 for Wombat, compared to 23 for UML. Similarly, the pipe test under Wombat is a factor of 1.75 slower than for native Linux, while for UML the penalty is more than a factor of 13.

Figure 5 shows the context switch time, (in microseconds), for each of the three systems. The number of active processes was varied from 2 to 96. Each system was measured with a 0 and 16 kilobyte cache footprint.

This benchmark shows a similar picture to the others: performance of the Wombat system is close to that of native Linux, while UML performs dramatically worse. In fact, context switching in the Wombat system is faster than in native Linux (this warrants further research). UML exhibits a large overhead of around 8μs per context switch, almost independent of the number of processes or the cache size. This is consistent with the overhead of the null system call.

The relative performance of Wombat compared to the native kernel is comparable or better than that seen in L^{4}Linux system [HHL$^+$97]. Taking a portable approach to Linux on L4 has not led to a significant performance degradation.

7 Conclusions

We believe that running Linux as a user-mode server on the L4 microkernel is a promising way of providing a Linux API (and all the benefits of a Linux system) on embedded systems, while keeping the trusted computing base small. This view is supported by the fact that a number of companies are already using Wombat and are considering deployment in actual products.[3]

The specific strengths of Wombat, compared to similar approaches, are:

- portable across architectures. It presently runs on three, with more in the pipeline. Treating L4 as a new architecture means that it is easy to port Wombat to an architecture where Linux is not yet running, as long as there is a working L4 port. For example, Wombat based on the 2.6 kernel runs on ARM, while native Linux 2.6 doesn't yet;

[3] Unfortunately we cannot be more specific at this time, but hope that we can talk about some of the work by the time of the conference.

- its dependency on Linux kernel internals is low, making it relatively easy to keep it in sync with the Linux distribution. For a number of months we have been tracking the Linux 2.6 head revision without major effort;
- as Wombat runs in the Iguana environment, it automatically benefits from the fast-address-space-switching (FASS) support built into L4 [WTUH03]. This is without requiring FASS support in Linux (see FASS paper submitted by Peter Chubb);
- Wombat uses the Iguana driver framework, which allows it to make use of drivers that can run (unmodified) in Iguana as well as native Linux.

There is work underway at NICTA which has the potential to dramatically increase the attraction of the Wombat approach: a formal verification of the correctness of the microkernel [NIC]. This would make a critical part of the trusted computing base provably *trustworthy*, and can be followed up by verification of other components of the TCB. Formal verification is, for the foreseeable future, only possible for moderately-complex systems, the microkernel (consisting of about 10,000 loc) is at the limit of this, the Linux kernel is way beyond.

8 Availability

Wombat has been running in our lab since late February 2004, and presently runs on x86, ARM and MIPS. It is scheduled for public release in late October 2004. While Wombat itself (i.e., the contents of the `arch/l4/` directory) is naturally under the GPL, the Iguana system will be released under a BSD-style license.

References

[Arc] Arcturus Networks Inc. μClinux. http://www.uclinux.com.

[Chu04] Peter Chubb. Get more device drivers out of the kernel! volume 1, pages 149–161, 2004.

[Dik00] Jeff Dike. A user-mode port of the linux kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, Georgia, USW, October 2000.

[EDC03] Embedded systems software development survey. Research report, Evans Data Corporation, Santa Cruz, CA, USA, Fall 2003.

[FSM] FSMLabs. RTLinux. http://www.fsmlabs.com/products/openrtlinux/.

[HEV+98] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, July 1998.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on OS Principles (SOSP)*, pages 66–77, St. Malo, France, October 1997.

[L4K] L4Ka Team. L4Ka::Pistachio microkernel. http://l4ka.org/projects/pistachio/.

[L4K01] L4Ka Team. *L4* e*X*perimental *Kernel Reference Manual*. University of Karlsruhe, version 4-x.2 edition, October 2001. http://l4ka.org/projects/version4/l4-x2.pdf.

[LES+97] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Herrman Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance (still the foundation for extensibility). In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 28–31, Cape Cod, MA, USA, May 1997.

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on OS Principles (SOSP)*, pages 175–88, Asheville, NC, USA, December 1993.

[Lin04] Linux now top choice of embedded developers. http://www.linuxdevices.com/news/NS2744182736.html, August 2004.

[MHH02] Frank Mehnert, Michael Hohmuth, and Hermann Härtig. Cost and benefit of separate address spaces in real-time operating systems. Austin, TX, USA, 2002.

[Mot04] Motorola launches enterprise Linux smartphone in China. http://www.linuxdevices.com/news/NS5920529122.html, February 2004.

[MS96] Layy McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 USENIX Technical Conference*, San Diego, CA, USA, January 2996.

[NIC] National ICT Australia's L4 microkernel verification pilot project. http://nicta.com.au/director/research/programs/fm/research_projects.cfm.

[RTA] RTAI — realtime application interface. http://www.aero.polimi.it/~rtai/.

[SM04] Dipankar Sarma and Paul E. McKenney. Issues with selected scalability features of the 2.6 kernel. 2004.

[WH00] Adam Wiggins and Gernot Heiser. Fast address-space switching on the StrongARM SA-1100 processor. In *Proceedings of the 5th Australasian Computer Architecture Conference (ACAC)*, pages 97–104, Canberra, Australia, January 2000. IEEE CS Press.

[WTUH03] Adam Wiggins, Harvey Tuch, Volkmar Uhlig, and Gernot Heiser. Implementation of fast address-space switching and TLB sharing on the StrongARM processor. In *8th Australasian Computer Systems Architecture Conference (ACSAC)*, Aizu-Wakamatsu City, Japan, September 2003. Springer Verlag.