The L4Ka::Pistachio Microkernel

white paper

System Architecture Group
University of Karlsruhe

April 29, 2024

1 L4Ka::Pistachio

L4Ka::Pistachio is the latest L4 microkernel developed by the System Architecture Group at the University of Karlsruhe. L4Ka::Pistachio is the first available kernel implementation of the L4 Version 4 kernel API, which is fully 32 and 64 bit clean, provides multiprocessor support, and superfast local IPC. It continues the L4 tradition of providing outstanding inter-process communication (IPC) performance, and a highly flexible kernel API.

The first release implements most of the core functionality of the API. The kernel is written in C++ with a significant focus on performance, portability, and reusability. L4Ka::Pistachio supports most existing mainstream hardware architectures, in particular Intel's IA32[1], and IA64[2], PowerPC 32bit, Alpha 21164, and MIPS R4000 and higher, with multiprocessor support for the first three architectures. In the near future additional support is planned for AMD64, Power4, ARM[3], and UltraSparc.

L4Ka::Pistachio provides a solid basis for research and development of highly specialized and general purpose operating systems for a wide variety of systems, ranging from tiny embedded devices to huge multiprocessor server systems. As an example, L4Ka::Linux, a modified Linux 2.4 kernel running on top of L4Ka::Pistachio, enables unmodified commodity applications to coexist with customized system components, which allows a smooth migration path towards customized services, and supports secure extensions to monolithic operating systems. This is a core requirement for upcoming consumer devices such as SmartPhones, secure payment, and digital rights management.

2 L4 Version 4 API

L4Ka::Pistachio implements the L4 Version 4 API (currently still referred to as eXperimental Version 2, or X2). The Version 4 API supersedes the seven year old

[1] Pentium and higher
[2] Itanium1 and SKI
[3] StrongARM and XScale

Version 2 API (V2), which was the basis for L4's success story. All L4 APIs are designed with the main focus on performance and flexibility. As described below, Version 4 introduces a set of new kernel features eliminating limitations experienced with V2.

2.1 Separation of API and ABI

L4 kernels have been implemented for multiple architectures, usually in an ad-hoc way, and often purely in assembly. Thus application binary interfaces (ABIs) were defined by the specific implementation, leading to different behavior and kernel features loosely following the initial IA32 reference implementation. As a result, system software was very specialized for one particular kernel implementation and therefore inherently non-portable.

Version 4 enforces a strict separation of API and ABI to support portable, architecture independent system software. The API specifies a set of high-level language bindings which are generic across all architectures.

2.2 Virtual Registers

Today's widely used processor architectures have very different register sets, featuring as few as seven general purpose registers on IA32 up to 128 integer and 128 floating point registers on IA64.

L4's IPC performance is achieved by strictly avoiding unnecessary memory references, thereby reducing cache and TLB footprint. However, having a single generic API for such very different architectures as IA32 and IA64 would have sacrificed performance for one or the other.

Version 4 therefore introduces the notion of a virtual register file per thread. Depending on processor characteristics, these registers map either to memory or to actual processor registers, allowing optimal use of hardware registers on all architectures.

2.3 Address Space and Threads

In Version 4 the fixed relation between address spaces and threads is removed. Any thread can be a member of any address space and threads can be transparently migrated between spaces. On 32 bit architectures up to 2^{18} and on 64 bit architectures up to 2^{32} concurrent threads are supported.

2.4 Kernel Interface Page

Rapidly increasing processor clock speeds require architectural extensions and modifications from one processor generation to the next. A recent example is the introduction of the SYSENTER/SYSEXIT instructions for IA32, which allow up to $10\times$ faster kernel entry and exit compared to the original software interrupt method. However, to utilize this new feature all legacy code has to be recompiled.

Version 4 introduces a kernel provided page, the kernel interface page (KIP), which contains function entry points for all system calls as well as frequently accessed system information. The invocation of a system call is now a simple function call to the entry point in the KIP. When new architectural features are added to next generation processors, these features can be directly utilized by replacing the page; old code will automatically use the latest processor features.

This methodology also allows for system calls which do not necessarily enter the kernel, e.g., the system clock or super-fast local IPC. The overhead of this flexible scheme is negligible when using an optimized dynamic linker which links system calls against the KIP's entry points.

2.5 SuperFast IPC

L4 provides a very limited yet extremely flexible and powerful set of abstraction and mechanisms: threads, address spaces, IPC, and mapping. All other primitives are built upon these four basic abstractions, including synchronization primitives.

Even though L4's IPC is extremely fast, it is still limited by the hardware architecture. The invocation of an IPC requires a change of the privilege mode, a costly operation on most hardware architectures. In many cases, however, IPC is used for synchronization and signaling of threads executing within the same address space. IPC can then be performed completely in user mode avoiding the overhead induced by the two unnecessary privilege level changes. First experiments have shown that it is possible to achieve an order of magnitude higher IPC performance.[4]

2.6 Multi-Processor Support

The Version 4 API has integral support for multi-processor systems, with its main focus on scalability. Load balancing and scheduling decisions are purely user-level based and thus support the implementation of arbitrary processor allocation policies.

A fundamental prerequisite for SMP scalability is the preservation of parallelism of user applications within the kernel, i.e., the kernel must not serialize operations of two parallel threads. The Version 4 API strictly follows this design rule by providing powerful orthogonal interfaces.

NUMA systems are supported with the aforementioned flexible thread allocation primitives. Thread migration allows use of node local memory and thus increases overall system performance. Similar optimizations are possible for simultaneous multi-threaded architectures, such as Intel's Hyper-Threading technology.

L4Ka::Pistachio's SMP support is experimental and not fully scalable yet due to a few coarse-grained locks, in particular in the memory subsystem. However, the critical IPC path is fully lock-free giving a good first performance indication. Multiprocessor support is available for IA32 and IA64 systems.

[4] In the current release L4Ka::Pistachio does not yet implement SuperFast IPC.

2.7 Interrupts

Similar to the Version 2 API, Version 4 abstracts system interrupts as kernel threads and interrupt delivery as IPC. Version 4, however, completely abstracts the first-level interrupt controller and only provides basic primitives for interrupt association. This allows for a higher level of parallelism, more efficient synchronization, and caching of interrupt controller state thereby increasing overall performance. Abstracting interrupt hardware was also a portability requirement, since some architectures allow access to interrupt controllers only in privileged mode.

2.8 Privileged Threads

Version 4 defines three privileged tasks: sigma0, sigma1, and the root task. All threads of a privileged task are allowed to execute certain system calls prohibited to non-privileged tasks. This scheme exports rights delegation completely to user level and supports implementation of arbitrary policies, such as access control lists, or capabilities using remote procedure calls (RPC).

3 Tools

Powerful development tools can significantly shorten the development cycle. Together with L4Ka::Pistachio we release L4Ka::IDL4, an IDL compiler which generates RPC stub code that is highly optimized for the kernel API and ABI, the hardware architecture, and the C/C++ compiler. L4Ka::IDL4's generated code quality is comparable to hand-optimized assembly stub code.

L4Ka::IDL4 not only supports the Version 4 API but also V2 and X.0 allowing a smooth migration path for existing software. It supports the CORBA and DCE syntax and has an integrated C++ parser which can parse C/C++ header files to, e.g., re-use type declarations.

The specification of Version 4's IPC interface is significantly influenced by and optimized for L4Ka::IDL4. Providing a large memory-based register file usually has a significant impact on the cache footprint of message transfers. However, using specifically optimized marshaling stubs the additional costs can be completely eliminated resulting in better overall performance.

RPC takes place as a three-step process. In the first step all parameters are marshaled into a message buffer. This message buffer is then transfered using the system's communication primitives and finally un-marshaled at the destination. The Version 2 API provides two message registers, X.0 up to three; messages exceeding this limit have to be transfered using a memory copy with a significant startup overhead. Version 4 provides up to 64 message registers with some of them backed by memory. The IDL compiler can marshal the parameters directly into these message registers and the kernel can transfer them from one address space to another with significantly less overhead. On the receiver's side the parameters can be directly used from the message register store.