LKL: The Linux Kernel Library

Octavian Purdila, Lucian Adrian Grijincu, Nicolae Tapus
Computer Science and Engineering Department
University Politehnica of Bucharest
Bucharest, 060042
Email: {octavian.purdila,lucian.grijincu,nicolae.tapus}@cs.pub.ro

***Abstract*—The Linux kernel is a repository of high-quality, extensively reviewed and tested, but highly interconnected code. There have been several attempts to free this code and employ it to other uses, and, in each case, great amounts of resources were dedicated to decoupling the subsets that were needed from the rest of the kernel. Successful attempts faced significant maintenance issues, as small projects find themselves unable to keep pace with upstream kernel development. In time, differences between the extracted code and its mainline counterpart get bigger and it becomes very difficult to benefit from bug fixes, new features and improvements implemented in newer versions of the Linux kernel.**

The Linux kernel library (LKL) enables other projects to use code from the Linux kernel, without requiring them to isolate the needed pieces of code, separate them from other tightly knit components they may interact with, and extract them from the kernel. At the same time it allows LKL based applications to transparently make use of new features and bug fixes developed in the upstream version of the Linux kernel. Several proof-of-concept applications indicate that complete subsystems such as file system drivers or the networking stack can be easily used in applications running in environments as diverse as the Linux and Windows user space, or the Windows kernel space.

Keywords – Linux kernel; operating systems; virtualization

I. Introduction

With almost 8 million lines of C code (as of version 2.6.34) [1], the Linux kernel is one of the largest software projects in the world. Thanks to its development model, the great number of contributors and reviewing stages, it hosts relatively high quality code [2], [3], [4] for a range of complex software components which may be of use outside of the Linux kernel, like the TCP/IP networking stack or file system drivers.

Some projects that found themselves in need of functionality implemented in some of the Linux subsystems chose to reimplement it from scratch. There are a number of independent implementations for `ext2` (the Second Extended File system) that were written without directly using code from the Linux kernel: Windows drivers Ext2 FSD [5] and Ext2 IFS [6], Explore2fs Windows GUI explorer [7], Haiku Ext2 file system driver [8], ext2lib library for Windows 3.x/9x [9], or the e2fsprogs file system utilities [10]. The same kind of functionality was reimplemented in different languages, for different operating operating systems; as a kernel module, an application and a library; but with different levels of support, different performance and different bugs. Little of this work can be reused when re-implementing other file systems or other subsystems of the Linux kernel; this is one of the reasons why there is now incomplete support or no support for similar, but more complex file systems such as `ext3` or `ext4`.

Due to complexity issues, it is not surprising that a number of projects chose to extract portions of the Linux kernel and reuse them instead of starting from scratch [11], [12]. However, selecting the necessary code subset, separating that part and maintaining it have been done separately by each project, consuming resources and needing highly-skilled developers. Once separated the code usually stagnates, as bug fixes or features added in the upstream kernel are introduced very slowly, if at all. The reverse is also true, since features or bugs fixed in the project are almost never pushed back into the upstream kernel. The tremendous pace at which the Linux kernel is changing [13], is without doubt the lead factor for the maintainability problems of this approach.

The Linux kernel library project is organizing the Linux code in a form such that it can be easily reused by an application [1]. When using LKL, the effort spent on getting access to Linux kernel functionality is reduced to compiling the Linux kernel (including our patches) and linking the resulting library into the application.

Not having to customize kernel code to work with their application, developers can concentrate on creating new applications based on the functionality provided by the Linux kernel code. As a bonus, upgrading to newer versions of LKL automatically brings bug fixes and new features developed in the main Linux tree, freeing developers from the time consuming tasks of monitoring the changes done on the kernel subsystem they are interested in, extracting relevant changes and patching their own branch of modified kernel code.

The project allows the application to make use of full Linux kernel subsystems like the virtual file system [14], the networking stack, task management and scheduling, and even memory management to some extent. LKL can run in any environment, as long as the environment provides a few basic primitives. We have demonstrated that it can be used in both user space (Linux, Windows) and kernel space (Windows).

The rest of this paper is organized as follows: section 2 deals with architecture issues; section 3 presents some of the applications built using LKL; performance is evaluated in section 4; section 5 surveys related work, and section 6 provides conclusions and outlines future work.

[1]In this paper we use the term application in its broad scope, to identify generic code that makes use of LKL; the application we refer to can be a program running in user space or a driver running in kernel.

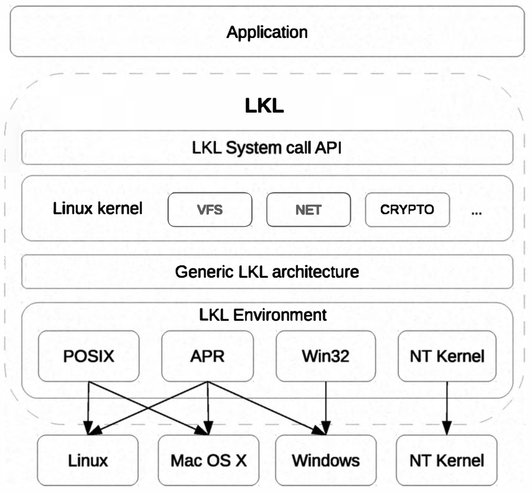

Fig. 1. The architecture of the Linux kernel library.

II. Architecture

The main goal of LKL is to provide a simple and maintainable way in which applications running in environments as diverse as possible can reuse Linux kernel code. Specifically, LKL should not require a particular operating system, it should be usable from both user and kernel space, it should allow easy tracking of the Linux kernel tree, it should require minimum "glue code" from the application and should provide a stable, and easy to use API for the application.

In order to be able to easily upgrade LKL to future Linux kernel releases, we require a mechanism to cleanly separate the LKL specific components from the mainline kernel.

Similarly to User-mode Linux [15], we chose to implemented LKL as a port of the kernel to a virtual computer architecture, named *lkl*. As a result, we do not need to change any of the core kernel components. In fact, the only non-architecture Linux kernel changes are build system related.

Since LKL is environment agnostic, the *lkl* architecture does not use any platform dependent code. Instead, the application needs to provide implementations for a small set of environment dependent primitives. These primitives, named *native operations* from hereafter, are used by the generic *lkl* architecture to build the virtual machine upon which the Linux kernel will execute.

To interact with the kernel, the application uses the LKL system call interface, an API based on the Linux system call interface. This assures that we offer a stable and familiar API to the application developers.

A. The LKL Architecture

The LKL port interacts with the application via an interface which includes the LKL native operations, the LKL system calls and an interrupt-like API which allows the application to notify the Linux kernel about external events. To define the specifics of this interface we needed to identify the architecture level primitives that the Linux kernel requires and then to adapt them to our context.

It quickly became apparent that not all primitives that need to be offered by an architecture layer for a full Linux port were required in our case. For example, since we use the Linux kernel for a single application, we do not need address space separation and protection between user and kernel or between multiple user address spaces. In fact, we do not even need some of the user space abstractions that the kernel offers, like user processes, process address spaces or signals. Besides simplifying the implementation, this allows us to also simplify the interface between the application and LKL. One of the most visible effects was that we could directly link LKL into the application.

1) Memory Management Support: Because LKL does not require memory protection mechanisms, support for memory management becomes trivial: LKL just needs a "physical" memory pool that the Linux kernel can use for its memory allocation needs. The actual memory reservation is under the control of the application: it controls both the allocation mechanism and the size of the memory pool. The poll of memory will be managed by the kernel using a mix of buddy, SLUB/SLAB/SLOB/SLQB algorithms just like on a normal system. This pool will only be used for buffers and structures dynamically allocated by the kernel. The kernel code and statically allocated data will not be part of this poll as they are managed by the external environment that loads LKL.

Note however, that some subsystems (e.g. VFS) need virtual memory management support. Fortunately, the Linux kernel implements the virtual management API even on architectures that don't have a MMU (e.g. the Motorola m68k platform) and hence no MMU emulation is needed in LKL itself, we just need to declare LKL as a non-MMU architecture.

2) Thread Support: Even though we do not need to implement support for user processes, we need to offer support for kernel threads. The Linux kernel uses such threads for internal house-keeping, like processing I/O requests or running softirqs, tasklets or workqueues. During early stages of design, we considered implementing threads internally, without explicit support from the environment.

However, completely portable implementations have various limitations. For example, implementations based on `setjmp - longjmp` require usage of a single stack space partitioned between all threads. As the Linux kernel uses deep stacks (especially in the VFS layer), in an environment with small stack sizes (e.g. inside another operating system's kernel) this will place a very low limit on the number of possible threads. Other solutions are either limited in similar ways or require some form of help from the environment.

Thus, we deferred the actual implementation to the application, which has to provide two basic primitives: to create a new thread and to terminate a thread.

3) Thread Switching: The threads used by LKL are implicitly scheduled by the environment, but Linux needs to control scheduling of its threads to function correctly and efficiently: RCUs [16] make assumptions about scheduling policies; Linux threads synchronized with Linux semaphores (not the environment's semaphores) need to sleep and switch

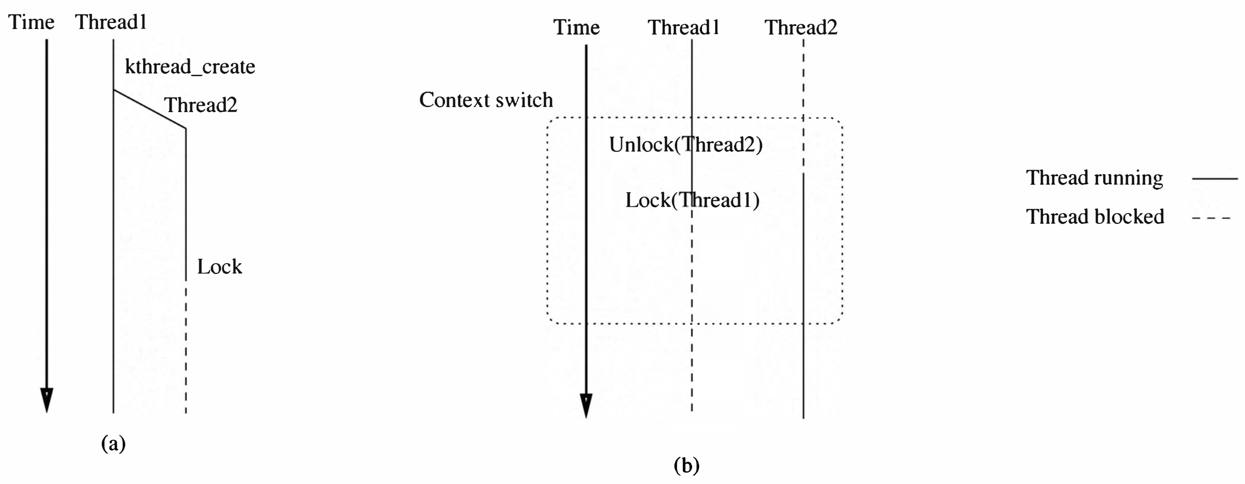

Fig. 2. A typical thread creation (a) and thread context switch (b) in which mutexes are used to give the Linux scheduler control of over the native threads.

to other Linux threads.

To regain control of scheduling, the generic LKL architecture layer associates an environment-provided semaphore with each LKL thread. Immediately after creation, and before running any Linux code each LKL thread acquires its corresponding semaphore, and gets blocked as the semaphore's initial value is 0. When the Linux scheduler selects a new thread to run, it releases the semaphore of the new thread and immediately acquires its own semaphore. The new thread will begin running and the old one stops (see figure 2). This token passing mechanism ensures that at any given time there is only one thread running and the scheduling order is dictated by the Linux scheduler. The environment cannot interfere in the scheduling order of the Linux threads as any thread not deemed runnable by the Linux scheduler waits on an environment-provided semaphore and is unschedulable by the environment.

Semaphore support must provided by the environment as a set of native operations. LKL needs basic and portable semaphore operations: allocate a semaphore with a given initial value, free a semaphore, and the traditional `up` and `down` operations.

4) IRQ Support: In a typical scenario an application that uses LKL needs to have LKL interact with some external entity: e.g. a driver for another operating system for a Linux file system like ext4, needs to read data from a disk. Such an application will use two device drivers to represent the disk in question: a Linux block device driver and a native kernel device driver.

The Linux device will act as a translator between the Linux kernel and the native kernel: the Linux device driver programs an I/O request by calling the native device driver, which in turn programs the hardware to do the I/O operation.

At some point the hardware device completes the I/O operation and generates a native IRQ. The native device driver processes it, and needs to signal to the Linux kernel that the operation has been completed. Since the native IRQ is asynchronous, we also need to signal the Linux kernel asynchronously that the I/O operation has been completed. This translates in a need of IRQ support in LKL.

Part of the API that LKL offers are operations to trigger IRQs. The application specifies the IRQ number it wants to generate and, optionally, (a pointer to) some data. Simple IRQs are used when the driver has nothing to communicate apart from the generation of that interrupt. Timer interrupts are of this kind as timers are periodic events and the kernel knows the period with which timers fire.

The "with data IRQs" are used by native device drivers to communicate the context of the IRQ (e.g. the operation that was completed, the completion status: error or success) to the Linux device driver.

The current implementation of LKL has neither SMP, nor preemptive support. Because the native environment could be SMP and it could trigger IRQs from a thread running parallel to the currently active LKL thread, we need to serialize IRQs handlers and LKL kernel threads. Our implementation creates a queue of outstanding IRQs which are handled serially from the idle thread.

5) Idle CPU Support: When no thread is runnable, Linux runs the so called "idle thread" (for historical reasons this thread is sometimes referred as the "swapper thread"). The idle thread's job is to throttle down the CPU via special architecture specific CPU instructions. Ideally the CPU should enter a low power mode and stay there until an asynchronous event like an IRQ wakes it up.

LKL cannot throttle down the CPU in the idle thread, because it must be used in user space environments, where there is no access to the necessary low level CPU instructions. And even if those instructions were available, it would be inappropriate to throttle down the CPU because, while there might not be any runnable LKL thread, other native threads might be schedulable. On the other hand, busy waiting for simulated IRQs wastes power and hogs the CPU. The proper way of handling the idle CPU issue would be to put the idle thread to sleep when there is nothing to do, and to wake it up when an IRQ has been signaled.

However, since LKL doesn't control native thread scheduling, it needs help from the application efficiently manage the idle state.

Initially we required the application to provide two native operations: `enter_idle` and `exit_idle`. `enter_idle` was called from within the idle thread and `exit_idle` was called by the IRQ triggering routine. A typical implementation used a semaphore `down` operation for `enter_idle`, and an `up` operation for `exit_idle`. As need of semaphores is necessary in other components as well, we discontinued this approach. Instead, we now require basic semaphore operations from the environment. This simplifies both our implementation and the native operations requirements.

6) Time and Timers Support: A Linux kernel without time support would not be very useful: the Linux kernel uses the current time to update the access and modification timestamps of files and timers are extensively used in the network stack. The RCU synchronization mechanism also needs a timer to function properly.

LKL requires both a timer and a time source which must be provided by the application via two native operations: one to return the number of nanoseconds that passed since the start of the Unix epoch, and one which should receive the number of nanoseconds after which the application should trigger an `IRQ_TIMER` interrupt.

B. LKL System Call API

The API that LKL offers to the application is a subset of the Linux system calls. As LKL is linked into the application, the application has access to every symbol exported by the Linux kernel. Even though it *can* make direct calls to any exported kernel function, this would not be generally appropriate, as it would skip safeguards in the kernel. The application should only make use of the public kernel API: system calls. Because LKL does not have SMP support, it would not be appropriate for the application to directly invoke system call handlers due to possible races between a system call handler, kernel threads, IRQ handlers, or other parallel system call handler invocations. Thus, we need to properly serialize system call handlers with respect to LKL threads and IRQ handlers.

The application accesses the kernel through a set of predefined system call wrappers. These functions issue "with data IRQs" with the IRQ number set to `IRQ_SYSCALL` and the data set to a structure in which the system call number and parameters are stored. Next, a slot for the system call result is initialized and a new native semaphore is allocated. They are both stored in the same structure. Finally, the calling thread sleeps on the semaphore until the system call results are ready.

The IRQ handler adds all system call requests to a `work_queue`. Normally, when the Linux kernel finishes its initialization it runs an `init` process, but LKL cannot do that as it does not support user space processes. Instead LKL runs a special-purpose routine which waits for events on the system call `work_queue`. For each request, the `init` thread calls the appropriate system call handler, puts the value returned by the handler in the result field of the structure associated with the system call and releases its semaphore unblocking the original calling thread. Unblocked, the system call wrapper frees the associated structure and returns the result to its caller.

C. API helpers

An application only requires the use of the LKL system call API for most of its work. However, there are some frequently used operations, which can be implemented only using portable LKL system calls, but which require a non-trivial amount of work. In order to ease application development, LKL provides a set of API helpers such as mounting and unmounting a file system, assigning an IP address to an interface, adding a default route, etc.

D. LKL Environments

LKL needs to access the environment through native operations. Initially the native operations were implemented by the application, but it quickly became apparent that for a given environment (e.g. Linux user space, Windows kernel, etc.) the implementation is independent of the application itself.

Thus we decided to implement the native operations in LKL itself in order to simplify the job of writing and maintaining a LKL application. So far we have implemented support for the following LKL environments: POSIX, NT (Windows user space), NTK (Windows kernel), and Apache Portable Runtime.

It's worth mentioning that adding support for a new LKL environment is a straightforward job. One needs to provide operations for printing a message to the console (a `printk` equivalent); allocating, freeing, acquiring and releasing a semaphore; creating and destroying a thread; allocating and freeing memory; retrieving the current time and scheduling a LKL timer interrupt some time in the future.

III. Anatomy of a LKL Application

A typical LKL application has a few standard components (figure 3): the Linux kernel library – configured to include the features that the application needs (e.g. the ext4 file system, the TCP/IP networking stack, etc.), application specific drivers (usually split in two parts: a Linux device driver and a native stub), the implementation for the native operations (applications sharing the same environment can share this part), and the application specific code.

The interactions between the various components of the application, as depicted in figure 3, can be summarized to the following:

- the application makes a call into the Linux kernel, via the LKL system call API
- the Linux kernel issues a request to a device driver (e.g. read/write data from/to the disk)
- the device driver calls the native stub
- the device driver stub programs the request in the environment (depending on the environment, this could be a system call, a request to a native device driver, or some other environment dependent operation)
- the environment reports to the native stub that the request was completed (via an native IRQ, or other native notification mechanism)

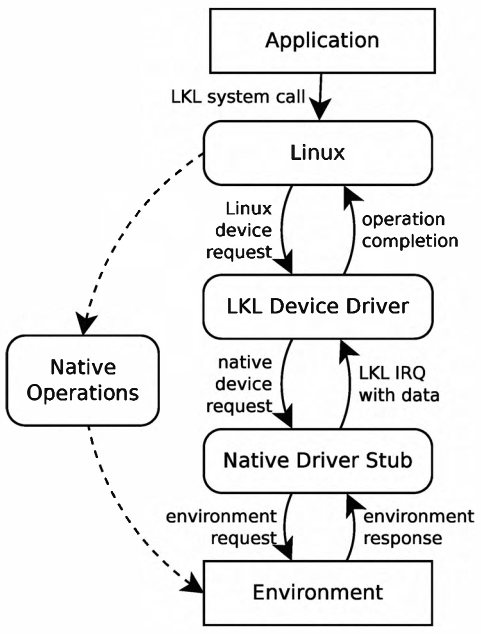

Fig. 3. Components and mode of interaction for a typical LKL. application.

- the native stub notifies the Linux kernel that the operation has been completed via an LKL interrupt
- the device driver notifies the Linux kernel that the operation has been completed

Note that the Linux kernel is also calling various native operations throughout the lifetime of the application, either from the Linux kernel threads or from the threads that run the LKL system calls.

IV. Evaluation

This sections presents a few proof of concept applications we have written in order to evaluate the usefulness of LKL in developing aplications that use the Linux kernel to perform complex tasks.

The first application that we developed on top of LKL is a portable FTP server which can read file and disk images formatted with Linux kernel supported file systems. We based our implementation on the the Apache Portable Runtime Library API [17] to ensure portability across different platforms.

While this application is an effective way of testing the LKL model, it is also a useful application in itself. FTP clients are available on any major platform, and the FTP protocol is powerful enough to provide the user a “file system like access”. In most major operating systems the GUI already presents the same interface for file system access and FTP access. Thus, it can be of real use for people that use Linux and another operating system in a dual boot mode, or for people wanting to read Linux formatted media in a non-Linux operating system.

We are yet to fully characterize the performance, but preliminary tests indicate that we can achieve similar throughput (approx. 40MB/s with a Pentium 4 2.00GHZ machine) for both the native path (i.e. FTP daemon using a natively mounted EXT3 file system) and the LKL path (i.e. FTP daemon using LKL to directly read from a partition).

Next, we have developed our first kernel application to use LKL. It is a generic Windows file system driver that can access read-only (at this time) any Linux file system [18]. All of the Windows file system operations are translated into Linux system calls (such as open, read, write, etc.) which go through LKL, the corresponding Linux file system driver, and eventually reach the native disk device driver. The Windows part of the file system driver is based on code from the ext2fsd [5] project.

Then, we decided to try and reuse the Linux networking stack. First, we have created a simple HTTP client that reuses the full Linux networking stack [19]. Performance tests indicate that even with no virtualization optimizations (e.g. GRO and GSO) we were able to achieve 150Mbps throughput on an embedded system (PowerPC 800MHz processor). Next, we started working on a realistic network simulator [20] which uses the full Linux networking stack for each node it simulates (which can be an end-station, a switch, a router or a firewall).

V. Related Work

This is the first attempt, to our knowledge, which attempts to reuse Linux kernel code at such a large scale and in such diverse environments. However, there are other works with different objectives relevant for our project.

User mode Linux is a user space virtual machine that is capable of running unmodified Linux applications. What differentiate UML from other virtualization systems like Xen [21], VMWare [22] or KVM [23] is that it does not require host kernel modification[2].

While UML is capable of running the Linux kernel in user space Linux, and there are efforts to port it in order to run in user space Windows, LKL is much more flexible as it already runs in user space Linux, Windows and in the Windows kernel. Also, it is easy to port to new environments as long as `gcc` is supported in that particular environment.

At this point, we should mention that LKL borrowed two essential and elegant ideas from UML’s own implementation. First, to reduce intrusive modifications to the Linux kernel, we implemented LKL as a port of the Linux kernel to a new virtual architecture. Second, we followed UML’s approach of using host threads instead of implementing micro-threads within LKL.

CoLinux [24] is a port of the Linux kernel which allows it to run as a lightweight virtual machine in Windows, with the same privileges as the Windows kernel. It achieves this by periodically switching between the host (Windows) and the guest (Linux) kernels. While running, both of the host and guest kernel have complete ownership over the hardware, including physical memory, MMU and interrupts. However, only the host kernel will manage some parts of the hardware. For example, the physical memory used by the guest kernel

[2]currently UML uses host kernel modifications for performance optimization, but they are not required

is allocated by the host kernel, and interrupts received while the guest kernel is running are forwarded to the host kernel. This scheme applies to the hardware resources that cannot be time shared. Other resources, such as the MMU and CPU are completely managed by both kernels, while they are active.

WinVFS [11] is one of projects that tried to reuse Linux kernel code on a large scale. The project ported the Linux virtual file system layer to Windows with the goal of creating Windows drivers for ext3, ReiserFS and other file systems supported by Linux. While the goal coincides with one of LKL's application's goals, the method used was quite different. WinVFS extracted parts of the Linux kernel and then adapted them so that they would compile without the rest of the Linux kernel code. Because of dependencies in the Linux kernel code, WinVFS pulled in and manually adapted significant amounts of code which at a first glance is not related to the VFS layer: spinlocks, semaphores, bit operations, the Linux kernel SLAB allocator, the page allocator, string operations. Even with all this code pulled in, modifications to the file system code were required. Even more problematic was the maintenance, which in the end was the factor that defeated the project. When WinVFS was started it used code from Linux kernel 2.4. By the time the prototype was completed, Linux 2.6 was released. A 2.6 porting attempt was started and it lasted a few months, but by that time the 2.6 rapid development model yet again drove WinVFS's base code obsolete.

RUMP [25] (Runnable Userspace Meta Programs) is a NetBSD project which compiles the NetBSD kernel into a library which can be used in userspace programs. While initially the project was targeted at NetBSD operating systems, there are projects underway [26] which have the goal of running RUMP in other operating systems (so far Linux and FreeBSD). RUMP has been used in fs-utils [27], a userspace application which allows file system access with UNIX-like commands (e.g. ls, cat, chmod, etc.) on a total of 12 filesystem supported by NetBSD; as a more convinient way of doing NetBSD kernel development [28] or as a proof of concept application which runs NetBSD networking stack in userspace [29].

VI. Summary and Future Work

In this paper we have shown that LKL can be used to easily create applications that reuse large and complex parts of the Linux kernel. We have also shown that LKL can be used in a very diverse set of environments, and that LKL can be easily ported to a new environment.

We continue working on LKL to improve its performance and to add support for other environments (e.g. we would like to port LKL to Mac OS X and the Haiku OS). We are also working on the proof of concept applications presented in this paper in order to expand them into useful tools.

You can explore the LKL code as well as the code of the projects described in the Evaluation section at *http://github.com/lkl.*

References

[1] "Ohloh source code analysis for linux kernel 2.6.34," Available from http://www.ohloh.net/p/linux/analyses/1226830.

[2] B. F. A. Chou and S. Hallem, "Linux kernel security report," Available from: http://www.coverity.com/html/library.php, 2005.

[3] "Analisys of the linux kernel," Available from: http://www.coverity.com/html/library.php, 2004.

[4] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, " An Empirical Study of Operating Systems Errors," in *Proceedings of the 18th Symposium on Operating Systems Principles*, 2001.

[5] "Ext2 fsd," Available from http://www.ext2fsd.com/.

[6] "Ext2 ifs," Available from http://www.fs-driver.org/.

[7] "Explore2fs," Available from http://www.chrysocome.net/explore2fs.

[8] "Haiku os," Available from http://www.haiku-os.org/.

[9] "ext2lib," Available from http://www.neonbox.org/ext2lib/ext2lib.html.

[10] "E2fsprogs: Ext2/3/4 filesystem utilities," Available from http://e2fsprogs.sourceforge.net/.

[11] "Winvfs," Available from http://winvfs.sourceforge.net/.

[12] "Rfsd," Available from http://www.acc.umu.se/~bosse/.

[13] A. M. G. Kroah-Hartman, J. Corbet, "Linux kernel development," Available from: http://www.linuxfoundation.org/publications/whowriteslinux.pdf, 2007.

[14] M. Johnson, "A tour of the linux vfs," Available from: http://tldp.org/LDP/khg/HyperNews/get/fs/vfstour.html, 1996.

[15] J. Dike, "A user-mode port of the linux kernel," in *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.

[16] A. Arcangeli, S. Labs, M. Cao, P. E. Mckenney, and D. Sarma, "Using read-copy-update techniques for system v," in *IPC in the Linux 2.5 kernel. In USENIX Annual Technical Conference, FREENIX Track*, 2003, pp. 297–309.

[17] "Apache portable runtime project," Available from http://apr.apache.org/.

[18] "Lkl based windows file system driver," Available from: http://github.com/lkl/lkl-win-fsd.

[19] "Simple lkl applications," Available from: http://github.com/lkl/lkl-proof-of-concept.

[20] "Complex network simulator using linux kernel library," Available from: http://github.com/lkl/lkl-net.

[21] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Wareld, "Xen and the art of virtualization," in *Proceedings of the 19th Symposium on Operating Systems Principles*, 2003.

[22] "Vmware," Available from http://www.vmware.com.

[23] "Kvm: Kernel-based virtualization driver," Whitepaper. Available from http://www.qumranet.com/wp/kvm_wp.pdf.

[24] D. Aloni, "Cooperative linux," in *Proceedings of the Linux Symposium*, 2004.

[25] A. Kantee, "Runnable userspace meta programs," Available from: http://www.NetBSD.org/docs/rump/.

[26] "Rump on non-netbsd operating systems," Available from: http://www.netbsd.org/~stacktic/rumpabroad.html.

[27] A. Ysmal and A. Kantee, "Fs-utils: File systems access tools for userland," in *EuroBSDCon*, 2009.

[28] A. Kantee, "Kernel development in userspace - the rump approach," in *BSDCan*, 2009.

[29] ——, "Environmental independence: Bsd kernel tcp/ip in userspace," in *AsiaBSDCon*, 2009.