The L4Ka Vision

Uwe Dannowski Kevin Elphinstone Jochen Liedtke Gerd Liefländer Espen Skoglund
Volkmar Uhlig Christian Ceelen Andreas Haeberlen Marcus Völp

University of Karlsruhe
System Architecture Group
76128 Karlsruhe, Germany
liedtke@ira.uka.de

Abstract

Microkernels are minimal but highly flexible kernels. Both conventional and non-classical operating systems can be built on top or adapted to run on top of them. Microkernel-based architectures should particularly support extensibility and customizability, robustness including reliability and fault tolerance, protection and security.

After desastrous results in the early 90's, the microkernel approach now seems to be promising, although it still bears a lot of research risks. Currently, University of Karlsruhe's system architecture group is defining the state of the art in microkernel research.

The *L4Ka* research project aims at substantiating and establishing a new methodology for system construction that helps to manage ever-increasing OS complexity and minimizes legacy dependence. Our vision is a microkernel technology that can be and is used advantageously for constructing any general or customized operating system including pervasive systems, deep-computing systems, and huge servers.

The technology should help to manage ever-increasing OS complexity, enable stepwise innovations in OS technology while preserving legacy compatibility, and lead to a widely-accepted foundation of system architecture.

1 Status Quo

Managing OS complexity is *the* challenge for OS technology. The complexity results from combining system requirements such as security, reliability, configurability, customizability, quality-of-service guarantees, and performance. The challenge becomes even harder due to (a) the ever-growing number of services and applications that increasingly interact with each other and (b) the increasing hardware complexity and parallelism.

OS research focuses therefore on techniques to structure systems dynamically, to isolate a system's components from each other, and to control interaction/communication between those components. Microkernels, extensible kernels, and sandboxing (e.g. through virtual machines) are relevant approaches in this area. Microkernels might have the broadest and highest impact on systems technology — if they fly.

1.1 Client Projects

Some projects already use *L4Ka* technology. Such *client projects* are extremely important and useful for *L4Ka*. As our "customers", they constantly evaluate our microkernels and also influence our research through their requirements. Closely related client projects that have strongly influenced preparatory *L4Ka* work are *SawMill*, *Drops*, and *Mungi.*

SawMill is a joint project between the IBM T. J.Watson Research Center in Yorktown Heights, NY, and the University of Karlsruhe. SawMill is mainly funded through IBM Research. *SawMill* aims at a general methodology for constructing highly-configurable component-based multi-server operating systems.

It complements the current *L4Ka* project: *SawMill* focuses on how to construct efficient servers and components on top of a microkernel. It does not only deal with construction from scratch but also tries to find principles how to take existing systems apart into components. (The first subproject is a multi-server Linux.) The project requires a robust, highly flexible, and extremely performant microkernel technology.

As such, it is a prominent customer of *L4Ka*. *SawMill* imposes practically relevant requirements on the *L4Ka* microkernels, and it gives us an ideal test case for our technology.

> "The Dresden Real-Time Operating Systems project *Drops* is a research project aiming at the support of applications with Quality-of-Service requirements. Although much research has been done on networking support for continuous-media applications, very few projects tackle related operating system issues, such as scheduling and file system support for bounded response time. The *Drops* project attempts to find design techniques for the construction of distributed real time operating systems whose every component guarantees a certain level of service to applications.
> A key component is L^4Linux, the Linux server on top of the L4 microkernel; it services standard Linux applications. In addition, separate real-time components, designed from scratch, provide deterministic service to real-time applications. At the moment, an ATM-based real-time protocol and a real-time file system are being developed." [Dre]

> "*Mungi* is an operating system based on the idea of a single address space, shared by all processes and processors in the system. The aim of the project is to show that such a single-address-space operating system (SASOS) can work on standard hardware, can be made as secure as traditional systems, is not inherently less efficient than traditional systems, and delivers performance advantages over traditional systems on some class of important applications.
> In addition, we are trying to build Mungi as a very pure SASOS, that is all data (even the systems') is in the single address space, and no other IPC mechanisms are supported by the OS. A 64-bit *Mungi* kernel has been developed. It is implemented on top of the L4 microkernel and currently runs on MIPS R4x00 platforms. Critical to efficient operation of *Mungi* is the implementation of its capability-based protection system, in particular the protection domain extension (PDX)

mechanism used to perform protected (cross-domain) procedure calls. Extensive caching of validation information is used to ensure that good performance." [Syd]

1.2 *L4Ka* Status Quo

The L4 microkernel for Intel's x86 processors is currently available in two independently developed versions:

"*Lemon Pip*" and "*Lime Pip*" Kernels. The original L4 assembler version for x86 ("*Lemon Pip*") has been developed 1995-98 at IBM Research and GMD. Based on our Joint Study Agreement with IBM Research, it was maintained and improved in Karlsruhe. The current version is highly efficient and stable. Most microkernel-based research and teaching activities are based on this version. We expect open-source availability in fall 2000.

A derivative of *Lemon Pip*, the "*Lime Pip*" kernel has been developed in Karlsruhe and currently serves as an experimental kernel in the *SawMill* project.

"*Hazelnut*" Kernels. For years, we were convinced that assembler-based microkernel implementations were necessary for achieving ultimate performance. High costs for maintaining, porting and adapting such microkernels are obvious disadvantages. A first C++ implementation of L4, Fiasco [Hoh], corroborated the claim that higher-level-language implementations offer substantially less performance. However, we started a new approach that might result a microkernel-implementation technology that combines higher-level-language benefits and performance:

Hazelnut kernel have been developed from scratch at Karlsruhe University over the past 10 months. Implementation languages are C++ and assembler. Currently, prototypes are available for x86 and ARM processors as Open Source under GPL since 09/00). Furthermore, IBM Research has started to adapt *Hazelnut* to Power PC 750.

Early performance measurements of *Hazelnut*/x86 look very promising. Exactly comparable microbenchmarks show IPC performance within a 10%-range of the original *Lime Pip* kernel. However, *Hazelnut* still lacks some important features, most notably the emulation of a tagged TLB on x86 which can improve IPC performance by a factor of 3 or more. (For sake of fairness, the above mentioned IPC comparison did not use that feature on the *Lime Pip*.) Implementing that necessary optimization in *Hazelnut* might thus lead to worse results. Another potential source of surprise may be the cache. Indeed, cache-miss cycles today dominate in many cases instruction-execution cycles. Microkernel costs are thus determined by both the raw execution time and the cache working set that is required for IPC or other primitives. The old *Lime Pip* kernel has an extremely small cache working set, about 2% of the L1 cache for an IPC (short message). We still do not know how small cache working sets can be achieved for *Hazelnut*. It depends on the additional features still to be implemented in *Hazelnut* and on to be explored code/data optimization techniques that can be combined with gcc.

If it turns out that we succeed in making the full *Hazelnut*/x86 perform comparable to the *Lime Pip* kernel, we will probably drop the *Pip* line and completely switch to *Hazelnut*. Otherwise, we have to include a very deep performance analysis comparing both versions in detail. Either we can thereafter fix the *Hazelnut* problems or have to keep the *Pip* line in parallel, and adapt it to all future kernel developments.

IDL^4. Interface-definition-languages (IDL) are well-known tools from Corba's world of middleware. However, IDL compilers,, e.g. Utah's Flick compiler [EFF+97], are also beneficial for intra-node communication,particularly client/server interaction.

In a component-based system, an IDL makes IPC (and RPC) mechanisms conveniently available for application-, component-, and server-writers. Through interface specifications, e.g. in CORBA [Obj] or DCOM [EE98] IDL, remote methods can be called as easy as local methods. The IDL compiler generates all necessary code for marshalling/unmarshalling parameters, method selection, etc. Surprisingly, we found that with increasing IPC performance, the IDL-generated stub code became a serious bottleneck. We had to invest in an experiment on generating sufficiently optimized stub code:

> "As IPC mechanisms become faster, stub-code efficiency becomes a performance issue for local client/server RPCs and inter-component communication. Inefficient and unnecessary complex marshalling code can almost double communication costs. We have developed an experimental new IDL compiler that produces near-optimal stub code for *gcc* and the L4 microkernel. The current experimental IDL^4 compiler cooperates with the *gcc* compiler and its x86 code generator. Other compilers or target machines would require different optimizations. In most cases, the generated stub code is approximately 3 times faster (and shorter) than the code generated by a commonly used portable IDL compiler. Benchmarks have shown that efficient stubs can increase application performance by more than 10 percent. The results are applied within IBM's *SawMill* project that aims at technology for constructing multi-server operating systems." [HLP+00]

Currently, IDL^4 is an experimental compiler. We plan to improve its flexibility and make it widely usable as a standard tool for the L4 API.

Real Time. Early work of the L4 group evaluated principle techniques of OS-based cache-partitioning to increase the predictability of real-time components in systems that run a mixture of real-time and non-real-time applications [LHH97]. Although most real-time dedicated work is done with our partner group at TU Dresden, the *L4Ka* group conducted some further research on real-time specific architectures, e.g. how multiprocessor systems could be used for real-time applications. The memory bus in such a system is a common resource for all processors and makes predictability and scheduling even harder than caches do:

> "Assume that we have 4 job mixes that can be correctly scheduled on 4 independent uniprocessors. What happens if we put those 4 job mixes on a 4-processor system with a single memory bus? Without any additional scheduling provisions, the shared memory bus can, in the worst case, stretch each schedule by a factor of 4. This is clearly unacceptable. In general, it would mean that the real-time capacity of an n-processor system is only $1/n$ of the capacity of a uniprocessor system. Multiprocessors would be unusable for real-time applications.
>
> Therefore, memory-bus scheduling is desirable. It should enable us to give soft and perhaps even hard guarantees in relation to memory bandwidth and latency to real-time applications. For non real-time applications, it should help optimize a system's overall throughput and/or latency.
>
> Work in this area seems to be rare. The author's are only aware of Frank Bellosa's work [Bel97]." [LVE00]

Persistence. Making data and perhaps even active programs persistent might largely simplify program construction. Data and programs should automatically survive shutdowns and crashes *without* that the programs explicitly write all their internal data into files all the time. Such operating systems have been around since decades [Lan92, Lie93, DdBF+94, SSF99]. The point that is extremely interesting to us is whether orthogonal persistence can be implemented easily and efficiently *on top* of the microkernel. In other words: is the microkernel sufficiently general and flexible that persistence does not require special kernel integration?

> "Orthogonal persistence opens up the possibility for a number of applications. We present an approach for easily enabling transparent orthogonal persistence, basically on top of a modern μ-kernel. Not only are all data objects made persistent. Threads and tasks are also treated as normal data objects, making the threads and tasks persistent between system restarts. As such, the system is fault surviving. Persistence is achieved by the means of a transparent checkpoint server running in user-level. The checkpoint server takes regular snapshots of all user-level memory in the system, and also of the thread control blocks inside the kernel. The execution of the checkpointing itself is completely transparent to the μ-kernel, and only a few recovery mechanisms need to be implemented inside the kernel in order to support checkpointing. During system recovery (after a crash or a controlled shutdown), the consistency of threads is assured by the fact that all their user-level state (user memory) and kernel-level state (thread control blocks) will reside in stable storage. All other kernel state in the system can be reconstructed either upon initial recovery, or by standard page fault mechanisms during runtime." [SL00]

Security. For a microkernel-based system that is built on interprocess communication, IPC control, i.e. communication interception, is a fundamental concept required to implement security servers and security policies.

Over the past years, we found that L4's original concept, *Clans & Chiefs* [Lie92], was not sufficiently flexible and efficient. The new *Elphinstone-Jaeger* redirection model [JEL+99] avoids most of the old model's problem (while being upward compatible to the Clans & Chiefs model).

Further research activities dealt with how to build security architectures, models, and policies based on our security primitives.

IA-64 In order to follow trends in hardware architecture development and broaden the impact area of microkernel systems, the L4 microkernel must support upcoming 64-bit architectures such as Intel's IA-64 and IBM's Power 4. IA-64 has been chosen as the main vehicle for this research, and current work has been purely theoretical, focusing on how to provide the right abstractions for dealing with EPIC, huge register sets, large amounts of memory, and multi-level caches.

Work is now entering a state where more experimental studies are needed to propel research any further. Awaiting the availability of real hardware, Intel's SoftSDV environment [UFG+99] is being used to create a preliminary version of the kernel—enabling accurate hardware timing measurements to be made without the underlying hardware.

2 Goals

The *L4Ka* project aims at substantiating and establishing a new methodology for system construction. Our vision is a microkernel technology that can be and is used advantageously for constructing any general or customized operating system.

The technology should help to manage ever-increasing OS complexity, enable stepwise innovations in OS technology while preserving legacy compatibility, and lead to a widely-accepted foundation of system architecture.

Strategic Goals

At first, we describe the strategic goals *why* we want to substantiate and to establish a microkernel technology.

"A Foundation To Manage OS Complexity." Ever-increasing complexity is a serious problem in OS construction. It might be *the* key challenge for the next decade.

Improved complexity management could have strong research and practical consequences. Research and industry could deal with more complex systems and/or improve system qualities. Furthermore, industry might reduce development and maintenance costs.

Our primary strategic goal, finding a usable way to manage complexity, dominates all further reasoning. Currently, the microkernel road looks most promising from this point of view. Nevertheless, our ideas of microkernels might substantially change on that path, perhaps even be replaced by new concepts beyond microkernels.

"Weakening The Legacy Dependence." Legacy software and legacy systems always form a "bed of *Prokrustes*" for research, development, and even customers. Innovative and incompatible new systems are hard to establish as research topics since such systems can not coexist with the "standard" (which will sooner or later become legacy). Far too often, innovative ideas can also not be explored because they would require to build an entirely new OS with full functionality more or less from scratch. The story continues at the industrial development and at the customer level: Legacy requirements make it impossible to develop or even use incompatible (sub)systems and/or services concurrently to legacy systems and services.

Our vision is a technology that substantially weakens this legacy dependence. OS research should more easily be able to build principally new systems and services while still using existing OS services. Typically, the new system/service would extend an existing OS by adding services, or modify it by replacing services, or coexist with the old OS. Microkernel architecture is not a sufficient precondition but probably a necessary one for such models.

"A System-Architecture Foundation." In contrast to hardware architecture and programming languages, we still do not have a general set of low-level basic concepts and paradigms in system architecture. Most existing concepts are either only available in theory, or they are of a too high level, or they are too much specialized. In particular, we are lacking concepts that orthogonally fit together and complement each other.

The bare availability of such a set of general basic paradigms would boost systems in science and engineering — provided the paradigms were implemented, performed nicely, would be flexible and powerful, and would be convenient to use.

Currently, the microkernel approach is one of very few approaches that might have the potential to lead to such a set of orthogonal, general, and practically available paradigms.

Tactical Goals

"WIDE APPLICABILITY." To substantiate and establish microkernel technology as a general basis for system construction, we must evaluate microkernels for all relevant classes of systems, applications, and hardware architectures. The goal is a technology that is as generally applicable as, e.g., processors, i.e. for almost all different types of systems and applications, from pervasive computing to deep computing (including classical applications and systems).

Consequently, we need to make microkernel technology available for (i) conventional workstations and PCs, (ii) pervasive systems including real-time systems, (iii) huge servers, (iv) and deep-computing systems including massively parallel systems and cluster systems. From application/system perspective, the "same" microkernel *(cum grano salis)*, i.e. the same API, should be used in all cases. Having also a common code base would be beneficial. However, its is still an open question to which degree a uniform implementation approach is possible. Different architectures might require different implementation methods.

Microkernels should be available for all relevant hardware architectures, including not only the basic 32-bit processor families but also upcoming 64-bit processor architectures, parallel architectures, and clusters. Only such a broad availability can ensure that microkernels are sufficiently general and offer a single, hardware-independent API. Also, availability on all major hardware platforms is necessary to convince a larger number of external research projects to use microkernel technology as a platform or a tool for their own research.

"OPEN BACKBONE PROJECT." To get broad acceptance in scientific and engineering communities and to come to a sufficiently broad evaluation, we need customers for our technology and we need to foster and support external experiments and research participation.

CLIENT PROJECTS. We have to ensure that we deal with practically relevant problems and optimizations and do not lose our focus. We must constantly evaluate *L4Ka* microkernels in the context of "real" systems and applications running on top. Closely related client projects such as *SawMill* and *Drops* are very effective in this context. They are "customers" of our technology and give us steady and quick feedback. More loosely related external research users should extend the set of client projects.

EASY ACCESS AND PARTICIPATION. All *L4Ka* technology must be easily accessible for everyone who wants to use it (open source). Also, the project must enable other research groups to participate in microkernel research.

STEADY STREAMLINING AND MAINTENANCE. Continuous streamlining is required to achieve our strategic goals and to meet our optimality requirements, *L4Ka* has to be steadily improved and extended. This includes ongoing efforts in validating, evaluating, improving, and minimizing fundamental kernel abstractions and concepts as well as simple maintenance. Correctness, performance, robustness, support for application and OSes have to be improved whenever possible. Maintenance complements streamlining and is necessary to support client projects and to improve global impact.

Technical Goals

"EVALUATING THE UPPER BOUNDS OF PERFORMANCE." For all classes of systems and applications, performance must be "close to infinity". Ideally, no adequately written program should be slower when running on a microkernel than when running on any other system architecture. Although this requirement can never be fulfilled in entirety and with full generality, we have to drive performance as far as possible. We operate on the architecture level and aim at wide applicability. We must therefore understand that performance of the primitives does not only influence current applications but also determines the applicability range, i.e. flexibility and generality, of the architecture. Primitives like IPC can in practice only be used as long as their relative costs are reasonable or, even better, negligible. Reducing costs from 200 to 20 cycles might then enable a whole new class of applications using the mentioned primitive.

Therefore, finding and evaluating the upper bounds of performance is the most crucial part to make the technology widely applicable.

Evaluating the upper bounds of performance is by far more than simply try to make a "fast" implementation. It requires performance analyses (*a priori* and *a posteriori*) that let us understand the reasons for the achieved performance, its limitations, and its dependencies from hardware and software architectural features and details. Sometimes, such analyses trigger inventive ideas that improve performance and invalidate the original analysis. Ideally, this road has to be followed as far as possible, practically only as far as reasonable. (Concluding when it becomes unreasonable is the hardest part.)

Since performance typically depends on many architectural features and even details of hardware, performance analyses and optimizations are necessary for all relevant classes of hardware platforms. This wide approach will most likely result in good synergetic effects.

x86. Intel's x86 family (and compatible processors from other vendors) form the *de facto* standard for PCs and workstations. An L4 implementation for x86 is a precondition for microkernel research: Many (if not most) OS research and development projects use this architecture, and it is absolutely dominant for classical systems and applications. *L4Ka*'s success relies to a a large degree on the existence and on the qualities of an x86 microkernel, particularly, because many external projects/parties will first evaluate the microkernel concepts on that basis. High performance and good stability of this implementation are essential.

ARM. Many pervasive systems are built on low-power architectures such as ARM processors. A corresponding microkernel is thus required to evaluate microkernel concepts for cell-phone OSes or other thin pervasive systems. Since the mentioned low-power platforms are all based on conventional 32-bit architectures with page-based memory management chances are high that the x86, ARM, and ... kernels can all be specializations of the same 32-bit microkernel.

IA-64 AND POWER4. Upcoming 64-bit architectures require a new internal kernel design to efficiently handle new processor architectures, such as EPIC and huge register sets, to handle caches of 3+ levels and Gigabyte capacity, and to ensure scalability of both physical and virtual memory. These architectures challenge microkernel API and implementation methodology. Corresponding microkernels are essential for evaluating the future firmness of microkernel technology.

MASSIVELY PARALLEL ARCHITECTURES. Clusters will probably not require special microkernel features or implementations. Massively parallel NUMA machines however will need

a special kernel implementation. For example, it has to include efficient handling of the interconnect, NUMA memory classes, and special cross-processor IPC implementations. In collaboration with the vendor of an MPP platform, such a microkernel should be developed, if possible based on an already existing microkernel for uniprocessors and SMPs.

"MAXIMIZING FLEXIBILITY AND GENERALITY." Effective widely applicable performance requires general concepts and primitives that are extremely flexible and support wide ranges of policies and customizations. For any system/application class, the microkernel mechanisms should enable adequate and efficient solutions.

From that perspective, flexibility and generality are higher-level performance principles ensuring performance of policies, algorithms, and applications based on the microkernel mechanisms. They have to be evaluated in a similar process as described for the performance-evaluation goal. Instead of analyzing mechanism performance depending on hardware architectures, we have to analyze policy performance depending on operating-system and application architectures.

Another dimension of flexibility is ease of modification, e.g., when extending, customizing, or configuring a system. Evaluating this dimension can only be done in collaboration with client projects such as *SawMill*.

CONVENTIONAL SYSTEMS. This is an obvious point, and we include it only for reasons of completeness. L^4Linux has shown the principle usability of the microkernel approach for classical operating systems. Of course, any new microkernel API and implementation has to be checked against L^4Linux.

PERVASIVE SYSTEMS. Ubiquitous or pervasive computing will be one of the key topics of the current decade. From the OS perspective, pervasive systems are to a certain degree evolutions from classical systems. For example, building systems highly configurable and based on components is crucial for pervasive systems (and nice to have for classical systems). Based on L4 microkernels, such fundamental research is, e.g., done in the *SawMill* project. However, pervasive systems also include new functionality that does not evolve from classical systems. Pervasive systems are, e.g., real-time systems, embedded systems, low-power systems, nomadic systems, wireless-connected systems. These functionality is partially evaluated in our client projects *SawMill* and *Drops*. Further external clients are necessary.

HUGE SERVERS. Huge servers impose very hard robustness requirements, e.g. 7x24 availability, on the underlying hardware and microkernel architecture. Performance problems are basically focused on memory management, cache management, and processor management. So far, we do not yet have an according client project.

SINGLE-ADDRESS-SPACE OS. Systems based on the paradigm of a single global address space and multiple protection domains impose untypical requirements on the virtual-memory system, e.g. efficient handling of sparse address spaces. On IA-64 processors, they could also benefit from MMU mechanisms that especially support protection domains. It is still an open question whether this MMU mechanisms can be efficiently used *without* extending the microkernel address-space paradigm and API. Client project in this context is UNSW's *Mungi*.

DEEP COMPUTING. Massively-parallel machines and clusters run very special operating systems and applications. For clusters, Karlsruhe's *Resh* will be our client project.

3 Work Plan

Research explores the unknown. Any workplan is thus more a snapshot of our current thinking than a plan that is likely to correctly predict the project future. Unexpected research results and insights will most likely modify the plan substantially. Nevertheless, it is useful to structure our research activities from our current point of view and from our current understanding. (However, nobody should read it in 2003.)

To structure the project, we group research activities under 5 headlines.

"GENERAL CONCEPTS AND ABSTRACTIONS" Basically, this group comprises all central and common scientific and engineering topics of *L4Ka*, such as general concepts, abstractions, mechanisms, methods, etc. It drives and triggers all other subprojects through the general concepts and API specifications it produces.

Key is to ensure that the concepts and APIs are orthogonal and complementary to each other, that they are general and flexible, and that they have an architectural performance[1] potential as high as possible.

"PERVASIVE KERNEL ARCHITECTURES" The goal is to deliver microkernel technology for pervasive systems. An important point is to design and construct microkernels and microkernel families for most relevant processors in this field, e.g. x86, ARM, SHx. Kernels for classical PCs and workstations — although not pervasive — are also included because pervasive kernels will most likely also be best-suited for PCs and workstations. (It will not pay to develop workstation-specialized kernels.)

Key criteria are performance and adaptability/portability. The tradeoff between both criteria should be minimized. Ideally, the highest possible performance should be combined with reasonable adaptability.

Furthermore, problems specific to pervasive systems are included, e.g. power control, real-time scheduling and cache control, SMP real-time support (bus scheduling), memory footprint, support for nomadic applications, wireless high-performance communication, etc.

"64-BIT KERNEL ARCHITECTURES" The focus are microkernels for upcoming 64-bit high-performance processor architectures, such as Intel's IA-64 and IBM's Power 4. We hope that the microkernel API will not need to be modified. However, to achieve the highest possible performance, the kernel implementation needs to be completely redesigned for those processors because of fundamentally new hardware features such as EPIC. Furthermore, activities dealing with problems specific to large 64-bit systems are included, e.g., multi-level cache control, memory-failure handling, virtualized processors, persistence, SASOS support, high 7x24 reliability.

"PARALLEL AND CLUSTER KERNEL ARCHITECTURES" The challenge is to make microkernel technology available for deep computing, i.e. for massively parallel systems and cluster systems. NUMA architectures, fast interconnects, and low-latency

[1] We use the term *architectural performance* as opposed to *implementation performance* to express the performance that the specified architecture permits for the best possible implementation. Architectural performance limits implementation performance.

cluster networks will require special kernel implementations. Perhaps, even conceptual API extensions might be needed. General goal is to achieve optimal performance with minimal conceptual kernel extensions (but probably very specialized kernel implementations).

"EVALUATION AND ANALYSIS" This headline comprises all activities that evaluate and analyze the systems that result from our *L4Ka* research activities. The evaluation is mainly based on macro and micro benchmarks; the analysis aims at understanding the benchmark results.

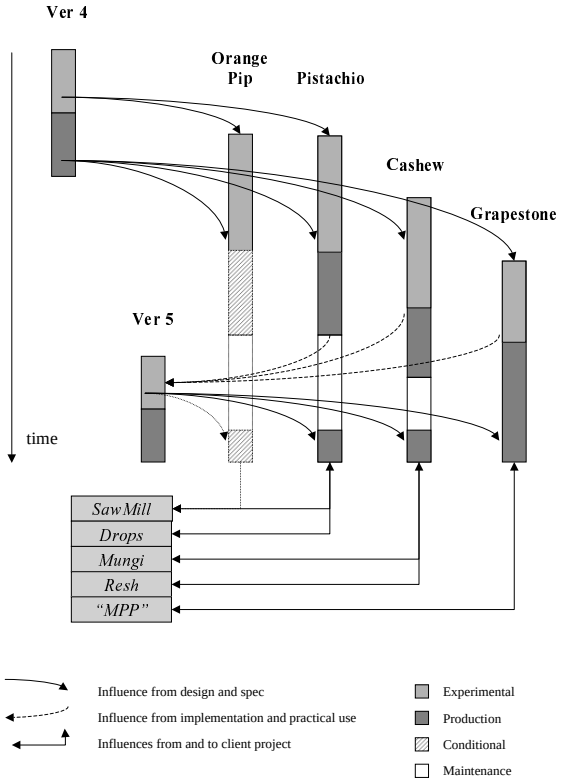

Figure 1: Main line activities.

Line Activities

Line activities for a longer time and typically result in a usable "research product", i.e. an implementation. Most line activities are very likely to happen and to be completed successfully. Their intended general outcome (target), e.g. a microkernel for IA-64 processors, is usually relative clear from the beginning, but specific properties and details will evolve over time. Typically, line activities are heavily influenced by bubble activities and incorporate their research results.

Figure 1 shows the main line activities and their interaction. The time axis expresses "before" and "after" relationships. However, the length of an activity bar is not proportional to the estimated time the activity needs. The figure also shows connections between activities and their primary client projects. Note that is relationship is active for the entire activity (although the connecting arrow starts from the end of the activity).

"GENERAL CONCEPTS AND ABSTRACTIONS"

Version 4 will be the next major revision and redesign of L4's API. Version 4 will incorporate research results and lessons we have learned in the past years. Besides purely technical changes, Version 4 will probably include relevant conceptual modifications. Currently, Version 4 is a moving target. We expect to have a first experimental definition at the end of this year and the final API definition in 2001.

Since "Version 4" is basically the synthesis and integration of many research results, engineering ideas, hardware-related technical solutions, and lessons learned over the past years, and since Version 4 is currently a moving target, we will not describe it in detail here. The interested reader can find detailed technical information in the current version of the L4 Version-4 Reference Manual which is attached to this application. Unfortunately, this document focuses on specification and does not contain much reasoning about "why" and "how to use". Those questions will be dealt with more deeply in the (to-be-written) System Programmer's Manual.

Version-5 Kernel API (*Gen/A)* In 2002 or 2003, lessons learned and new insights will result in an API revision. A sensitive topic for this revision may be to introduce threads as virtual objects (see bubble activity **??**. The address space, would then also include a task's view of the external tasks and threads; i.e., mapping would be the single controlling primitive in the system and also control IPC. Work on Version 4 will probably foster further sensitive topics for Version 5.

IDL^4 (*Gen/B)* The preliminary IDL^4 performance results are promising (see page 2). Remaining activities are —

- Extend the compiler to handle all data types.
- Add IDL^4 code generation for ARM processors.
- Stabilize the compiler so that it can be used as a standard tool.
- Make the compiler Version-4 compliant.
- Extend the IDL^4 compiler to generate also pure IPC code. This feature will enable to combine non-standard communication protocols, e.g. on the server side, with IDL interfaces.
- Evaluate whether and how IDL^4 code generation can be included in the Flick distribution. Include IDL^4 into Flick if possible with reasonable effort and without loss of performance.
- Support code-generator modifications for further processor architectures and target compilers. (Current target compiler is *gcc*.)

"PERVASIVE KERNEL ARCHITECTURES"

The *Hazelnut* kernels have been developed from scratch at Karlsruhe University over the past 10 months. Implementation languages are C++ and assembler. Currently, prototypes are available under GPL for x86 and ARM processors (see page 2).

So far, the *Hazelnut* prototypes do not yet include the emulation of tagged TLBs. Both x86 and ARM processors use untagged TLBs (Translation Lookaside Buffers). Therfore, any cross-address-space IPC has to flush the TLB which imposes substantial TLB-refill overhead on the subsequently executed code. To avoid that costs, the *Pip* kernels can emulate tagged TLBs for a limited number of address spaces. The mechanism is described in detail in [Lie95b] and [Lie95a]. This optimization is crucial for performance. For example, the TCP throughput of *SawMill*'s network stack increases by 25% on a Pentium 166 MHz through this optimization. Evidently, the mentioned technique must be included into *Hazelnut* to make it competitive, even though substantial internal modifications of the kernel are required.

A second point yet to be done is measuring and minimizing *Hazelnut*'s cache working sets. Effectively, a microkernel's cache working set is even more important for performance than its execution cycles. The goal for all performance-critical *Hazelnut* operations are cache working sets which are roughly comparable to those of *Lime Pip*, e.g. 2–3% of the L1 cache for short IPC.

Pistachio is the code name for a Version-4 compliant kernel that is based on the *Hazelnut* technology. It should replace the *Hazelnut* kernels on all processors.

Based on our past experiences and on the ideas that lead to Version 4, the *Pistachio* kernel will internally largely differ from *Hazelnut*. New internal structures and methods include partly user-accessible kernel thread-control blocks, superfast IPC (**??**), fine-granular timeouts (**??**), and improved real-time-scheduling support (**??**).

***Orange Pip* Kernel** (*Per/A*) *Orange Pip* is the code name for a Version-4 kernel for x86 processors. The kernel will be based on the existing *Pip* technology, i.e. developed processor specific, mainly in assembler.

The experimental *Orange Pip* kernel will serve as a vehicle for experimental implementation of new version-4 concepts and mechanisms. The activity is scheduled to run in parallel of the corresponding *Pistachio/x86* implementation for better and easier exploration of new mechanisms and concepts.

Although it seems to be surprising at a first glance, such low-level architectural experiments are sometimes easier in an environment that is not restricted by a compiler, its coding conventions, and code-generator properties. Architecturally relevant items such as cache-line usage, instruction parallelism, kernel-stack reduction, and controlling special hardware features strongly influence the achievable performance. Ease of modification is crucial in this context to find the best performing methods. Corresponding experimental implementations and evaluations often require less effort in an unrestricted assembler environment. Furthermore, avoiding to be biased by unknown or unwanted compiler/code-generator influences/optimizations is easier.

For the described exploration method, experimental and sometimes only partial implementations are sufficient. Once the optimal solution is identified, a complete and stable solution can be implemented in *Pistachio*. The experimental assembler solution serves as a guideline and defines the performance goal.

Fully-functional *Orange Pip* (*Per/A.1*) A fully functional and stable *Orange Pip* kernel will only be developed if the *Hazelnut* technology turns out not to perform sufficiently well or if fixing the *Hazelnut* problems would take too much time and thus delay the availability of a Version-4 kernel. We hope that a fully functional *Orange Pip* kernel will not be needed.

"64-BIT KERNEL ARCHITECTURES"

Cashew kernels aim at upcoming high-performance 64-bit processor architectures such as IA-64 and Power4. They will evolve as much as possible from *Pistachio* technology but will need substantially different internal algorithms. So far, we have identified the following basic new *Cashew*-related problems:

Intel's IA-64 platform is the first target for *Cashew*. Specific IPC-performance problems result from Intel's EPIC architecture with its large number of registers and its register stack engine. In short, the large number of registers contributes to a potentially massive context (more than 2KB) to be stored on each thread context switch. This added context switch overhead may prove fatal to microkernel systems. A combined hardware/software solution is therefore required to reduce the amount of information stored. Another solution might be to weaken the trust-relationship between certain threads, so that a thread might be allowed to read, but not modify, the register contents of other threads.

Other problems of the IA-64 architecture relate to providing policy free abstractions of the memory management hardware, so that an OS personality may harness the special hardware mechanisms which enables, e.g., a SASOS to be implemented efficiently.

"PARALLEL AND CLUSTER KERNEL ARCHITECTURES"

Version-4 kernel specialized for massively-parallel systems and clusters. Currently, we discuss two alternative approaches:

- a single microkernel that runs as one distributed kernel on all nodes, or
- a microkernel per node, complemented by a set of basic system servers.

So far, we favor the second approach. The basic servers then manage inter-node communication, inter-node scheduling and load balancing, cross-node address spaces, and cross-node memory access. Furthermore, they comprise special low-latency drivers for controlling the interconnect and/or specialized multi-gigabit inter-node networks.

We aim at building *Grapestone* kernels based on their corresponding *Cashew* kernels. Ideally, *Grapestone*/IA-64, e.g., should consist of the *Cashew/IA-64* kernel complemented by a *Grapestone* package. Once the first *Grapestone* kernel has been constructed successfully this way, we should be able to build *Grapestone* kernels at relatively low costs for any *Cashew* hardware platform.

We hope that we can basically use the same *Grapestone* kernel for cluster systems as well as for MPPs. Only some drivers, interconnect, etc., will be different. However, this hope is not yet substantiated.

References

[Bel97] F. Bellosa. Process cruise control: Throttling memory access in a soft real-time environment. Technical report, Dept. of Comp. Sci., University of Erlangen-Nürnberg, 1997. Available: http://www4.informatik.uni-erlangen.de/ELiTE/pub.html.

[DdBF+94] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindström, J. Rosenberg, and F. Vaughan. Grasshopper: an orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, Summer 1994.

[Dre] TU Dresden. Drops. http://os.inf.tu-dresden.de/drops.

[EE98] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, 1998.

[EFF+97] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstorm. Flick: A flexible, optimizing idl compiler. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, pages 44–56, June 1997.

[HLP+00] A. Haeberlen, J. Liedtke, Y. Park, V. Uhlig, and L. Reuther. Stub code performance is becoming important. In *First Workshop on Industrial Experiences with Systems Software (WIESS)*, San Diego, CA, October 2000. To appear.

[Hoh] M. Hohmuth. Fiasco. http: //os.inf.tu-dresden.de /fiasco /doc.html.

[JEL+99] T. Jaeger, K. Elphinstone, J. Liedtke, V. Panteleenko, and Y. Park. Flexible access control using ipc redirection. In *Hot Topics in Operating Systems (HotOS VII)*, Rio Rico, AZ, March 1999.

[Lan92] C. R. Landau. The checkpoint mechanism in KeyKOS. In *Proceedings of the 2nd International Workshop on Persistent Object Systmes (POS2)*, pages 24–25, Paris, France, September 1992.

[LHH97] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems (RTAS). In *3rd IEEE Real-time Technology and Applications Symposium*, pages 213–223, Montreal, May 1997.

[Lie92] J. Liedtke. Clans & chiefs. In *12. GI/ITG-Fachtagung Architektur von Rechensystemen*, pages 294–305, Kiel, March 1992. Springer.

[Lie93] J. Liedtke. A persistent system in real use – experiences of the first 13 years –. In *3^{rd} International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 2–11, Asheville, NC, December 1993.

[Lie95a] J. Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Arbeitspapiere der GMD No. 933, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1995.

[Lie95b] J. Liedtke. On μ-kernel construction. In *15^{th} ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.

[LVE00] J. Liedtke, M. Völp, and K. Elphinstone. Preliminary thoughts on memory-bus scheduling. In *9^{th} SIGOPS European Workshop*, pages 207–210, Kolding, Denmark, September 2000.

[Obj] The Object Management Group (OMG). *The Complete CORBA Services Book*. http: //www.omg.org /library /csindx.html.

[SL00] E. Skoglund and J. Liedtke. Transparent orthogonal checkpointing through user-level pagers. In *9^{th} International Workshop on Persistent Object systems (POS9)*, Lillehammer, Norway, September 2000. Springer LNCS. To appear.

[SSF99] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *17th ACM Symposium on Operating System Pronciples (SOSP)*, pages 12–15, Kiawah Island Resort, SC, December 1999.

[Syd] UNSW Sydney. Mungi. http://www.cse.unsw.edu.au/∼disy /Mungi/.

[UFG+99] R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, and H. Wang. SoftSDV: A presilicon software development environment for the IA-64 architecture. *Intel Technology Journal*, Q4, 1999.