THINK: A Software Framework for Component-based Operating System Kernels

Jean-Philippe Fassino
France Télécom R&D
jeanphilippe.fassino@francetelecom.com

Jean-Bernard Stefani
INRIA
jean-bernard.stefani@inria.fr

Julia Lawall
DIKU
julia@diku.dk

Gilles Muller
*INRIA**
gilles.muller@emn.fr

Abstract

Building a flexible kernel from components is a promising solution for supporting various embedded systems. The use of components encourages code re-use and reduces development time. Flexibility permits the system to be configured at various stages of the design, up to run time. In this paper, we propose a software framework, called THINK, for implementing operating system kernels from components of arbitrary sizes. A unique feature of THINK is that it provides a uniform and highly flexible *binding model* to help OS architects assemble operating system components in varied ways. An OS architect can build an OS kernel from components using THINK without being forced into a predefined kernel design (e.g. exo-kernel, micro-kernel or classical OS kernel). To evaluate the THINK framework, we have implemented KORTEX, a library of commonly used kernel components. We have used KORTEX to implement several kernels, including an L4-like micro-kernel, and kernels for an active network router, for the Kaffe Java virtual machine, and for a Doom game. Performance measurements show no degradation due to componentization and the systematic use of the binding framework, and that application-specific kernels can achieve speed-ups over standard general-purpose operating systems such as Linux.

1 Introduction

Embedded systems, such as low-end appliances and network routers, represent a rapidly growing domain of systems. This domain exhibits specific characteristics that impact OS design. First, such systems run one or only a small family of applications with specific needs. Second, for economic reasons, memory and CPU resources are scarce. Third, new hardware and software appear at a rapid rate to satisfy emerging needs. Finally, systems have to be flexible to support unanticipated needs such as monitoring and tuning.

Implementing an operating system kernel for such systems raises several constraints. Development time should be as short as possible; this encourages systematic *code re-use* and implementation of the kernel by assembling existing components. Kernel size should be minimal; only services and concepts required by the applications for which the kernel is targeted should be embbeded within the kernel. Efficiency should be targeted; no specific hardware feature or low-level kernel functionality should be masked to the application. Finally, to provide flexibility, it must be possible to instantiate a kernel configuration at boot time and to dynamically download a new component into the kernel. To support these features, it should be possible to resolve the bindings between components at run time.

Building flexible systems from components has been an active area of operating system research. Previous work includes micro-kernel architectures [2, 14, 25], where each component corresponds to a domain boundary (i.e. server), extensible systems such as SPIN [3] that support dynamic loading of components written in a type-safe language, and more recently the OSKit [7] or eCos [5] which allow the re-use of existing system components. One problem with the existing component-based approaches lies in the predefined and fixed ways components can interact and be bound together. While it is possible to create specific components to implement a particular form of binding between components, there are no supporting framework or tools to assist this development, and little help to understand how these different forms of binding can be used and combined in a consistent manner.

By *binding* we mean the end result of the general process of establishing communication or interaction channels between two or more objects. Bindings may take sev-

*Now at Ecole des Mines de Nantes

eral forms and range from simple pointers or in-address-space references, to complex distributed channels established between remote objects involving multiple layers of communication protocols, marshalling and unmarshalling, caching and consistency management, etc. As argued in [27], the range and forms of bindings are so varied that it is unlikely that a single generic binding process or binding type can be devised. This in turn calls for framework and tool support to help system developers design and implement specific forms of binding.

The attention to flexible binding is not new. Several works, e.g. [4, 10, 27], have proposed flexible binding models for distributed middleware platforms. The Nemesis operating system [13] introduces bindings for the handling of multimedia communication paths. The path abstraction in the Scout operating system [19] can be understood as an in-system binding abstraction. And channels in the NodeOS operating system interface for active routers [22] correspond to low-level packet-communication-oriented bindings. None of these works, however, has considered the use of a general model of component binding as a framework for building different operating system kernels.

This paper

This paper presents the THINK[1] framework for building operating system kernels from components of arbitrary sizes. Each entity in a THINK kernel is a component. Components can be bound together in different ways, including remotely, through the use of bindings. Bindings are themselves assemblies of components that implement communication paths between one or more components. This structuring extends to the interaction with the hardware, which is encapsulated in Hardware Abstraction Layer (HAL) components.

The contributions of this paper are as follows. We propose a software architecture that enables operating system kernels to be assembled, at boot-time or at run-time, from a library of kernel components of arbitrary size. The distinguishing feature of the framework is its flexible binding model that allows components to be bound and assembled in different and non-predefined ways.

We have designed and implemented KORTEX, a library of kernel components that offers several models of threads, memory management and process management services. KORTEX implements different forms of bindings, including basic forms such as system calls (syscalls), up-calls, signals, IPC and RPC calls. We have used KORTEX to implement L4-like kernel services. Our benchmarks show excellent performance for low-level system services, confirming that applying our component model and our binding model does not result in degraded performance compared to non component-based kernels.

[1] THINK stands for Think Is Not a Kernel

We have used KORTEX to implement operating system kernels for an active network router, the Kaffe Java virtual machine, and a Doom game. We have evaluated the performance of these kernels on a Macintosh/PowerPC machine. Our benchmarks show that our kernels are at least as efficient than the implementations of these applications on standard monolithic kernels. Additionally, our kernels achieve small foot-prints. Finally, although anecdotal, our experience in using the THINK framework and the KORTEX library suggests interesting benefits in reducing the implementation time of an operating system kernel.

The rest of the paper is structured as follows. Section 2 discusses related work on component-based kernels and OSes. Section 3 details the THINK software framework, its basic concepts, and its implementation. Section 4 describes the KORTEX library of THINK components. Section 5 presents several kernels that we have assembled to support specific applications and their evaluation. Section 6 assesses our results and concludes with future work.

2 Related Work

There have been several works in the past decade on flexible, extensible, and/or component-based operating system kernels. Most of these systems, however, be they research prototypes such as Choices and μ-Choices [31], SPIN [3], Aegis/Xok [6], VINO [26], Pebble [8], Nemesis [13], 2K [12], or commercial systems such as QNX [23], VxWorks [34], or eCos [5], still define a particular, fixed set of core functions on which all of the extensions or components rely, and which implies in general a particular design for the associated family of kernels (e.g. with a fixed task or thread model, address space model, interrupt handling model, or communication model). QNX and VxWorks provide optional modules that can be statically or dynamically linked to the operating system, but these modules rely on a basic kernel and are not designed according to a component-based approach. eCos supports the static configuration of components and packages of components into embedded operating systems but relies on a predefined basic kernel and does not provide dynamic reconfiguration capabilities.

In contrast, the THINK framework does not impose a particular kernel design on the OS architect, who is free to choose e.g. between an exo-kernel, a micro-kernel or a classical kernel design, a single or multiple address space design. In this respect, the THINK approach is similar to that of OSKit [7], which provides a collection of (relatively coarse-grained, COM-like) components implementing typical OS functionalities. The OSKit components have been used to implement several highly specialized OSes, such as implementations of the programming languages SML and Java at the hardware level [15]. OSKit components can be statically configured using the Knit tool [24]. The Knit compiler modifies the source code to replace calls across component boundaries by direct calls, thus enabling standard compiler optimizations. Unlike THINK however, OSKit does not provide a framework for binding components. As a result, much of the common structures which are provided by the THINK framework have to be hand-coded in an ad-hoc fashion, hampering composition and reuse. Besides, we have found in practice that OSKit components are much too coarse-grained for building small-footprint, specific kernels that impose no particular task, scheduling or memory management model on applications. Other differences between THINK and OSKit include:

- Component model: THINK has adopted a component model inspired by the standardized Open Distributed Processing Reference Model (ODP) [1], whereas OSKit has adopted Microsoft COM component model. While the two component models yield similar run-time structures, and impose as few constraints on component implementations, we believe that the THINK model, as described in section 3.1 below, provides more flexibility in dealing with heterogeneous environments.
- Legacy code: OSKit provides several libraries that encapsulate legacy code (e.g. from FreeBSD, Linux, and Mach) and has devoted more attention to issues surrounding the encapsulation of legacy code. In contrast, most components in the KORTEX library are native components, with the exception of device drivers. However, techniques similar to those used in OSKit (e.g. emulation of legacy environments in glue code) could be easily leveraged to incorporate in KORTEX coarse-grained legacy components.
- Specialized frameworks: in contrast to OSKit, the KORTEX library provides additional software frameworks to help structure kernel functionality, namely a resource management framework and a communication framework. The resource management framework is original, whereas the communication framework is inspired by the x-kernel [11].

Other operating system-level component-based frameworks include Click [18], Ensemble [15] and Scout [19]. These frameworks, however, are more specialized than THINK or OSKit: Click targets the construction of modular routers, Ensemble and Scout target the construction of communication protocol stacks.

We thus believe that THINK is unique in its introduction and systematic application of a flexible binding model for the design and implementation of component-based operating system kernels. The THINK component and binding models have been inspired by various works on distributed middleware, including the standardized ODP Reference Model [1], ANSA [10], and Jonathan [4]. In contrast to the latter works, THINK exploits flexible binding to build operating system kernels rather than user-level middleware libraries.

3 THINK Software Framework

The THINK software framework is built around a small set of concepts, that are systematically applied to build a system. These concepts are: *components*, *interfaces*, *bindings*, *names* and *domains*.

A THINK system, i.e. a system built using the THINK software framework, is composed of a set of domains. *Domains* correspond to resource, protection and isolation boundaries. An operating system kernel executing in privileged processor mode and a set of user processes executing in unprivileged processor mode are examples of domains. A domain comprises a set of components. Components interact through bindings that connect their interfaces. Domains and bindings can themselves be reified as components, and can be built by composing lower-level components. The syscall bindings and remote communication bindings described in section 4 are examples of composite bindings, i.e. bindings composed of lower-level components. Bindings can cross domain boundaries and bind together interfaces that reside in different domains. In particular, components that constitute a composite binding may belong to different domains. For example, the aforementioned syscall and remote communication bindings cross domain boundaries.

3.1 Core software framework

The concepts of *component* and *interface* in the THINK framework are close to the concepts of object and interface in ODP. A component is a run-time structure that encapsulates data and behavior. An interface is a named interaction point of a component, that can be of a server kind (i.e. operations can be invoked on it) or of a client kind (i.e. operations can be invoked from it). A component can have multiple interfaces. A component interacts with its environment, i.e. other components, only through its interfaces. All interfaces in THINK are strongly typed. In the current implementation of the THINK framework, interface types are defined using the Java language (see section 3.2). Assumptions about the interface type system are minimum: an interface type documents the signatures of a finite set of operations, each operation signature containing an operation name, a set of arguments, a set of associated results (including possible exceptions); the set of interface types forms a lattice, ordered by a subtype relation, allowing multiple inheritance between interface types. The strong typing of interfaces provides a first level of safety in the assembly of component configurations: a binding can only be created between components if their interfaces are type compatible (i.e. are subtypes of one another).

An interface in the THINK framework is designated by a *name*. Names are context-dependent, i.e. they are relative to a given *naming context*. A naming context encompasses a set of created names, a naming convention and a name allocation policy. Naming contexts can be organized in naming graphs. Nodes in a naming graph are naming contexts or other components. An edge in a naming graph is directed and links a naming context to a component interface (which can be another naming context). An edge in a naming graph is labelled by a name : the name, in the naming context that is the edge source, of the component interface that is the edge sink. Given a naming graph, a naming context and a component interface, the name of the component interface in the given naming context can be understood as a path in the naming graph leading from the naming context to the component interface. Naming graphs can have an arbitrary forms and need not be organized as trees, allowing new contexts to be added to a naming graph dynamically, and different naming conventions to coexist (a crucial requirement when dealing with highly heterogenous environments as may be the case with mobile devices).

Interaction between components is only possible once a *binding* has been established between some of their interfaces. A binding is a communication channel between two or more components. This notion covers both language-level bindings (e.g. associations between language symbols and memory addresses) as well as distributed system bindings (e.g. RPC or transactional bindings between clients and possibly replicated servers). In the THINK framework, bindings are created by special factory components called *binding factories*. A binding typically embodies communication resources and implements a particular communication semantics. Since several binding factories may coexist in a given THINK system, it is possible to interact with a component according to various communication semantics (e.g. local or remote; standard point-to-point at-most once operation invocation; component invocation with monitoring, with access control, with caching; event casting à la SPIN; etc). Importantly, bindings can be created either implicitly, e.g. as in standard distributed object systems such as Java RMI and CORBA where the establishment of a binding is hidden from the component using that binding, or explicitly, i.e. by invocation of a binding factory. Explicit bindings are required for certain classes of applications such as multimedia or real-time applications, that impose explicit, application-dependent quality of service constraints on bindings. Creating a binding explicitly results in the creation of a binding component, i.e. a component that reifies a binding. A binding component can in turn be monitored and controlled by other components.

```
interface Top { }
interface Name {
        NamingContext getNC();
        String toByte();
}
interface NamingContext {
        Name export(Top itf, char[] hint);
        Name byteToName(String name);
}
interface BindingFactory {
        Top bind(Name name, char[] hint);
}
```

Figure 1: Framework for interfaces, names and bindings

These concepts of naming and binding are manifested in the THINK software framework by the set of Java interface declarations shown in Figure 1. The type `Top` corresponds to the greatest element of the type lattice, i.e. all interface types are a subtype of `Top` (all interface types in THINK “extend” `Top`). The type `Name` is the supertype of all names in THINK. The operation `getNC` yields the naming context to which the name belongs (i.e. the naming context in which the name has been created through the `export` operation). The operation `toByte` yields a simple serialized form of the name instance.

The type `NamingContext` is the supertype of all naming contexts in THINK. The operation `export` creates a new name, which is associated to the interface passed as a parameter (the hint parameter can be used to pass additional information, such as type or binding data, required to create a valid name). As a side-effect, this operation may cause the creation of (part of) a binding with the newly named interface (e.g. creating a server socket in a standard distributed client-server setting). The operation `byteToName` returns a name, upon receipt of a serialized form for that name. This operation is guaranteed to work only with serialized forms of names previously exported from the same naming context. The type `NamingContext` sets minimal requirements for a naming context in the framework. More specific forms of naming contexts can be introduced if necessary as subtypes of `NamingContext` (e.g. adding a `resolve` operation to traverse a naming graph).

The type `BindingFactory` is the supertype of all binding factories in THINK. The operation `bind` creates a binding with the interface referenced by the name passed as a parameter (the hint parameter can be used to pass additional information required to establish the binding, e.g. type or quality of service information). Actual binding factories can typically add more specialized `bind` operations, e.g. adding parameters to characterize the quality of service required from the binding or returning specific interfaces for controlling the newly constructed binding.

3.2 Implementing the THINK framework

In our current prototype, THINK components are written in C for efficiency reasons. An interface is represented by an interface descriptor structure, whose actual size and content are unknown to the client, and which contains a pointer to the code implementing the interface operations, as shown, in figure 2. This layout is similar to a C++ virtual function table.

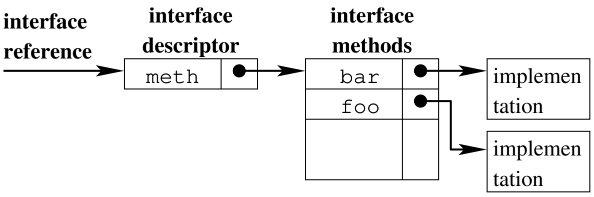

Figure 2: Run-time interface representation

The exact location of private component data is the responsiblity of the component developer. Depending on the nature of the target component, the implementation supports several optimizations of the structure of the interface representation. These optimizations help reduce, for instance, memory and allocation costs when handling interface descriptors. They are depicted in figure 3. If the component is a singleton, i.e. there is no other component in the given domain implementing the same interface, then the interface descriptor and the component private data can be statically allocated by the compiler. If the component is not a singleton but has only one interface, then the private data of the component can be allocated directly with the interface descriptor. Finally, in the general case, the interface descriptor is a dynamic structure containing a pointer to the interface operations and an additional pointer to the component's private data. In the component library described in section 4, most components are either singletons or have a single interface, and are implemented accordingly.

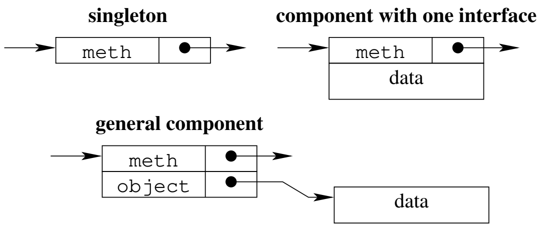

Figure 3: Optimization on interface representation

This implementation abides by C compiler ABI calling conventions [35]. Thus, arguments in a PowerPC implementation are passed on the stack and in registers to improve performance. It is important to notice that all calls to a component interface are expressed in the same way, regardless of the underlying binding type and the location of the component. For example, a server component in the local kernel domain is called in the same way as a server component in a remote host; only the binding changes[2].

3.3 Code generation and tools

Building a particular kernel or an application using the THINK framework is aided by two main off-line tools.

- An open interface compiler, that can be specialized

[2]This does not mean that the client code need not be prepared to handle the particular semantics associated with a binding, e.g. handling exceptions thrown by a remote binding component in case of communication failures.

to generate code from interface descriptions written in Java. For instance, it is used to generate C declarations and code that describe and produce interface descriptors, and to generate components (e.g. stub components) used by binding factories to create new bindings. This generated code can contain assembly code and exploit the specific features of the supporting hardware.

- An off-line configurator, that creates kernel images by assembling various component and binding libraries. This tool implicitly calls a linker (such as `ld`) and operates on a component graph specification, written in UML by kernel developpers, which documents dependencies between components and component libraries. Dependencies handled by the configurator correspond to classical functional component dependencies resulting from *provides* and *requires* declarations (*provides* means that a component supports an interface of the given type, *requires* mean that a component requires an interface of the given type to be present in its environment in order to correctly operate). An initialization scheduler, analogous to the OSKit's Knit tool [24], can be used to statically schedule component initialization (through calls to component constructors) at boot-time. The configurator also includes a visual tool to browse composition graphs.

Using the open interface compiler, interface descriptions written in Java are mapped onto C declarations, where Java types are mapped on C types. The set of C types which are the target of this mapping constitutes a subset of possible C signatures. However, we have not found this restriction to be an impediment[3] for developing the KORTEX library.

Code generation takes place in two steps. The first step compiles interface descriptions written in Java into C declarations and code for interface descriptors. These are then linked with component implementation code. Binding components are also generated from interface descriptions but use a specific interface compiler (typically, one per binding type), built using our open interface compiler. The second step assembles a kernel image in ELF binary format from the specification of a component graph.

During execution, a kernel can load a new component, using the KORTEX dynamic linker/loader, or start a new application, by using the KORTEX application loader.

[3]Note that, if necessary, it is always possible to specialize the open interface compiler to map designated Java interface types onto the required C types.

4 KORTEX, a component library

To simplify the development of operating system kernels and applications using THINK, we have designed a library of services that are commonly used in operating system construction. This library, called KORTEX, is currently targeted for Apple Power Macintoshes[4]. KORTEX currently comprises the following major components:

- HAL components for the PowerPC that reify exceptions and the memory management unit.
- HAL components that encapsulate the Power Macintosh hardware devices and their drivers, including the PCI bus, the programmable interrupt controller, the IDE disk controller, the Ethernet network card (mace, bmac, gmac and Tulip), and the graphic card (frame-buffer).
- Memory components implementing various memory models, such as paged and flat memory.
- Thread and scheduler components implementing various scheduler policies, such as cooperative, round-robin and priority-based.
- Network components, architected according to the x-kernel communication framework, including Ethernet, ARP, IP, UDP, TCP and SunRPC protocols.
- File system components implementing the VFS API, including ext2FS and NFS.
- Service components that implement a dynamic linker/loader, an application loader and a small trader.
- Interaction components that provide different types of bindings.
- Components implementing part of the Posix standard.

While many of these components are standard (for instance, the thread and memory components have been directly inspired by the L4 kernel [9]), several points about KORTEX are worth noting. First, KORTEX systematically exploits the core THINK framework presented above. In particular, KORTEX interaction components presented in section 4.5 all conform to the THINK binding model. The diversity of interaction semantics already available is a testimony to the versatility of this

[4]The choice of PowerPC-based machines may seem anecdotal, but a RISC machine does offer a more uniform environment for operating system design.

model. Second, KORTEX remains faithful to the overall THINK philosophy which is to not impose specific design choices to the OS architect. This is reflected in the fact that most KORTEX components are very fine-grained, including interaction components. For instance, syscall bindings (whose structure and semantics are typically completely fixed in other approaches) are built as binding components in KORTEX. Another example can be found with the HAL components in KORTEX, which strictly reflect the capabilities of the supporting hardware. Third, KORTEX provides additional optional frameworks to help OS architects assemble specific subsystems. KORTEX currently provides a resource management framework and a communication framework. The former is applied e.g. to implement the thread and scheduling components, while the latter is applied to implement remote bindings. Finally, we have strived in implementing KORTEX to minimize dependencies between components. While this is more a practical than a design issue, we have found in our experiments that fine-grained, highly independent components facilitate comprehension and reuse, while obviously yielding more efficient kernels, with smaller footprints. This is an advantage compared to the current OSKit library, for instance.

4.1 HAL components for the PowerPC

KORTEX provides HAL components for the PowerPC, including a HAL component for PowerPC exceptions and a HAL component for the PowerPC Memory Management Unit (MMU). The operations supported by these components are purely functional and do not modify the state of the processor, except on explicit demand.

The KORTEX HAL components manifest strictly the capabilities of the supporting hardware, and do not try to provide a first layer of portability as is the case, e.g. with μChoices' nano-kernel interface [31].

Exceptions

The PowerPC exceptions HAL component supports a single interface, which is shown in Table 4. The goal of this interface is to reify exceptions efficiently, without modifying their semantics. In particular, note that, on the PowerPC, processing of exceptions begins in supervisor mode with interrupts disabled, thus preventing recursive exceptions.

When an exception *id* occurs, the processor invokes one of the internal component methods $\texttt{TrapEnter}_{id}$. There is an instance of this method in each exception vector table entry. These methods first save the general registers, which form the *minimal execution context* of the processor, at a location previously specified by the system using the method `TrapSetContext`. This location is specified by both its virtual and physical addresses, because the Power PC exceptions HAL component is not aware of the memory model used by the system. $\texttt{TrapEnter}_{id}$ also installs a stack for use during the handling of the exception. A single stack is sufficient for exception handling because the processor disables interrupts during the handling of an exception. Next, $\texttt{TrapEnter}_{id}$ invokes the handler previously registered by the system using the method `TrapRegister`. When this handler finishes, the handler calls `TrapReturn` to restore the saved execution context. The cost for entering and returning from an exception on a PowerPC G4 running at 500 Mhz is shown on table 1.

```
interface Trap {
    void TrapRegister(int id, Handler handler);
    void TrapUnregister(int id);
    void TrapSetContext(int phyctx, Context virtctx);
    Context TrapGetContext();
    void TrapReturn();
}
```

Figure 4: Interface for PowerPC exception

Operation	instructions	time (μs)	cycles
$\texttt{TrapEnter}_{id}$	57	0.160	80
`TrapReturn`	48	0.110	55
total	105	0.270	135

Table 1: Cost for handling a exception

Although minimal, this interface provides enough functionality, e.g. to directly build a scheduler, as shown in section 4.4. The exceptions HAL component is completely independent of the thread model implemented by the system that uses its service.

Memory Management Unit

The PowerPC Memory Management Unit (MMU) HAL component implements the software part of the PowerPC MMU algorithm. This component can be omitted in appliances that need only flat memory. Table 5 shows the interface exported by this component.

```
interface MMU {
    void MMUsetpagetable(int virt,
                int phys, int sz);
    void MMUaddmapping(int vsid, int virt,
                int phys, int wimg, int pp);
    void MMUremovemapping(int vsid, int virt);
    PTE MMUgetmapping(int vsid, int virt);
    void MMUsetsegment(int vsid, int vbase);
    void MMUsetbat(int no, int virt, int phys,
                int size, int wimg, int pp);
    void MMUremovebat(int no);
}
```

Figure 5: Interface for PowerPC MMU

The `MMUsetpagetable` method is used to specify the location of the page table in memory. Since the PowerPC is a segmented machine, the `MMUsetsegment` method is used to set the sixteen 256 MB segments, thus providing a 4 GB virtual address space. The `MMUaddmapping`, `MMUremovemapping` and `MMUgetmapping` methods add, remove and obtain information about page translation.

The methods `MMUsetbat` and `MMUremovebat` reify the PowerPC Block Address Translation (BAT) registers. These registers provide a convenient way to build a single flat address space, such as can be used in low-end appliances. The two main benefits are speed of address translation and economy of page table memory use.

4.2 Resource management framework

KORTEX provides a resource management framework which can be applied to all resources in the system at various levels of abstraction. The framework comprises the resource and manager concepts as given in Figure 6. A resource manager controls lower-level resources and uses them to construct higher-level ones. New resources (e.g. threads) can be created through operation `create`, whereas resource allocation is effected through the `bind` operation which creates a binding to a given resource. In other words, a resource is allocated to a component when a binding has been created by the resource manager between the component and the resource. In this case, the hint parameter of the `bind` operation can contain managing information associated with the resource (e.g. scheduling parameters to be associated with a thread).

Several KORTEX components are architected according to the resource framework: threads and schedulers, memory and memory managers, network sessions (resources) and protocols (resource managers).

```
interface AbstractResource {
    void release();
}
interface ResourceManager extends BindingFactory {
    AbstractResource create(...);
}
```

Figure 6: Resource management framework

4.3 Memory management components

KORTEX provides memory management components that implement various memory models, such as paged memory and flat memory. A paged memory model can be used by systems that need multiple address spaces, for example to provide a process abstraction. The flat memory component can by used by systems that need only a kernel address space, as can be the case e.g. in low-end appliances. KORTEX also provides a component that implements the standard C allocator. Components implementing the two memory models and the allocator are described below.

The flat memory components implement a single kernel address space component that includes all of physical memory. This address space is provided by using `MMUsetbat` exported by MMU HAL (see Section 4.1). This component supports an address space interface providing methods to map and unmap memory in this address space. The implementation of this component is essentially void but the address space interface it supports is useful to provide a transparent access to memory for components, such as drivers, that need to map memory and that can operate similarly with either flat memory or paged memory.

Components providing the paged memory create, during initialization, a page table in memory and an address space for the kernel. An address space manager component provides an interface for creating new address space components. Address space components support interfaces of the same type as that of the flat memory address space component. Physical memory page allocation is provided by a standard buddy system component.

Finally, a dynamic memory allocator component provides the implementation of the standard GNU memory allocator.

4.4 Thread and scheduler components

KORTEX provides three preemptive schedulers that provide an same interface of the same type: a cooperative scheduler, a simple round-robin scheduler and a priority-based scheduler. They allow the usual operations on threads: creating and destroying threads, as well as allowing a thread to wait on a condition and to be notified. If threads are not running in the same address space, then the scheduler performs the necessary address space switch in addition to the thread context switch.

These schedulers are implemented using the PowerPC exceptions HAL component described in section 4.1. They can be implemented by simply installing a timer interrupt handler, in fact the PowerPC decrementer. On a decrementer exception, the handler uses `TrapSetContext` to replace the pointer to the execution context of the current thread with a pointer to the execution context of the newly scheduled thread. Due to the simplicity of the HAL, these schedulers can be very efficient. Table 2 presents context switching costs on a PowerPC G4 at 500 Mhz. For example, a context switch between two threads in the same address space costs 0.284 μs, and between two threads in different address spaces costs 0.394 μs. This permits the use of extremely small time slices, which can be useful e.g. for a real-time kernel.

	instructions	**time (μs)**	**cycles**
thread switch	111	0.284	142
process switch	147	0.394	197

Table 2: Context switching costs

4.5 Interaction components

KORTEX provides many different types of bindings between components, which may be localized in different domains (e.g. the kernel, an application, or a remote host).

Local binding

This binding type is the simplest form of binding and is used for interactions between components in the same domain. It is implemented by a simple pointer to an interface descriptor.

Syscall binding

This binding type can be used by systems that support multiple address spaces to provide application isolation. The syscall binding allows an application to use services provided by the kernel. A syscall binding is implemented using a client stub that performs a hardware syscall instruction `sc`, thus triggering an exception. The syscall trap handler then calls the target interface component. The application can pass up to seven arguments in registers (`r4` through `r10`) to the target. The remaining arguments, if any, must be passed in shared memory or on the user stack.

An optimization of the syscall binding can exploit the System V ABI specification calling conventions [35]. Registers (`r1`, `r14` to `r31`) are non volatile between method calls and it is not necessary to save them in the calling stub. Other registers (`r0`, `r3` to `r13`) are lost during method calls, and it is not necessary to save them either. Obviously this optimisation assumes that the ABI call conventions are obeyed. This optimization can save about 70 cycles per syscall.

Upcall and Signal binding

The upcall and signal bindings allow the kernel to interact with an application. A signal binding is used to propagate an exception to the currently running application, while an upcall binding is used to propagate an exception to an application running in a different address space than the current one. Upcall and signal bindings are very efficient because they merely invoke a dedicated handler in the application. The binding first updates the instruction and stack pointers, and then invokes the handler in the application using the special instruction `rfi`. The exception context is also propagated. This handler then calls the target component interface, which is designated by its memory address stored in the `r3` register.

Because the exception context is propagated, the upcall binding is not completely secure: an upcalled component may never return, thus monopolizing the processor. Several standard solutions can be used to build a secure upcall binding, for instance activating a timeout (unmasked prior to switching control to the upcalled address space) or using shared memory and a yield mechanism to implement a software interrupt.

Synchronous LRPC binding

An LRPC binding implements a simple synchronous interaction. It uses the syscall and upcall bindings. The syscall binding stub directly calls the upcall stub which calls the target application component interface.

Remote RPC binding

A remote binding implements a simple remote operation invocation protocol, which provides transparent access to components on a remote host. The binding directly builds and sends Ethernet packets, using the network protocol components. Although the binding is designed to work between kernels, it can support interaction between remote applications when combined with the syscall and upcall bindings.

5 Evaluation

In this section, we describe several experiments in assembling different operating system kernels using THINK. We have implemented a minimal extensible distributed micro-kernel, a dedicated kernel for an active router, one for a Java virtual machine, and another for running a DOOM game on a bare machine.

All measurements given in this paper are performed on Apple Power Macintoshes containing a PowerPC G4 running at 500Mhz (except for the PlanP experiment, which has been done on a PowerPC G4 at 350Mhz), with 1MB external cache and 128MB memory. Network cards used in our benchmarks are Asanté Fast PCI 100Mbps cards based on a Digital 21143 Tulip chip.

5.1 An extensible, distributed micro-kernel

We have built a minimal micro-kernel which uses L4 address space, and thread models. Instead of L4 IPC, we used KORTEX LRPC binding. The resulting kernel size is about 16KB, which can be compared with a 10KB to 15KB kernel size for L4 (note that L4 has been directly hand-coded in assembly language). Figure 7 depicts the component graph associated with this minimal micro-kernel. The figure shows the relationships between resources and resource managers, interfaces exported by components, as well as language bindings and local bindings used for combining the different components into a working kernel.

Table 3 summarizes the performance of synchronous bindings provided by the KORTEX library. Each call has a single argument, the `this` pointer, and returns an integer. An interaction via a local binding takes 6 cycles. This shows that a basic interaction between THINK components does not incur a significant penalty. The KORTEX syscall binding takes 150 cycles, which can be reduced to only 81 cycles when applying the optimisation described in section 4.5. By comparison, the Linux 2.4 syscall for implementing the `getpid` syscall takes 217 cycles.

Interaction	**instructions**	**time(μs)**	**cycles**
local	6	0.016	8
syscall	115	0.300	150
optimized syscall	50	0.162	81
signal	35	0.128	64
upcall	107	0.346	173
LRPC	217	0.630	315
optimized LRPC	152	0.490	245

Table 3: Performance of KORTEX bindings

Adding a dynamic loader component to this small micro-kernel yields a dynamically extensible kernel, although one without protection against faulty components and possible disruptions caused by the introduction of new components. The size of this extensible kernel is about 160KB with all components, including drivers and managers needed for loading code from a disk.

By adding remote RPC components to the extensible kernel, we obtain a minimal distributed system kernel, which can call previously exported resources located on remote hosts.

Table 4 shows the costs of interaction through our remote RPC binding. The table gives the time of completion of an operation invocation on a remote component, with null argument and an integer result. The measurements were taken with an Ethernet network at 10 Mbps and at 100 Mbps. A standard reference for low-latency RPC communication on a high speed network is the work done by Thekkath et al. [32]. Compared to the 25Mhz processor used in their test, a back of the envelope computation[5] would indicate that our results are on a par with this earlier work[6]. Furthermore, the KOR-

[5] $(500/25)*(11.3+4) = 306$ microsecond at 10Mbps to compare with 296 microsecond found in [32].

[6] Especially since the breakdown of the costs is consistent with

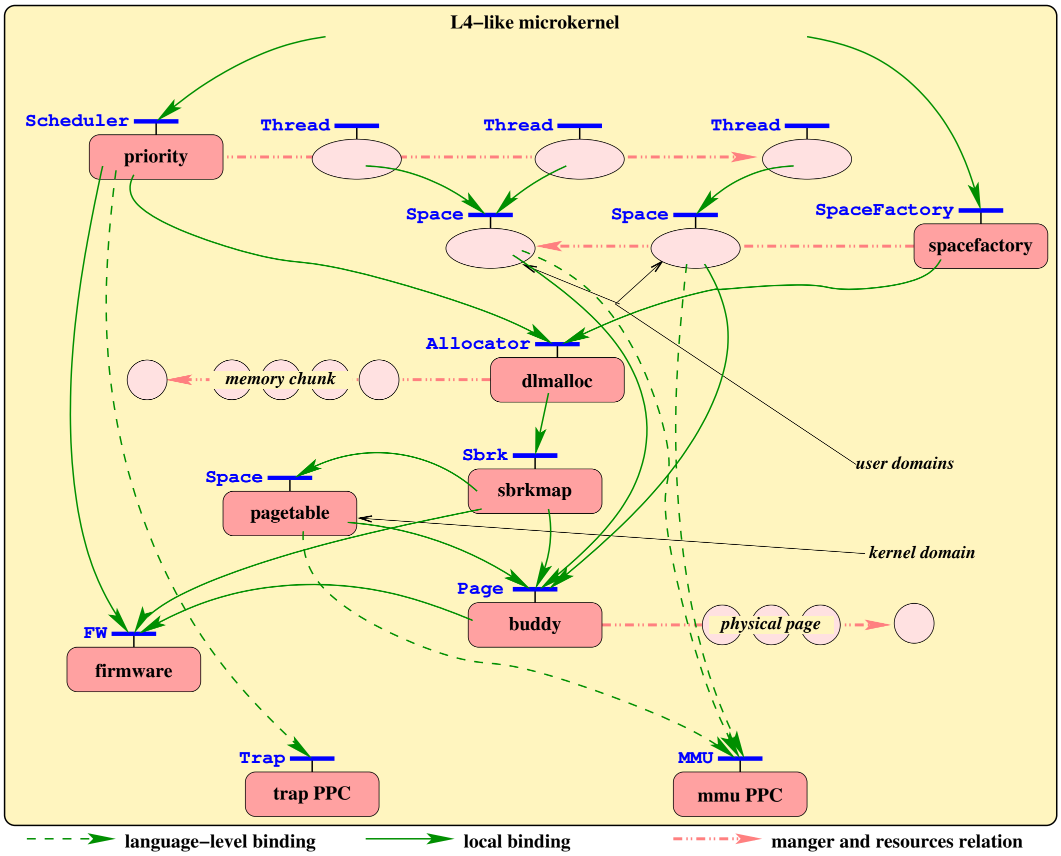

Figure 7: An example kernel configuration graph

TEX remote RPC binding can be compared with various ORBs such as Java RMI. Here, the 40 microsecond synchronous interaction performance (even adding the costs of syscalls at both sites) should be compared with the typical 1 millisecond cost of a similar synchronous interaction.

Network type	total	Time (μs) network link	network driver	marshall. +null call
10baseT	180	164.7 (91.5%)	11.3 (6.3%)	4 (2.2%)
100baseT	40	24.7 (61.7%)	11.3 (28.3%)	4 (10%)

Table 4: Performance of synchronous remote binding

These figures tend to validate the fact that the THINK framework does not preclude efficiency and can be used to build flexible, yet efficient kernels.

those reported in [32].

5.2 PlanP

PlanP [33] is a language for programming active network routers and bridges, which has been initially prototyped as an in-kernel Solaris module, and later ported to the Linux operating system (also as an in-kernel module). PlanP permits protocols to be expressed concisely in a high-level language, yet be implemented efficiently using a JIT compiler. While PlanP programs are somewhat slower than comparable hand-coded C implementations, a network intensive program such as an Ethernet learning bridge has the same bandwidth in PlanP as in the equivalent C program. This suggests that the Solaris and Linux kernels must be performance bottlenecks.[7]

To show that we can get rid of this bottleneck in a THINK system, we took as an example a learning bridge protocol `Plearn`, programmed in PlanP, and we measured throughput on Solaris, Linux and a dedicated kernel built with KORTEX. The configurations used in our four ex-

[7]PlanP runs in the kernel in supervisor mode; there is no copy of packets due to crossing kernel/user domain boundaries.

periments were as follows. In all experiments, the hosts were connected via a 100Mbps Ethernet network, and the two client hosts were Apple Power Macintoshes containing a 500Mhz PowerPC G4 with 256Mb of main memory and running the Linux 2.2.18 operating system. In the first experiment we measured the throughput obtained with a null bridge, i.e. a direct connection between the two client hosts. In the second experiment, the bridge host was a 167Mhz Sun Ultra 1 Model 170s with 128Mb of main memory running Solaris 5.5. In the third experiment, the bridge host was an Apple Power Macintosh G4 350Mhz with 128Mb of main memory running Linux 2.2.18. In the fourth experiment, the bridge host was the same machine as in the third experiment but running KORTEX. Throughput was measured using `ttcp` running on client hosts. Table 5 shows the throughput of the `Plearn` PlanP program running on Solaris, Linux and KORTEX. As we can see, using the KORTEX dedicated kernel increased the throughput more than 30% compared to Linux (from 65.5MBps for Linux to 87.6Mbps for KORTEX).

bridge	**throughput**
none	91.6Mbps
PlanP/Solaris, Sparc 166Mhz	42.0Mbps
PlanP/Linux, PowerPC 350Mhz	65.5Mbps
PlanP/KORTEX, PowerPC 350Mhz	87.6Mbps

Table 5: Performance of the THINK implementation versus Solaris and Linux implementation

5.3 Kaffe

Kaffe is a complete, fully compliant open source Java environment. The Kaffe virtual machine was designed with portability and scalability in mind. It requires threading, memory management, native method interfacing and native system calls. Kaffe was ported to a dedicated THINK kernel by mapping all system dependencies to KORTEX components. For example, exception management makes direct use of the exceptions HAL component, whereas preemptive threads have been implemented on both the priority-based scheduler, which provides a native thread like semantics, and the cooperative scheduler which provides a Java thread like semantics. Thanks to our binding and component framework, making this change requires no modification in the threading code. Table 6 compares the performance of Kaffe when running on Linux and when running on our dedicated kernel. As we can see, exception management is better on the dedicated kernel due to the reduced cost of signals, whereas native threads perform as well as Java threads.

When porting a JVM, most of the time is spent in adapting native methods. Thanks to the reuse of KORTEX components, implementing the Kaffe dedicated kernel took one week.

When executing standard Java applications with small memory needs, the memory footprint is 125KB for KORTEX components, plus 475KB for Kaffe virtual machine, plus 1MB for bytecode and dynamic memory, for a total of 1.6MB.

5.4 Doom

An interesting experiment is to build a dedicated kernel that runs a video game (simulating e.g. the situation in a low-end appliance). To this end, we have ported the Linux Doom, version LxDoom [16], to THINK, using the KORTEX flat memory component. The port took two days, which mainly consisted in understanding the graphic driver. The memory footprint for this kernel is only 95KB for KORTEX components, 900KB for the Doom engine and 5MB for the game scenario (the WAD file).

The THINK implementation is between 3% and 6% faster than the same engine directly drawing on the frame-buffer and running on Linux starting in single user mode, as shown in table 7. Since there are no system calls during the test, and the game performs only computation and memory copy, the difference is due to residual daemon activity in Linux and to the use of the flat memory which avoids the use of the MMU. To pinpoint the cost of the latter, we have built the same application by simply switching to the use of the KORTEX paged memory management component. As we can see, the use of the MMU adds about 2% on the global execution time. While the performance benefits are barely significant in this particular case, this scenario illustrates the potential benefits of the THINK approach in rapidly building optimized, dedicated operating system kernels.

	external resolution		
	320x200	640x480	1024x768
KORTEX(flat)	1955	491	177
KORTEX(MMU)	1914	485	171
Linux	1894	483	167

Table 7: Doom frames per second

Benchmark	Kaffe/Linux (java-thread)	Kaffe/KORTEX (java-thread)	Kaffe/KORTEX (native-thread)
synchronized(o) { }	0,527 μs	0,363 μs	0,363 μs
try { } catch(...) { }	1,790 μs	1,585 μs	1,594 μs
try {null.x()} catch(...) { }	12,031 μs	5,094 μs	5,059 μs
try {throw} catch(...) { }	3,441 μs	2,448 μs	2,434 μs
Thread.yield()	6,960 μs	6,042 μs	6,258 μs

Table 6: Evaluation of the Kaffe dedicated THINK kernel

6 Assessment and Future Work

We have presented a software framework for building flexible operating system kernels from fine-grained components and it associated tools, including a library of commonly used kernel components. We have evaluated our approach on a PowerPC architecture by implementing components providing services functionally similar to those implemented in the L4 kernel, and by assembling specific kernels for several applications: an active network router, a Java virtual machine, and a Doom game. The micro-benchmarks (e.g. context switching costs and binding costs) of our component-based microkernel show a level of performance that indicates that, thanks to our flexible binding model, building an operating system kernel out of components need not suffer from performance penalties. The application benchmarks for our example dedicated kernels show improved performances compared to monolithic kernels, together with smaller footprints. We have also found that developing specific operating system kernels can be done reasonably fast, thanks to our framework, component library, and tools, although our evidence in this area remains purely anecdotal.

This encourages us to pursue our investigations with THINK. In particular, the following seem worth pursuing:

- Investigating reconfiguration functions to support run-time changes in bindings and components at different levels in a kernel while maintaining the overall integrity of the system.
- Investigating program specialisation techniques to further improve performance following examples of Ensemble [15] and Tempo [20].
- Developing other HAL components, in particular for low-end appliances (e.g. PDAs), as well as ARM-based and Intel-based machines.
- Developing a real-time OS component library and exploiting it for the construction of an operating system kernel dedicated to the execution of synchronous programming languages such as Esterel or Lustre.
- Exploiting existing OS libraries, such as OSKit, and their tools, to enhance the KORTEX library and provide a more complete development environment.

Availability

The KORTEX source code is available free of charge for research purposes from the first two authors.

Acknowledgments

Many thanks to our shepherd, F. Bellosa, and to anonymous reviewers for their comments. S. Krakowiak helped us tremendously improve on an earlier version of this paper.

This work has been partially supported by the French RNRT Project Phenix.

References

[1] ITU-T Recommendation X.903 | ISO/IEC International Standard 10746-3. *ODP Reference Model: Architecture*. ITU-T | ISO/IEC, 1995.

[2] M. Acetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for unix development. In *Proceedings of the USENIX Summer 1986 Conference*, pages 93–112, 1986.

[3] B.N. Bershad, S. Savage, P. Pardyak, E. Gün Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Ex-

tensibility, safety and performance in the SPIN operating system. In SOSP'95 [28], pages 267–283.

[4] B. Dumant, F. Horn, F. Dang Tran, and J.B. Stefani. Jonathan: an Open Distributed Processing Environment in Java. In Middleware'98 [17].

[5] eCos. http://sources.redhat.com/ecos/.

[6] D.R. Engler, M.F. Kaashoek, and J.W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In SOSP'95 [28], pages 251–266.

[7] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In SOSP'97 [29], pages 38–51.

[8] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The pebble component-based operating system. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 267–282, Monterey, CA, USA, June 1999.

[9] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-Kernel-based systems. In SOSP'97 [29], pages 66–77.

[10] R. Hayton, M. Bursell, D. Donaldson, and A. Herbert. Mobile Java Objects. In Middleware'98 [17].

[11] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[12] F. Kon, R.H. Campbell, M.D. Mickunas, K. Nahrstedt, and F.J. Ballesteros. 2K: a distributed operating system for heterogeneous environments. Technical Report UIUCDCS-R-99-2132, University of Illinois, Dec 1999.

[13] I. Leslie, D. McAuley, R. Black, Timothy Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE JSAC*, 14(7), 1997.

[14] J. Liedtke. On μ-kernel construction. In SOSP'95 [28], pages 237–250.

[15] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building reliable, high-performance communication systems from components. In SOSP'99 [30].

[16] Lxdoom. http://prboom.sourceforge.net/.

[17] *Proceedings of the 1998 IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Lake District, UK, September 1998.

[18] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. In SOSP'99 [30].

[19] D. Mosberger and L. Peterson. Making Paths Explicit in the Scout Operating System. In OSDI'1996 [21].

[20] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 240–249, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[21] *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.

[22] L. Peterson, Y. Gottlieb, M. Hibler, P. Tullmann, J. Lepreau, S. Schwab, H. Dandekar, A. Purtell, and J. Hartman. An OS Interface for Active Routers. *IEEE Journal on Selected Areas in Communications*, 19(3):473–487, 2001.

[23] QNX Software Systems Ltd, Kanata, Ontario, Canada. http://www.qnx.com.

[24] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, California, October 2000.

[25] V. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, April 1992.

[26] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In OSDI'1996 [21], pages 213–227.

[27] M. Shapiro. A binding protocol for distributed shared objects. In *Proceedings of the 14th International Conference on Distributed Computer Systems (ICDCS)*, Poznan, Poland, June 1994.

[28] *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, USA, December 1995.

[29] *Proceedings of the 1997 ACM Symposium on Operating Systems Principles*, St-Malo, France, October 1997.

[30] *Proceedings of the 1999 ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999.

[31] S-T. Tan, D. K. Raila, and R. H. Campbell. A Case for Nano-Kernels. Technical report, University of Illinois at Urbana-Champaign, Department of Computer Science, August 1995.

[32] C. A. Thekkath and H. M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, 1993.

[33] S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *17th IEEE Symposium on Reliable Distributed Systems*, pages 135–143, West Lafayette, Indiana, October 1998.

[34] Wind River Systems Inc, Alameda, CA, USA. http://www.windriver.com.

[35] S. Zucker and K. Karhi. System V Application Binary Interface: PowerPC Processor Supplement. Technical report, SunSoft, IBM, 1995.