Exokernels, Protocol implementation and Erlang

Björn Knutsson* (Bjorn.Knutsson@DoCS.UU.SE)

Per Gunningberg* (Per.Gunningberg@DoCS.UU.SE)

January, 1999

Technical Report DoCS 103

Uppsala University, Dept. of Computer Systems,
Box 325, S-751 05 Uppsala, Sweden.

Abstract

In this report, we will discuss how ideas in the experimental operating system architectures called exokernels can be used to improve performance of the Erlang systems and programs written in Erlang, with an emphasis on protocol implementations written in Erlang. We will present experiences and conclusions.

1 Introduction

The current "breed" of traditional operating systems abstract and multiplex hardware resources in a way that will work for almost all possible applications. Often portability is the objective — they try to provide abstractions that are not specific to a certain implementation of e.g. hard disks or network cards. They typically do not stop there, they also implement additional general abstractions such as file systems, virtual memory systems and protocol stacks. The abstractions provided by an OS are often integrated with safe multiplexing of the hardware resources in a such way that the abstractions cannot be circumvented without losing the safe multiplexing. Safety is an important issue for operating systems because we do not want one application to be able crash others, or indeed the whole system.

The generality of traditional OSs means that operating systems tend to abstract away the underlying hardware semantics, effectively hiding them from the applications. This has the great advantage that applications become portable across a wide array of different hardware with different underlying semantics. It does, however, also have the unfortunate disadvantage of making performance optimized implementations of applications hard and/or expensive to implement.

The abstractions provided are also a package deal, we cannot pick and choose the parts we want or need, instead we must either accept the whole package, or start from scratch with little or no help from the kernel. Furthermore, the solutions that are open once we reject the abstraction package often have unwanted side effects.

A new approach in operating systems architectures are the Exokernels[13, 25]. They attempt to solve the problems with abstractions hindering optimizations by reducing or eliminating OS abstractions not explicitly needed for safe multiplexing of the underlying hardware. An exokernel will export all information that is safe to export to the application, and also allow applications to directly administer its own resources and tailor its own abstractions. This means that OS abstractions will be implemented in user space either as part of the application or, more commonly, as a separate library, a *library operating system*.

Three implementations of exokernels are known at this time: Aegis[13], created for the MIPS R3000 processor (Digital DECstation), Glaze[27], created for the Fugu multiprocessor (MIT research

*Supported by Ericsson CSLAB/SARC

processor), and XOK[25], created for the Intel x86 processor family (Pentium, Pentium Pro and Pentium II based PC-compatibles).

It is our belief that for exokernels to be viable, it must be shown that exokernels will not incur a penalty for non-specialized applications, will deliver on the promises of better performance for specialized applications and finally, and perhaps most important, it must be shown that is is possible to improve the performance of existing applications without requiring a redesign and rewrite, i.e. to run on a library OS implementing the API of a traditional OS.

The aim of our research is to determine to what extent exokernel ideas are useful in a few typical and representative applications. Some applications have already been explored by the MIT Exokernel team — the Cheetah (see section 3.2) web server implementation have shown that exploiting domain specific knowledge can substantially improve performance for a resource intensive application.

To investigate the effects of moving a large system to an exokernel, and also to get an interesting platform for further experiments, we have ported an implementation of the concurrent functional language Erlang[5] from UNIX to the library OS ExOS[13, 25, 33] running on the exokernel XOK[25]. We have focussed our efforts on how to enable efficient protocol implementations written in Erlang.

Erlang combines the best and worst of scenarios for an exokernel: On one hand, any performance gain achieved on the Erlang system by exploiting exokernel features will benefit all programs written in Erlang. On the other hand, we should not make changes to the Erlang implementation that are not easily back-ported or maintained in the mainstream version, neither can we modify the semantics of the language to accommodate a more efficient implementation.

The organization of this report is as follows: In section 1 an introduction to the problems is presented. In section 2, the basics of exokernels are presented. In section 3, some of the experiments of the MIT exokernel team are presented. We discuss their results and draw some conclusions. We also present one of our early experiments and our conclusions. Section 4 presents how we can exploit knowledge of resource utilization to improve applications, and specifically the Erlang system. We present a strategy for improvement, and some specific ideas for improvements that we believe would be beneficial for Erlang and protocol implementations written in Erlang. Finally, in section 5 we present our results and conclusions, and a road map how work to improve Erlang should proceed.

2 Exokernels

The exokernel operating system architecture is based on the idea of not providing any abstractions not needed for safe multiplexing of hardware resources. All other abstractions must be implemented in user space.

Basically, an exokernel can be said to provide low-level device drivers and low-level multiplexing of hardware resources, such as scheduling of the processor, low-level memory and disk allocation, network multiplexing etc. Along with multiplexing, it also provides access control for resources. The functionality provided by the exokernel can then be used to build traditional operating system functionality such as communication protocol stacks, file systems, virtual memory systems etc.

Removing standard OS abstractions may seem like a step backwards to pre-OS days, but actually the most useful part is kept: safe multiplexing. Just as applications today are linked with runtime libraries, they could also be linked with an operating system module. This will allow the applications to retain the benefits of the traditional abstractions — wide source portability across different platforms, while still allowing flexibility in modifying abstractions in the operating systems without affecting other applications.

ExOS[13, 25, 33] is the default library operating system running on top of the exokernels Aegis and XOK. ExOS implements most of the functionality and API of a BSD 4.4 UNIX system. Many UNIX applications can be compiled and linked with ExOS with no more changes than when moving them between UNIX dialects.

Linking operating system modules to the applications has the advantage of allowing the programmer to modify or replace parts of the OS modules — the library operating system — without affecting other applications. It also means that an application programmer can choose a pre-made library operating system, like ExOS, which has implementations of OS abstraction that fits her application. Functions in the library OS can then be modified or overridden with specialized implementations of performance critical abstractions.

Many ideas of Exokernels can be traced to the micro kernel architectures such as Mach[19][36, Chapter 20]. Basically Exokernels are the micro kernels taken one step further towards less abstractions and more responsibility for user level code to manage the hardware resources.

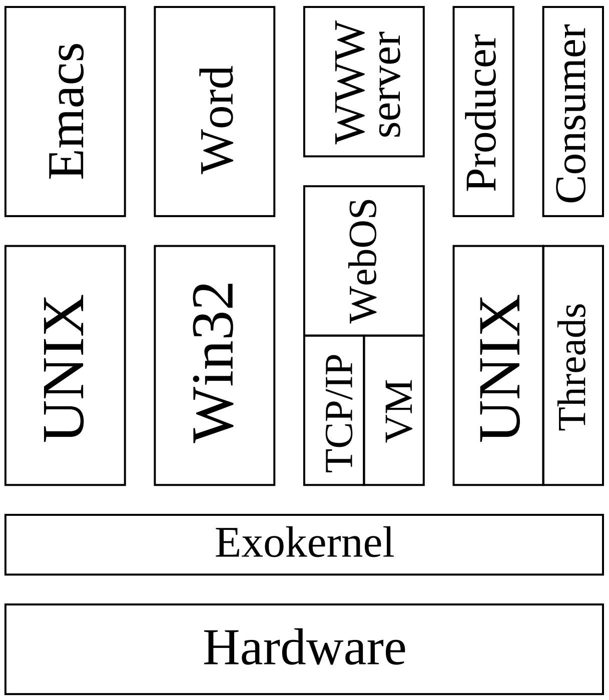

Figure 1: Applications running on top of an exokernel. The library operating systems provide different abstractions to the different applications.

In figure 1, an exokernel based system is shown. It is running two standard applications, Emacs and Word, on top of two different library operating systems that provides the API and functionality of two common operating system, UNIX and Win32. It runs a specially written WWW Server that is running on top of a custom made library operating system. Finally, a threaded Consumer/Producer application is running on top of a UNIX library OS that has been extended with customized thread support.

All four applications have their own library operating system that implements the OS abstractions needed by the application. An application and a library operating system executes in user space, i.e. outside the kernels protection domain, in the context of the same process and protection domain[1].

Having most of the OS in user space has some interesting consequences with respect to performance. A call from an application to a OS service is no longer more expensive than a function call, because it is just a function call — no system context switch is necessary. Also, the parameters supplied in an OS call need not to be checked as thoroughly, since an OS crash will only affect the calling application, not the whole machine. Also, any resource accessible to the OS can be assumed to be accessible to the application, reducing the need for access controls. Once the resource has been allocated to the application, the kernel is not involved in the management of the resource until it is deallocated.

There is, however, a price to pay for this: Although some OS functions can be handled without needing to call the exokernel, others may cause multiple calls to the kernel when the OS function

[1]The exokernel papers sometimes discuss *environments*, the context in which a process executes, but will often use the words *environment* and *process* interchangeably.

is made up of several primitive exokernel operations. Even though calls to the exokernel are more efficient than kernel calls in traditional operating systems, this situation requires careful design in order to minimize the number of exokernel calls needed.

One strategy to minimize the number of exokernel calls required is to directly expose kernel structures, e.g. by mapping them read-only into the applications address space, to avoid calls to examining the status of a resource. This means that calls are only needed to modify the state.

Keeping the exokernel access control management efficient is also essential, and is achieved by using a simple capability model. Capabilities to a resource is given at allocation time. Capabilities are thereafter shared and managed by the processes.

3 Previous exokernel experiments

The MIT exokernel team have explored possibilities offered by exploiting exokernel ideas. Below we will present some of their most important results concerning library OS and specific applications along with a discussion about the results. We will finish with one of our early experiments and some conclusions from that experiment.

3.1 ExOS

ExOS[13, 25, 33] is a library UNIX and it implements most of the functionality of an OpenBSD (BSD 4.4) system. Most UNIX applications can be compiled and run on ExOS.

The primary goal of the first version of ExOS was to show that a library OS can run legacy applications on top of an exokernel[33].

It was designed to be simple, flexible and give performance comparable to normal UNIX system. Speed of implementation was very important, so given a choice between a simple or a complex but efficient algorithm, the simple algorithm was chosen.

Flexibility and clean interfaces between modules was also important, since the code was meant to be customized or replaced for specific applications. In the FLUX OSKit[17, 16], these ideas are taken even further.

Finally, performance was important, since if the functionality can only be provided at a noticeable performance decrease, this would be a strong argument against an exokernel-based solution.

The measured benchmarks of ExOS 1.0 is not quite on par with OpenBSD running on the same hardware. However, most of the cases where ExOS is lagging behind is believed to be due to the choice of non-optimized algorithms and the lack of general optimization[33].

ExOS 1.0 does not provide the full semantics of OpenBSD — parts of it has been sacrificed to enhance flexibility. Other parts have been left out to because the goal was not complete compatibility — only the ability to run most common programs. Finally, some parts are implemented in a unsafe manner, both in terms of security and fault isolation, to simplify the implementation.

The goals for version 2.0 of ExOS are to deal with those areas left open by version 1.0: Better protection and fault-isolation, better security, better performance and more compatibility with UNIX.

3.2 Cheetah

Cheetah is a HTTP-server that has been designed to take advantage of exokernels to provide performance. The optimizations done in Cheetah are not dependent on the hardware architecture, Cheetah[26, 25] has been tested and benchmarked on both the Aegis[13] (DECstation/MIPS) and the XOK[25] (PC/Intel x86) exokernels, with similar results.

The Cheetah server running on an exokernel outperforms other comparable servers. For small (<= 1KByte) requests it up to eight times faster than the best comparable performance on OpenBSD and for large requests, Cheetah is limited by available bandwidth rather than by the server hardware. Cheetah delivers over 29.3 MByte/s with the CPU idle over 30% of the time.

Cheetah is an application on top of the library OS component XIO, an extensible I/O library for implementation of fast servers. The purpose of XIO is to allow programmers of I/O intensive applications to exploit knowledge about I/O resource usage.

Cheetah has been benchmarked against the Harvest[8] proxy cache, and systematically outperforms it on every test. The Harvest[8] proxy cache is a HTTP server which has been shown to outperform many other contemporary HTTP servers on traditional operating systems.

The Cheetah HTTP server is illustrative, because it achieves good performance via a fairly small number of easily understood techniques:

Co-location of related files Files that are related are co-located on the disk. This means that if an HTML-document consists of a base document and two pictures, these three files will be placed on adjacent blocks in the order they are most likely to be requested. For non-cached pages, this is show to double the throughput of the disk subsystem[25].

Packet merging The HTTP-server is integrated with the TCP stack, which means that if the server knows that it will produce a response to a request, it can hold off sending the ACK for the packet containing the request until the server responds, and piggy-back the ACK on the response. For small documents, this alone can reduce the number of packets sent by 20%[25].

Avoiding memory copies and CPU touching The HTML documents in Cheetah are stored as "canned" responses, complete with HTTP-response headers and pre-computed TCP checksums. This means that only a few bytes in the TCP/IP header needs to be changed.

The data is sent directly from the disk buffers, which eliminates the need to copy data from the disk buffers to the network buffers before sending them. Transmitted packets which are still un-acknowledged are locked in the file buffer cache until they are acknowledged, which eliminates the need for an explicit retransmission pool.

3.3 C-FFS

The Co-locating Fast File System (C-FFS)[18] is a file system implemented as a user level library. It provides all the protection and guarantees needed to implement UNIX file semantics, while still providing significant flexibility to user level software.

The C-FFS has been ported to OpenBSD, making it possible to compare the performance gain due to the file system design as well as to gains due to the underlying operating system.

In a set of benchmarks using unmodified UNIX programs such as `cp`, `rm`, `diff`, `gzip`, the benchmarks running on OpenBSD and C-FFS generally come out ahead of OpenBSD running with the standard file system. When the same benchmarks are run on the XOK[25] exokernel with C-FFS the results are typically as fast or faster compared to the OpenBSD/C-FFS combination.

The MIT exokernel team are keen to point out that while it is possible to realize similar improvements with traditional operating systems, as witnessed by the OpenBSD implementation of C-FFS, the porting of C-FFS to OpenBSD took more effort than it took to both design and implement it on XOK in the first place[25]. This should be considered as an indication that the flexibility of exokernels reduces the effort needed to implement new approaches compared to traditional systems.

3.4 XCP

The standard UNIX `cp` program is a lot less efficient than it could be. The MIT exokernel team has written an optimized file copy programs called `xcp` that gives an indication of how seemingly simple programs can be improved.

Copying files from one place to another should only be a matter of reading the data from the source location and then write it to the destination. The dominating factor affecting the performance of a copy should be the speed of the disks I/O operations when it is done properly. In practice, it turns out that `xcp` is three times faster than the standard UNIX `cp` command.

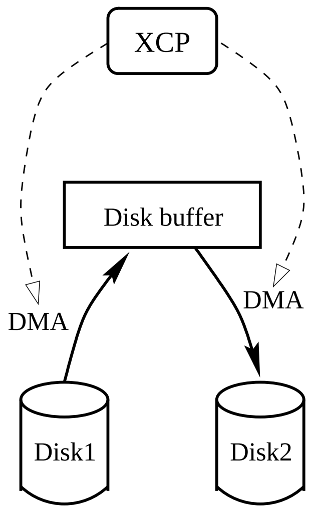

Figure 2: XCP copies files from one disk to another with DMA I/O, not requiring the CPU to ever touch the data being copied.

The difference between the two is that `xcp` exploits the low-level disk interface available to it. Compared to `cp` it removes artificial ordering constraints, improves disk scheduling through large schedules, eliminates data touching by the CPU and performs all disk operations asynchronously. In figure 2, the idea is presented. `xcp` only issues read and write DMA operations.

When operating on a set of files, `xcp` will construct a read schedule by sorting the disk blocks of all files and issue large, asynchronous disk reads using this schedule. While waiting for the file data read operations to complete, `xcp` will allocate inodes and disk blocks for the destination files and create the files. As the read operations complete, it issues write operations to write the blocks of the files to the destination directly from the disk buffers it previously read them to. The CPU is only involved in the creation of the inodes and the disk block allocations, the rest is just DMA.

The normal `cp` program could gain some of the benefits of `xcp` by issuing parallel asynchronous reads and hope that the disk device driver will merge read the block reads from the different files. However, the big gain of eliminating memory copies cannot be realized.

3.5 Binary emulation

The MIT exokernel team has written a partial OpenBSD binary emulator for XOK to allow OpenBSD binaries to be run directly on an exokernel.

Binary emulation is gaining in popularity. SCO is claiming that their UNIX can run unmodified Linux binaries via the `lxrun` package[34]. The reverse, i.e. Linux running SCO UNIX binaries, has been true for many years via iBCS[11] support. In the same vein, Sun has announced Linux binary compatibility for their Solaris operating system[30].

The XOK OpenBSD emulator is running side by side in user space with the application, which means that it needs no special privileges or changes to the kernel. The emulator currently only supports 90 out of 155 OpenBSD system calls, but has successfully run big applications such as Mosaic.

The performance is comparable to running the program natively, and in some cases it is even faster, due to the mapping of OpenBSD system call to cheaper exokernel implementations of the same functionality.

3.6 Discussion

When handling resources, the application programmer knows what he is trying to achieve. E.g., in a web server like Cheetah, the programmer knows that he is reading a block of data that will immediately be sent over the network to another machine. Typically, the OS will read data into a kernel disk buffer and then copy the contents to a user buffer. The programmer will then proceed to direct the networking code to send the contents of this buffer, which will probably be copied to a kernel network buffer and then sent by the network device. But the programmer know that moving the data to the user buffer is unnecessary — it could have been sent directly from the kernel disk buffer since the data did not need to be touched by the application.

The work done on the optimized file system C-FFS shows that by modifying just a single component in a system, in this case the file system, gains can be made for all I/O intensive applications using this library OS. This improved disk handling was achieved because exokernels export low-level disk resources and allow programmers to implement her own file system abstraction within the library OS.

Some things should however be noted with the examples that are discussed above:

- Cheetah has been designed specifically with an Exokernel in mind. It gives very good performance, but the question we should ask ourselves at this moment is if this is a typical application that will benefit from exokernels? Which applications can be improved like this?

  We believe that the answer is "no" and that Cheetah is an extreme case where exokernels have an edge and that are difficult to duplicate on traditional systems.

  Also, if a complete rewrite of an application is required to realize these improvements, how much of these improvements are just due to the application code being rewritten? Cheetah running on a traditional OS performs as well, or better, than the best traditional web server it is compared with.

- The C-FFS file system can and has been implemented on a traditional operating system. Even on a traditional OS, it performs better than the file systems it is compared to. Again, how much is due to a better design, and how much is due to exokernel features?

  But as pointed out, what's interesting with C-FFS is that implementing it does not require special privileges in an exokernel.

3.7 Incremental changes

The application and library OS base from the MIT exokernel team is quite large. Their ambition have been to design new applications that specifically exploit exokernel features.

We believe redesigning and re-implementing is too expensive. Incremental tweaking can, however, be motivated if the gain is sufficient to motivate the effort. We have experimented with some such minor modifications, not so much for the actual gains, but rather to see how much effort is needed to realize the gains. It turns out that many of the really major improvements would require that the design of the involved programs are changed.

It is a broadly accepted fact that applications often are designed based on the operating system and hardware they were first implemented on. This is typically most noticeable when porting a program from one platform, e.g. UNIX, to another that has a dissimilar architecture, e.g. MS Windows. Since most traditional operating systems have a fairly similar approach to abstraction, i.e. they will hide as much as possible of the underlying hardware semantics, it is not surprising that most applications originally written for a traditional OS will ignore most low-level details. When moving such an application to an exokernel architecture, it is hard to suddenly start exploiting the new features, because the necessary low-level knowledge that the applications programmer may have had, has not been incorporated in the application, since it could not be used on the system it was originally written for. The work needed to afterwards track this information and exploit may be on par with redesigning and re-implementing the application.

In one of the first experiments we[2] performed on an exokernel, we changed the semantics of NFS

[2]This experiment was done in collaboration with Thomas Pinckney and Héctor Briceño of the MIT exokernel team. It was done more as an exercise in modifying the library OS, than for the actual performance optimization.

writes to allow caching of writes. We took this slightly altered NFS and linked it with an application A which is used in a pair with a second application B that reads the file resulting from the first. By using the modified version of A, the execution time of application B is reduced by almost 50%. The real solution to the above problem would be to alter applications A and B so that they communicated via a pipe instead, however it may be the case that we do not have the source to application B, and then this example may be slightly more realistic.

This is a change that could and should not go into the mainstream version of the NFS code, since we basically violate the NFS semantics. It is safe for us to do this in this specific case, since we exploit the knowledge that the file has no significance except as input to the next program, and that any writes performed to this file by other processes should be ignored.

4 Exploiting knowledge of resource utilization to improve Erlang

One of the main reason for experimenting with new operating systems is performance. We want our programs to run faster on the existing hardware. This can be achieved both by improving the implementations of the features used, but also, and often leading to more dramatic improvements, by better exploiting knowledge of how resources are used. The two forces working against many performance optimizations are lack of portability and the extra effort needed.

In this section we will first examine why it is hard to use knowledge based improvements when programming for traditional operating systems using the Erlang system as an example. Then we will look specifically at the implementation of the Erlang system to see what kind of improvements can be done to it within the exokernel paradigm. Finally, we will list specific areas of improvements relevant to Erlang that the exokernel paradigm allow us to exploit.

4.1 Erlang and some definitions

We are looking at Erlang[5] from the outside, mostly from the perspective of the components “beneath” the Erlang system. This will affect the terminology, and may cause some confusion.

We have worked primarily with the JAM[4] version of the Erlang system, which is based on a byte code compiler and an abstract machine that interprets the byte code. There are other versions like the BEAM[20] version that compiles Erlang to C code, as well as HIPE[24], the JAM-based JIT-compiling Erlang implementation. We believe that most of what is said in this report is also applicable on the BEAM and HIPE versions.

For our discussions we identify three parts of a running Erlang program:

1. The application written in Erlang.
2. The Erlang interpreter
3. The run-time system (RTS). The RTS serves as the interface for the Erlang interpreter to the underlying OS, and will vary from OS to OS since it bridges differences between different OSs. The RTS also provides any OS-like functionality needed by the Erlang interpreter.

The actual distinction between the Erlang interpreter and runtime system may vary from one Erlang implementation to another. However, this does not affect the conclusions from this work.

4.2 Exploiting knowledge

In a typical multitasking OS, e.g a UNIX system, if a programmer wants to make use of knowledge of the underlying hardware architecture, she must do one of:

- allocate a whole hardware device (e.g a whole disk partition)

- modify the kernel
- give the application privileges to directly access hardware

In the first and last cases, the application will be given exclusive access to the resource because there is no low-level multiplexing mechanism. In the last case also means that privileges outside of the needed may be granted, because the operating system privileges cannot be confined to a specific hardware. E.g direct access to hardware under UNIX and similar operating systems means that all OS protection can be circumvented, both intentionally or unintentionally.

The middle alternative, modifying the kernel, has the unfortunate side effect of changing the behavior not only for the specific application, but for all applications. Also, it can only be undertaken if the source code of the OS is available. Finally, it means that the kernel will grow for each application users want to be *able* to optimize performance for. Some OS's provide a loadable kernel module functionality, but this will not solve the main problems. Since a kernel module will execute in kernel context, any malicious code or programming error in it can bring the whole machine down.

All in all, this means that, programmers are left with either the general abstractions, or has to create a dedicated system running directly on top of the hardware. With exokernels, a middle ground between these two choices is created.

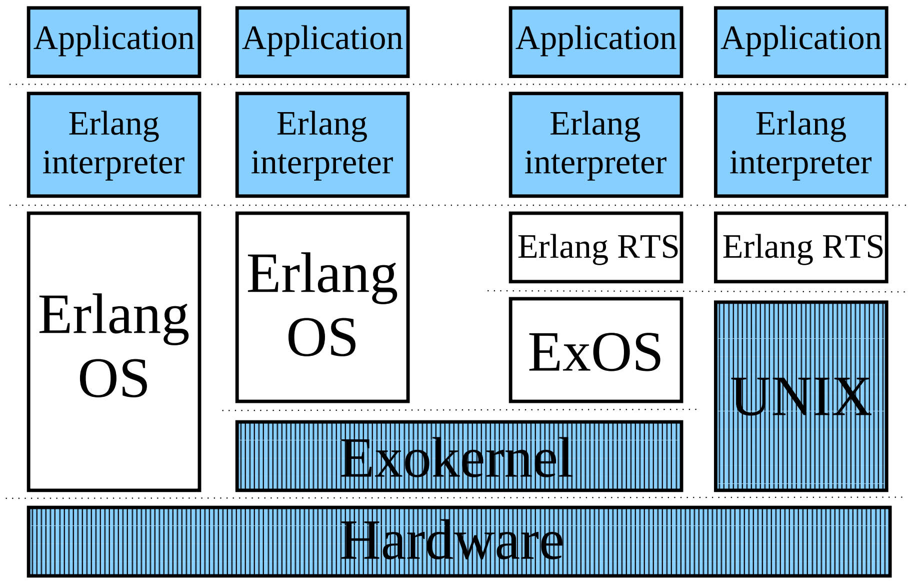

Figure 3: Four different approaches for an Erlang application: Dedicated system, dedicated library OS on exokernel, standard library OS on exokernel and traditional OS (UNIX) solution.

In figure 3 four alternatives are illustrated with Erlang running on different software platforms on the same hardware. From left to right we have:

1. A dedicated system. On such a system the implementor has full freedom to implement all necessary abstractions to perfectly fit the Erlang system, but at a very high labor cost since he has to do all the work for each new hardware platform.

   Correctly done, this should result in the most efficient system possible.

2. A dedicated Erlang library OS. With this system, we retain most of the flexibility of the previous case, but still a substantial amount of work needs to be done to to write a dedicated library OS. The exokernel will, however, provide all the low-level hardware interfaces.

3. A standard Erlang system running on top of the UNIX look-alike library OS, i.e. ExOS[33].

4. The standard Erlang system running on a UNIX system.

The two last systems should be virtually identical — it is the same code, compiled for two different UNIX-systems. The only difference is that on an exokernel, the Erlang implementor can start with a library operating system mimicking a standard UNIX and incrementally modify the library OS and Erlang runtime system until they are merged and specialized into an Erlang OS. A nice feature is that two versions of the Erlang system, one modified an one unmodified, can be run in parallel, allowing their behavior to be compared to determine if the change changed semantics and to see how much difference the change made. With a dedicated system, two sets of identical hardware would be needed to do this.

4.2.1 The trade-offs

Looking at figure 3, the interfaces between different parts of systems has been illustrated as a thin dotted line. All four systems have the same hardware API. Similarly, all systems running on top of an exokernel has the same exokernel API, e.g. XOK or Aegis. ExOS is a 4.4 BSD UNIX look-alike, and thus an application running on a normal UNIX system and on top of ExOS share the same 4.4 BSD API.

We have tried to illustrate the fact that we do not wish to modify the Erlang interpreter or the language, by showing all four implementations alternatives of Erlang to use the same API from the interpreter and upwards. Or put in another way: The white boxes in the figure are the ones where we can make changes with impunity. The darker striped boxes we consider impossible or impractical to change. While changes are possible in the plain dark boxes, any changes we make here must not change semantics and must be simple to maintain across different operating systems. We believe changes here should be kept to a minimum to avoid forking the development into an exokernel version and a mainstream OS version.

4.2.2 Performance, effort and visible modifications

We believe that three factors determine if, how and when performance optimizations should be done to an existing portable application:

1. The performance gain.
2. The effort needed to realize it.
3. The visible modifications necessary.

The improvements we can show in an application is the motivation for the extra effort and the visible modifications of the common code base[3] necessary. If improvements are small, they may not be worth the effort at all, and if they are very big, can motivate a complete rewrite.

The effort needed to realize an improvement can be hard to estimate beforehand, but regardless, we can expect there to be a non-zero effort needed to improve the code, even if the change only involves linking with a special library OS component.

The final factor, the visible modifications, is probably the one needing the most explanation. With visible modifications, we are not necessarily talking about changes that are noticeable when running the program, but rather changes visible in the common source code. Any improvement that can be done without requiring a change in the sources common to all platforms that the program is running on, is essentially "for free". That is, it can be done without regard to its implications for other platforms.

Changes that affect the common code base will cause portability problems every time the it is updated. If the changes are merged in with the common code base, and conditionally compiled, then this alternative code may slip behind the baseline version. If the code bases are not merged, then updating the code will require manual intervention every time a new version of the common code base is released.

[3]By *the common code base* we mean the machine/OS independent parts of the code.

4.3 Improving Erlang

Erlang is a general programming language, which means that we cannot transparently improve resource use for all application written in Erlang. The resource use of programs written in Erlang is up to the programmer and cannot be predicted a priori.

We can, however, improve the Erlang system itself. Since the most commonly used Erlang implementation is interpreted, this opens a new way to improve performance — improving the interpreter by utilizing knowledge about the interpreter's resource usage, rather than the interpreted program's. Attempt can also be made to improve the function library and runtime system. The runtime system (RTS) is also attractive because it provides the interface for Erlang to underlying operating systems abstractions and thus allows us to remove redundant abstractions.

Generally, a runtime system is a layer that serves to hide the underlying OS from the application, to make the application OS independent. The application routes all OS-type requests through the RTS, effectively making the RTS the OS for the application. If the abstractions provided by the RTS is very similar to the underlying OS, the RTS can be made very thin. This leads us to conclude that the only OS functionality needed is the one used by the runtime system.

Therefore, in an exokernel based system, we can eliminate the distinction between RTS and OS, or rather merge them into one system. Having a merged RTS/OS would mean that we could retain most, or all, of the API used by the interpreter intact, while having full freedom to modify the actual implementation of the RTS/OS.

The resulting system should give most of the advantages of a dedicated system in terms of freedom of implementation and tuning, while retaining all of the benefits of a traditional operating system. In fact we could implement complementary sets of RTS/OS functionality that can be run side-by-side in the system.

4.3.1 Specific classes of applications

If we allow ourself to limit us to a smaller class of programs than "all programs that can be written in Erlang", we thereby gain more domain specific knowledge of resource usage. As discussed above, this allows us to make a domain specific implementation that is more efficient than a general purpose implementation.

The situation is analogous to traditional operating systems — if we know beforehand what types of applications an OS will run, we can include support to optimize performance for these specific types. In a similar way, we can export functionality to programmers that is optimized for the kind of programs we know that they will be writing.

Protocol implementations is an important area for Erlang and we will discuss how performance can be improved for this subclass of Erlang programs.

An interesting question that we do not address is *how* to provide support for protocol implementation — should this be done by modifying existing language abstractions, or by providing function libraries that implements enhanced functionality? We are focusing on the means to improve performance, but at some point, the way we integrate these improvements with Erlang will affect how they can be used.

Our recommendation is to chose one of two possible ways:

- Create an annex containing specific function for the specific task. There are plenty of precedents in the Erlang function libraries[32] for this.
- Create a library OS specifically tuned for the type of problems you want to optimize for, and build a dedicated Erlang system linked with this library OS.

In a compiling Erlang system[20], the latter could be used to chose the library OS components that best fit the program being compiled.

4.4 Specifics

We have looked at Erlang and exokernels from two different viewpoints: General speedups and optimizing for protocol implementation. In in table 1 we present the operating system areas we believe can be exploited. The areas listed are discussed and summarized in the following sections. We include areas where we believe that changes would violate our objective of not changing the common code base or change semantics of the language.

Area	Erlang	Runtime	Protocol implementation
Virtual memory for tagged addresses	XX		
Physical addresses and cache behavior	XX	XX	X
Virtual memory and page handling	XX		
Memory mapping and copying	XX	XX	XX
CPU scheduling		XX	
Disk management		X	
Memory management		XX	
Interprocess communication	XX	XX	
Downloaded code		XX	XX
Interrupt handling		X	XX
Packet filters		X	XXX
Wake-up predicates		XX	XX

Symbol	Meaning
X	Minor gain
XX	Gain
XXX	Big gain

Table 1: Areas we have identified as possible candidates for optimizations, which components they affect and the gain we estimate.

Our estimation of the usefulness of different features is based on our own experiments as well as other published research. Where relevant, we give references to published papers in the summaries.

4.5 The affected components

For the interpreter, we believe that it is mostly in the area of memory management we can improve performance, since this is the primary resource used by the interpreter.

While some memory management features, especially the cache-related features, can also be used in the runtime system, we believe that most of the features useful to the runtime is with other resources than memory.

The Erlang interpreter and runtime system are components that will be used by all programs written in Erlang. When we narrow our focus on protocol implementation, we also gain specific knowledge about resource use. We know that we will have to deal with protocol data units (PDUs), and can implement handling optimized for delivering PDUs between applications and networking hardware.

4.6 Areas that can be used for improvements

Here follows a short description of the different areas of exokernels which can be used to enhance performance.

4.6.1 Tagged addresses

Erlang uses tagged addresses (see figure 4) to identify the type of the data structures pointed to by an address. This is done by using a few bits of the address as a tag. Doing so will, however, have a performance penalty, since it requires several instructions to dereference a pointer as well as reducing the amount of memory addressable by Erlang.

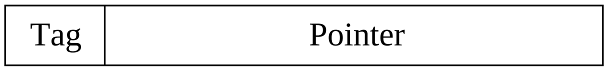

Figure 4: A tagged address — part of the actual address is a tag identifying the type of address.

The advantage of using part of the address as a tag rather than a structure consisting of an address and a tag is that it will use less memory, and that the standard pointer type can be used when handling tagged pointers. Special handling is only required when dereferencing the pointer.

The downside of this strategy is that we must remove the tag and restore the original pattern in the tag bits every time we want to dereference the pointer, i.e. we must "wash" pointers before using them, see figure 5. Washing a pointer can be done by an AND-operation to remove the tag, and an OR-operation to restore the original contents of the tag bits. Thus dereferencing, which normally takes one instruction, would instead take three instructions, adding to memory and cache footprint as well as execution time.

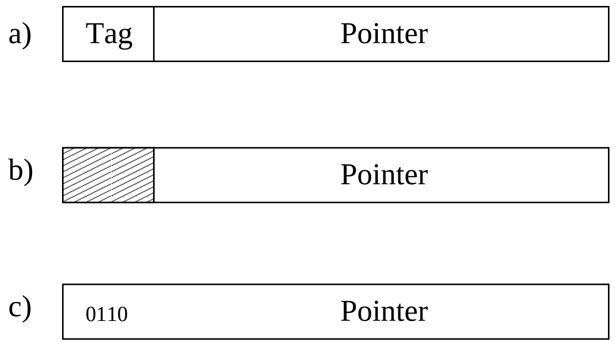

Figure 5: Washing the pointer. **a)** The original tagged pointer. **b)** The tag part of the address is discarded. **c)** The original pattern is restored.

Using part of the address as tag also means that we cannot the whole memory space. For every bit we use, we halve the usable address space. For small tags and limited applications this may not be noticeable: In many operating systems, parts of the address space is used by the kernel, and thus not mappable by applications anyway, because the operating system reserves it[15, 13, 29]. Also, if the size of the objects pointed to by tagged pointers are known to always be a multiple of bytes and always placed on aligned addresses, then the low order bits of the address are redundant and could be used as tag bits without losing information.

The main problem is thus the processing of addresses needed, rather than the reduced address space. If the tagged pointer could be used directly, this would eliminate the need to wash the pointer as well as allowing Erlang to use a larger part of the virtual address space.

In theory this should be a fairly simple thing to do if we have full control over the memory system as we are supposed to have in an exokernel. For XOK[25], however, it turns out that this solution is not feasible because kernel structures are mapped into parts of memory that we would want to be able to use for Erlang structures, in effect XOK have the same limitations as other operating systems in this area, i.e. part of the virtual address space is reserved by the kernel.

The second best solution would then be to at least reduce the number of operations needed to dereference a tagged pointer. As seen above, two operations are needed to wash the pointer. However, if data is mapped to addresses for which all the tag bits should be zero, then we could skip the OR-operation, and only require an AND to clear the tag bits, reducing the instructions needed for dereferencing a tagged pointer to two instructions.

The tags of the Erlang version we have been working with are four bits wide. Unfortunately, the normal mappings of ExOS/XOK (see figure 6) place the application heap on an address that would require us to both clear the tag and restore the original pattern. By moving the starting point of the heap to a lower address, we can get the desired properties and can eliminate the OR-operation.

Moving around the heap and modifying the layout of memory would on a traditional operating

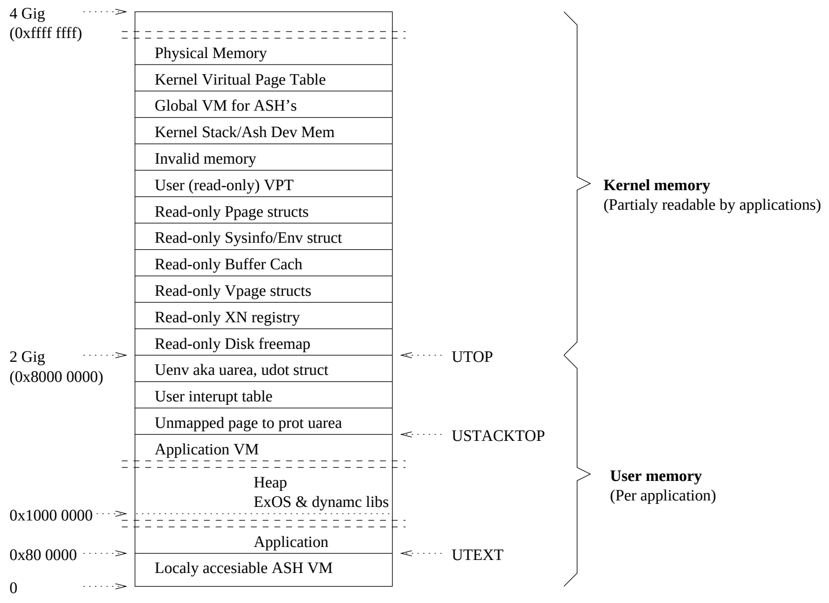

Figure 6: Normal memory layout of ExOS/XOK.

system be a major undertaking, and we would probably want to instrument the Erlang code to determine if dereferencing is such a common operation that changing the operating system to optimize it would be motivated. Since we're working on an exokernel where operating system changes are relatively simple, we can use a different experimental methodology: The time needed to actually make the change to the operating system is comparable to the time needed to instrument the program, so instead of instrumenting to determine feasibility, we can implement the change and benchmark the two versions.

The bad news is that in this particular case, it turns out that for all our test cases, the difference between the three and two instruction washing is negligible.

4.6.2 Cache behavior and memory speeds

Cache lookups are typically performed by the CPU using physical addresses. Most operating systems hide the physical addresses from applications, and exports virtual addresses. This makes it impossible for an application to predict and control cache behavior to get higher and more consistent performance. With an exokernel, a process can explicitly ask for specific physical pages to be mapped at specific virtual addresses, thus enabling the application to chose physical addresses so as to be able to layout addresses so that cache conflicts are avoided. E.g. if two chunks of data is used simultaneously, performance will suffer if the chunks are placed on conflicting cache lines.

A final consideration is that some systems cannot cache all addresses in their physical address space, for example the Intel Pentium II 233-333MHz processors can address 64GB of memory, but only the first 512MB can be cached[23]. In some systems, parts of the physical address space have an access speed penalty compared to other parts. Most operating system do not have a mechanism to indicate the characteristics of different types of memory, which means that addresses from these slower memory areas are not available at all to applications, or addresses from these areas will be handed to the application with no indication of the speed penalty.

If the application and compiler know about the characteristics of different physical memory addresses, they can avoid this slower memory for time critical parts. The application could also knowingly use the slower memory for storage of infrequently accessed data.

In an exokernel, an application can request that specific physical pages be used to a virtual address, which means that in an exokernel, the library OS can be designed to provide fine grain control over what types of memory it offers to an application. It also means that the application can request memory that will not cause cache conflicts with existing memory areas.

4.6.3 Virtual memory and page handling

A virtual memory system has typically a global strategy for handling page replacement. The Least Recently Used (LRU) algorithm is often used[37]. It evicts the page least recently used when the system runs out of physical memory.

While LRU is a reasonable strategy in most cases, it can give rise to unwanted behavior in some circumstances, i.e. when touching a page is actually not a good indication that it in most cases will soon be needed again. This mismatch happens e.g. during a garbage collection (GC). For GC we have two unfortunate scenarios: a) Pages will be brought in from disk that will not be needed again anytime soon. This will cause other pages, that may be needed again soon, to be written out to disk. b) GC will touch pages that are in reality good candidates for paging out to disk, causing other, more useful, pages to be paged out instead.

Exokernels allows full control over paging and the algorithm controlling page replacement. By allowing the application to control page replacement, it can modify the page replacement algorithm to take these unwanted effects on paging into account. The normal page replacement algorithm can be suspended, and another strategy can be used during operations that have unwanted effects.

For a garbage collect, a small set of physical pages could be allocated as a page pool for the GC. The GC can page in any pages it desires, but when its pool of pages is used up, it has to page out a physical page from its own pool, not the physical memory of the whole application.

4.6.4 Memory mapping to avoid copying

Modern operating systems have support for shared memory between processes. Exokernels add a finer control over memory mappings, as well as a kernel infrastructure that gives the application more flexibility in exploiting memory mapping.

Many OS operations involve moving data between buffers, either between kernel and user space, or between kernel internal buffers. There are also operations performed by applications that involve copying. When copying small amounts of data, or when copies are infrequent, the effect on performance is often negligible. If copies of large memory blocks are done frequently, then they will have a serious impact on performance.

In these cases, the destination virtual addresses can be mapped to the same physical memory pages as the source addresses. This is an efficient solution for the cases where the copied data is not altered or when changes are tolerated in both copies. In the cases where changes may not be propagated between copy and original, remapping can still be profitable. The addresses of source and destination will be mapped to the same physical pages, as before, but the mappings of both copy and original will be marked copy-on-write. If an attempt is made to modify a page that is mapped copy-on-write, then the page will be copied, the virtual mapping will be changed to map to the copy, and only then the modification is done. The result is that only those pages that are modified will be copied, the rest will just be remapped.

The cost of remapping must be considered. For small amounts of data, a copy can be cheaper than remapping memory. On the other hand, if we can replace several copy operations by one remapping, the cost of remapping can be amortized over all copy operations. The actual cost of remapping vs copying data varies between different hardware architectures, so the exact trade off vary from system to system.

An example of an operation that involves a bulk copy of data that will be partially modified is the UNIX `fork()` call. In most systems this call is implemented as a copy-on-write remapping of the pages from the parent process to the child process[29] rather than as an actual copy.

In an exokernel based system, most operating system functionality will be implemented outside the kernel. This means that to achieve the same performance as on traditional systems, memory mapping must be available to applications in exokernel systems. The decision to use copy-on-write mappings rather than a copy is a "gamble" that relies on knowledge that most pages will not be written to. The application writer is in a position to know if that is the case, and can instruct the exokernel to map these pages copy-on-write rather than copying their contents.

Device I/O is another area where remapping strategies can be used. By mapping the device buffers into the virtual address space of the application, data do not need to be copied from the device buffer to the application. For a device buffer mapping scheme, the best strategy would probably be to reuse mapped buffers for several I/O operations in order to amortize the cost of mapping the page over several uses of the buffer. Doing so will require cooperation from the application (or library OS) to avoid that pages currently used for I/O are used by the application[2].

For hardware supporting scatter/gather I/O[29, 10] for devices such as network cards, memory mapping schemes are very attractive, since scatter/gather allow the kernel or library operating system to place the header in private memory while placing data in application readable buffers[1].

Other uses of memory mapping is when exposing kernel data structures to the application and library OS. We could map the kernel page containing the structures read-only into the applications address space rather than providing a system call to copy the information from the kernel to the application. This mapping would have to be done only once for the whole life time of the process. Exokernels use this technique to expose kernel information to the application and library OS[13].

4.6.5 CPU scheduling

In most operating systems, control over CPU scheduling for an application is a question of assigning a priority to the process. In some operating system, the application can additionally affect how it is scheduled by choosing a scheduling class that fits its profile. No control over how to handle re-scheduling is given to processes, and this means that critical sections must be implemented via system wide semaphores, since the process can be de-scheduled at any time.

Threads can either be available as kernel controlled objects, or implemented by the user level process. Kernel threads means that the kernel is aware that a process can have several threads of control running simultaneously, and will schedule them pre-emptively and ensure that the threads are handled as first class operating system objects.

User level threads, on the other hand, allow an application more control over how threads are implemented. The application can impose its own thread semantics and chose an implementation that better fit the applications needs. The down side is that the OS only expects a single thread of control in a process. This will manifest itself in different ways, but typically it means that one user thread calling a blocking OS function will cause all other user threads within then same process to block as well. Hence, pre-emptive scheduling of user level threads cannot be done, and some type of cooperative scheduling must be done instead.

The process model of exokernels is very simplistic. Most functionality has to be implemented by the application, not only control over threads but also over process activation. The exokernel will generate an up-call (much like an exception) when a process' time slice starts and when it ends. The process has to handle context restoring and saving by itself, which has the advantage of allowing scheduler activations[3]. The process is also free to chose what context to restore on activation, i.e. the process can implement pre-emptive scheduling of sub-processes (threads) as it sees fit.

When an up-call to end the time slice has been received, it is up to the process to decide what parts of the context it must save. E.g. if the programmer knows that the process does not to use the FPU, then he may chose not to save the FPU state.

If a process overruns its time slice, e.g. it wants to finish a critical section before relinquishing control and does not release control within its time slice, the overrun will be deducted from the process' subsequent time slices until it has forfeited an amount equal to its overrun. If the overrun is too big, the process will be terminated by the exokernel. The downside of allowing overruns is that it will make real-time scheduling hard. As long as all running processes cooperate, it should work, but no hard scheduling guarantees can be given by the exokernel.

Scheduling of processes is not done by an algorithm. Instead it is done by letting the application allocate time slices in a time vector. This means that the scheduling policy of a process is a question of how many and which time slices are allocated in the vector to the process. In figure 7 an example time vector is shown. Big CPU-bound jobs tries to allocate as many adjacent slots as possible, to reduce context switching overhead, while interactive and real-time processes would try to allocate evenly spaced slots for fast response times. A small program needing CPU only for occasional updates, would only allocate a single slot, big enough to let it finish the update.

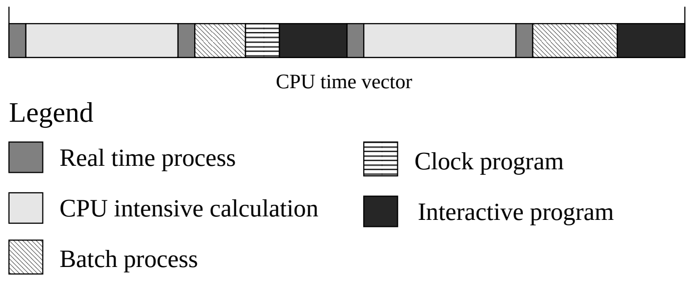

Figure 7: The CPU time vector. In this example, five processes have allocated all available time slots in accordance to their scheduling preferences. The length of the vector is determined by the implementation.

Background batch programs, finally, allocates any time slots available.

Since all processes allocate their own slices, a greedy application could lock out all other processes. A global scheduling can be implemented by letting the first process that starts to allocate all time slices, and then implement a global scheduler using the yield primitive that donates the rest of a time slice to a named process. It could schedule other processes by successively yielding time slices to them according to its scheduling mechanism. This first process could also could also serve as a priority based manager for the time vector, overriding allocations based on the priority of the allocating process. In figure 7, this would mean that even if the batch process had allocated all available slots, it would lose them if a higher priority process wants slots.

The MIT exokernel team have implemented and tested a stride scheduler[39] to demonstrate that a user level scheduler can be implemented given the primitives provided in the exokernel.

For Erlang, these mechanisms could be used to implement Erlang processes. This could combine the advantages of the current implementation with the advantages of mapping the on OS threads[21].

4.6.6 Disk management

A traditional OS normally offers two different ways of using disks: As structured file systems, or as raw disk. In the raw form, the unit of multiplexing is typically a disk partition of a static size — i.e. we can assign a partition to an application. Databases often have their own disk handling using a dedicated partition in order to increase performance and get better control over data placement.

If we want to share disk with a finer grain granularity than a partition, we have to use the file systems available. But in doing so, we also delegate control over placement and format of data on the disk to the file system.

In exokernels the unit of multiplexing is individual blocks, rather than files or partitions[25]. An application can implement its own file system using the blocks. An application file system can exist side by side with a standard file system.

The main difference between implementing a file system on top of a raw disk and in an exokernel is that under an exokernel, physical disk blocks must be allocated from a common pool shared with other applications. I.e., if an application wants a specific disk blocks, it may already have been allocated by another application. Also, the exokernel requires that the application provides a mechanism to identify allocated blocks. This is done via a Untrusted Deterministic Function (UDF) that allows the exokernel to interpret the user file systems own metadata. I.e. the blocks used by a user file system will not be tracked and stored by both the kernel and the file system, instead the exokernel will use the file systems metadata via the UDFs.

The disk system also provides a user controlled buffer cache mechanism to handle cached disk blocks without actually imposing a specific disk caching scheme. Cached blocks are handled by the user application, the disk system only provides a way for file systems to register cached disk blocks.

For casual disk use, the standard file systems will probably by sufficient. The type of applications that will benefit are those where disk I/O is a bottleneck, and have disk seek patterns which can be predicted or where disk block data should not be touched as in Cheetah (see section 3.2) and

XCP (see section 3.4).

4.6.7 Memory management

It can be assumed that any given application will need memory. For simple applications, only memory for statically allocated variables and perhaps a few variables on the stack will be needed. Applications that have more dynamic memory needs will allocate and deallocate memory dynamically during the execution. But why should we stop there? We have already discussed how an application could affect its cache behavior and also seen how an application could increase its performance by modifying the page replacement algorithm.

Applications can use knowledge about the contents of pages for more efficient memory handling. E.g., if an application knows that it is no longer using a page, the application can unmap the page and release the physical page to the common pool of free pages. Doing this means both that applications will utilize resources better (they are only holding pages they really need), but also that the overall performance will be better since when a process that needs memory for a page, it can get one immediately from the pool, instead of first having to write out a page to disk. If a garbage page has been written out and we want to reuse the same virtual address for new data, we will have to read the page back from disk, even though the application already knows that the current contents of the page is garbage.

There are a few options available when a physical page is needed. If the application has structures in memory that can be recalculated, it might be more efficient to release the memory used by those structures rather than page out other code/data. If the application uses speculative caching or pre-fetching to increase its performance, then it could elect to release a cached or pre-fetched page rather than lose a page containing code/data. Speculative caching and pre-fetching is often a good strategy to use otherwise unused memory, but if the caches push out code or data, this may end up hurting performance. In fact, if the infrastructure to selectively release pages does not exist, the application programmer may hesitate to add caching and pre-fetching.

The most recent exokernel implementation, XOK, does not support much in the way of paging. The idea presented in the exokernel papers[13, 25] is that the exokernel will ask, via an up call, an application to release a physical page. The application then has a certain time to comply to the up call, and if it does not release a page within the set time, a page will forcibly be removed by the exokernel and the application will be notified that the page was reclaimed.[4]

To lessen the impact of such a drastic action a *repossession vector* is proposed[13]. The application puts pointers to pages it does not mind that the exokernel revokes into this vector. The applications should also provide the exokernel with capabilities for backing store resources for storing the contents of reclaimed pages.

4.6.8 Interprocess communication

Interprocess communication (IPC) is used to allow processes on the same machine to communicate with each other to exchange data or to synchronize with each other. The mechanisms used to implement IPC varies, but ultimately it is a question of transferring control and a buffer of data from one process to another. The simplest form would be to use a shared memory buffer, and to poll a pre-defined part of the buffer to determine if a call has been made by the other process. Another possibility would be to use a messaging mechanism in the kernel that signals the receiving process when a message is available[36].

An important performance factor is the latency IPC has, i.e. how long time does it take from the time process A sends a message until it is received by process B? Latency is especially important if IPC is used to call functions in another process i.e. Remote Procedure Calls (RPC), since if calls between processes are much more expensive than calls within a process, then it may make more sense not to split functionality into multiple processes, regardless of other advantages.

Exokernels provides mechanisms to transfer the thread of control between two processes as well as the possibility to share memory, and by using these mechanisms the application programmer can

[4]A small set of guaranteed mappings should always be provided to the application, so that it can store its management routines (pager etc).

build his or her own IPC abstractions. The actual transfer of control is simply done by changing the program counter to an previously agreed address in the callee and donating the rest of the current time slice to the callee. Protection is managed by the callee, i.e. it is up to the called process to determine if should accept the call or not.

The transfer of control can be done asynchronously or synchronously. The former only donates the current time slice, whereas the second donates the current and all future time slices until the callee returns control to the caller. The latency of a synchronous transfer of control in the Aegis exokernel is 30 instructions[13].

All process visible registers are left intact during the transfer, which means that the registers can be used as message buffer[9]. If this is not sufficient, then a permanently shared memory buffer can be used instead.

The IPC building blocks provided by exokernels can be used to build many different flavors of IPC and RPC abstractions. An untrusted RPC mechanism would not trust the callee to save the contents of registers during the RPC, but if server and client trust each other, then the responsibility can be migrated to the callee, which can save only those registers used by the RPC call. This will further reduce the latency of RPC calls[22].

4.6.9 Downloaded code

A problem in an exokernel operating system architecture is that the kernel does not know anything about the resources handled by the application and library operating system. Since the kernel does not know how to react to different events such as I/O completion, packet arrivals etc, it must rely on the application to handle the event. This, however, is expensive since it requires the application to be scheduled.

The idea of downloadable code is that the application provides code that identifies or transforms in-kernel data for the kernel. The code is checked by the kernel to ensure that all memory accesses are legitimate, effectively confining the code to a sandbox[38], and that the tests performed does not overlap with previously downloaded tests. Once the code has been checked, the kernel can compile it to machine code and install it. It can then call this code to perform the relevant tests and transformations and then interpret the results, without actually having to know what is being tested or what the format of the original data was[12].

The real power of downloaded code is, however, not speed, but the ability to delegate management from the kernel to application code even in situations where the kernel cannot trust the application: Downloaded code can be checked and execution times and access be bounded at download time. This is for example true of the exokernel disk support in which ownership of blocks is resolved by a downloaded piece of code, a UDF, or the packet filters and wake-up predicates discussed below.

Other operating systems that allow user programs to download code include the SPIN[7] micro-kernel and Vino[35].

4.6.10 Interrupt handling

By hiding interrupts, traditional operating systems simplify for the programmer, but at the same time this will restrict the flexibility. By exporting interrupts to an application, it can directly manage hardware devices associated with the interrupts. Also, to implement efficient critical sections, control over interrupts must be delegated to the application so that incoming interrupts will not break the sections.

For some interrupts, the latency involved in a calling the application to serve an interrupt is not acceptable. For these it would be beneficial to download interrupt handling code into the kernel.

The exokernel implementation Aegis[13] provide the application with an interface to interrupts as well a method to download interrupt handlers into the kernel. While downloadable interrupt handlers offer elimination of kernel crossings, the low cost of kernel crossings in exokernels renders this advantage very small. By providing a fast up-call to the application, most of the benefits of downloadable code will be retained. For mainstream operating system with expensive context switches, downloadable interrupt handlers may still be a good solution[25].

4.6.11 Packet filters

The multiplexing of network resources normally done in traditional operating system is to use a shared protocol abstractions that allows applications to establish connections and send/receive packets. This, however, limits applications to use the protocols provided by the operating system and the network interface.

For the outgoing direction this is not a problem, but for the incoming direction the underlying fundamental problem for the kernel is how to de-multiplex incoming packets and establish ownership of them. If the OS does not know about the protocols used or connection to process id, it cannot determine what process should get an incoming packet.

A solution is downloading code, a packet filter[31], into the kernel that the kernel can run to determine ownership of a packet. The packet filter allows the application to describe to the kernel the packets belonging to it. Once identified, the packet can be dispatched to the proper process, which then is free to implement the actual protocol, i.e. there is no need to use a shared protocol abstraction. In the Cheetah HTTP-server, this is used to allow a custom TCP/IP protocol implementation that in turn allows Cheetah to do intelligent packet merging (see section 3.2).

As the code is downloaded, the kernel will test to see if this packet filter conflicts with any previously downloaded packet filters. If not, it will be merged with the previous packet filters.

The implementation of packet filters, Dynamic Packet Filters (DPF)[14], used in the Aegis and XOK exokernels differs from most other implementations[31, 6, 28, 40] in that it compiles the packet filters to machine code, rather than using an interpreter. This makes installation of packet filters slower than in other systems, but the demultiplexing performance is reported to be on the order of 10-20 times faster than the other systems[14, 13].

Fortunately, packet filter installation is only needed at connection setup and closing time. If a programmer knows beforehand that he will need multiple connection during the lifetime of the application, a filter that matches the packets of all connections can be downloaded at startup, rather than installing several more restrictive filters. In this case, the fine grain filtering is done by the application.

The packet filtering mechanisms can be exported directly to a Erlang program via an annex. This would allow the Erlang programmer to describe PDUs and then, using zero-copy mechanisms, allow the Erlang application to directly operate on data from the network buffers.

4.6.12 Wake-up predicates

One of the big differences between an exokernel and a monolithic kernel design is that the monolithic kernel has access to all OS structures, which means that it is fairly straight forward to construct mechanisms to let processes sleep while they are waiting for events such as I/O completion, expiring timers etc.

If such abstractions are removed from the kernel and all those structures and mechanisms are delegated to the user level OS, this means that if a process is waiting for an event, it must be woken up regularly to let it check for the event. Unless we are prepared to accept significant latency between events happening and the process reacting to them, this means busy-waiting for events.

To avoid this situation, exokernels provide kernel wake-up predicates. These predicates are downloaded into the kernel by the process, much in the same way packet filters are, and specifies a set of events that should cause the kernel to wake this process up. The events are essentially just patterns in memory at locations accessible to the process. The memory locations could hold anything — a memory mapped timer register, a buffer structure or any exported exokernel structure.

A wake-up predicate may involve multiple sub-predicates. For example to allow for several events to generate a wake-up, while waiting for I/O to complete, the application wants to handle both I/O completion, as well as error indications or timeouts.

Since the downloaded wake-up predicate is compiled to machine code, they should be comparable to the cost in a traditional kernel.

```
void cat_test(void)
{
  LOOP(5) {
    fork_and_run_function_in_child(do_cat_test)
  }
  wait_for_children_to_return();
}

void do_cat_test(void)
{
  spawn("cat", fd_in, fd_out);
  LOOP(4800) {
    write(fd_out, buffer, 200);
    select(fd_in);
    read(fd_in, buffer, 200);
  }
  LOOP(4800) {
    write(fd_out, buffer, 5);
    select(fd_in);
    read(fd_in, buffer, 5);
  }
  return(0);
}
```

Figure 8: Pseudo-code for a simple test program runs multiple processes that "ping-pongs" data via an external process.

Wake-up predicates are executed in the context of the kernel. This means that they could be used to let one process to snoop on other processes private data. To guard against this, all addresses references are checked for access rights before the predicate is compiled to machine code and installed.

Our practical experiences with wake-up predicates in ExOS/XOK[33, 25] for Erlang show an unanticipated performance loss. The reason is the cost of installing the wake-up predicate. It cannot be amortized over multiple uses in the same way as for a long-lived communication connections. A wake-up predicate is likely to be discarded after its first positive match. If the time between installing a predicate and wake-up is long, then it will of course still be a gain to have installed it.

We have experimented with the `select()` call in ExOS, which basically installs a simple wake-up predicate that will wake the process up when data is available for a specific file descriptor. This experiment was prompted by the fact the **Port I/O** demo benchmark for Erlang on ExOS/XOK ran significantly slower than on OpenBSD or Linux (see section 5.1).

We have two versions of a simple test program that captures the essence of the Erlang Port I/O demo. The program forks off a number of processes that ping-pong data to and from an external program. Version A (see figure 8) waits for data using `select()` before reading, and version B, which is identical to A except for the `select()` calls, just tries to read data immediately.

We expected that version A, which uses the `select()` call, would be slower on all platforms since it does more work. Running the test programs under OpenBSD and Linux, we found that the version A indeed is about 20% slower than version B. Performing the same test under ExOS/XOK, we find that version A is more than 500% slower than version B.

Instrumenting the code, we find, as we suspected, that it is indeed inside the `select()` call most of the time is spent in version A.

The ExOS `select()` call basically consists of four parts:

1. Check if wake-up conditions are already met, and return immediately if they are.
2. Construct a wake-up predicate.
3. Install the wake-up predicate.
4. Sleep until the wake-up condition is met.

Further instrumentation of the `select()` call reveals that just constructing and installing the predicates in the call consumes twice as much time as is spent in the `read()` and `write()` calls.

The conclusion here is that while everything seems to indicate that while wake-up predicates is a powerful mechanism that has the potential to improve overall system performance, it must be used with care.

If the kernel could cache predicates, it seems that the program in figure 8 could be run with only a single installation of a wake-up predicate, but just caching will not solve the problem in the general case, only the case where identical wake-up predicates are repeatedly installed.

It would be interesting to build an alternate mechanism that do not compile the wake-up predicates to machine code, and compare the performance to the current implementation. This experiment has, however, not yet been performed.

A different approach that would probably not require changing the exokernel would be to re-design the way the mechanism is used. The current implementation will compile hard coded tests. This can be changed to allow the values tested for to be changed, without requiring the predicates to be recompiled and reinstalled. The test for a given resource, e.g. file, pipe or timer, would be installed the first time the resource was used as a wake-up condition. Subsequently, only the parameters would have to be changed.

A parameter could be used to toggle a certain test on and off, allowing previously used wake-up predicates to remain installed, but dormant, when testing for another condition. This would make the wake-up predicates run slightly slower, but offset against the cost of installing a predicate, this is very likely to be a win.

If predictability is important, the wake-up predicate could be installed ahead of time, i.e. when opening a file, rather than when first used, to make the time of activating a test equal regardless if it is the first or subsequent time a specific test is used.

The biggest downside of a scheme like the one outlined above is that it would require the library OS to keep track of what wake-up predicates it has installed, and make sure they are removed when the resource is no longer used.

5 Conclusions and results

In this section we will present measurements and our conclusions from the work we have done so far. In section 5.1 we discuss the port of the Erlang system, and in section 5.2 we present our conclusions.

5.1 The Erlang system on an exokernel

We have ported the Erlang system (JAM 4.6.3) to the XOK exokernel, running it on top of ExOS, the 4.4BSD UNIX-compatibility library OS. The effort needed to port the system was comparable to what would be needed to the system to a previously unsupported UNIX version. The performance of the Erlang system running on XOK is comparable to the performance we get from running it on OpenBSD and Linux. As can be seen in table 2, the difference between the OpenBSD and Linux system is greater than the difference between the XOK/ExOS version and OpenBSD.

System	**Estones**
Linux 2.0.33	6376
OpenBSD 2.2.0	5557
XOK/ExOS 1.0.2	5153

Table 2: Results from the Estone Erlang benchmark suite, running Erlang 4.6.3 system on a Pentium II/233 system. Higher Estone value is better.

The Linux and OpenBSD implementations of UNIX abstractions are more refined than the ones in ExOS, as was discussed in section 3.1 and is discussed in paper [33]. Table 3 show a detailed comparison between the OpenBSD and XOK/ExOS benchmarks. As can be seen, the results vary between favoring OpenBSD and XOK/ExOS, or being equal. The one major discrepancy is for the Port I/O benchmark, where XOK/ExOS has less than half the performance of OpenBSD. In fact, had the results for XOK/ExOS of this benchmark been equal to OpenBSD, the XOK/ExOS benchmark result had been superior to that of OpenBSD.

Benchmark	**OpenBSD**	**XOK/ExOS**	**Difference**
List manipulation	335	324	3.3%
Small messages	549	546	0.5%
Medium messages	514	529	-2.9%
Huge messages	138	143	-3.5%
Pattern matching	204	200	2.0%
Traverse	154	150	2.6%
Port I/O	944	432	**118.5%**
Work with large dataset	129	124	4.0%
Work with large local dataset	137	131	4.5%
Alloc and dealloc	117	118	-0.9%
BIF dispatch	583	590	-1.2%
Binary handling	230	254	-9.5%
Ets datadictionary	594	644	-7.8%
Generic server (with timeout)	327	376	-13.1%
Small Integer arithmetics	165	164	0.6%
Float arithmetics	38	38	0.0%
Function calls	236	236	0.0%
Timers	124	115	7.8%
Links	39	39	0.0%

Table 3: Detailed comparison between the benchmarks of the OpenBSD and XOK/ExOS versions of the Erlang system. The unit is Estones, and the difference column indicates how much faster than the XOK/ExOS version the OpenBSD version is.

The discrepancy for the Port I/O benchmark inspired the detailed analysis in section 4.6.12, in which the reason for the bad performance of the XOK/ExOS version of Erlang is explained.

5.2 Conclusions from the port of Erlang to XOK/ExOS

The fact that Erlang could be moved to an exokernel with a reasonable effort, and without any significant performance loss, tells us that the flexibility we buy does not cost us very much in terms of work or performance. Thus, the first step to an optimized version, moving the program to an exokernel based system, can be taken at a low cost. Subsequent steps may require a larger effort, but those steps should be considered in terms of the expected performance gains only — the cost of moving moving the program does not need to be amortized over many enhancements.

5.3 Suggestions for Erlang improvements

As a first step towards increased performance of the Erlang system on XOK/ExOS, the ExOS UNIX emulation library OS should be improved until it is on par with existing UNIX implementations. This work has already started at MIT. One of the goals of work on ExOS version 2.0[33] is to match and surpass the performance of contemporary UNIX implementations.

Further improvements should be done bottom up. While there are some opportunities for improvements that would require changes in the Erlang interpreter, these should be put off until those improvements that do not require changes in the interpreter have been implemented. The further from the Erlang system changes are made, the better, since this will reduce the number and scope of the changes made to the Erlang system. The changes we suggest below have been put into four categories, depending on how visible they will be, and are as follows:

5.3.1 Transparent changes

The exact OS semantics needed by the Erlang runtime system should be examined, and the implementation of these semantics in ExOS should be tuned to provide maximum performance for the Erlang system.

Typical changes here include streamlined implementations of I/O using packet filters and zero-copy mechanisms to get packet data into Erlang memory space directly as outlined in section 4.6.11, light weight IPC mechanism as outlined in section 4.6.8, things like the changes of the memory map outlined in section 4.6.1 and choosing algorithms that best fit the execution profile of Erlang. These changes stand a good chance of significantly boosting the performance of the system, using mainly the ability in an exokernel to tune implementations of OS functionality, rather than making use of "tricks".

The changes outlined here are all in the library OS and will not affect the API or perceived semantics of the operations, and will thus be completely transparent to the Erlang implementation.

5.3.2 Semi-transparent changes

There are many changes we can do that will for the most part be hidden to the Erlang system, but will require information and hints from the Erlang system to enable. These hints will mainly consist of a call from the Erlang system to inform the library OS what the Erlang system is currently up to, so that the library OS can adapt its behavior.

Changes here include changes in the virtual memory system to account for different memory usage patters, as outlined in section 4.6.3. E.g. we need some kind of hints from the Erlang system to inform the library OS that we are now inside a garbage collect, and should turn LRU off and switch to another page replacement scheme. It could also include other memory management techniques, such as dropping empty pages, but again this would require some cooperation from the Erlang system.

It may also include an improved thread implementation, but this will depend very much on the exact nature of the current Erlang internal threads. For the Erlang versions that use OS threads[21], using a exokernel thread implementation should be completely transparent.

All in all, these changes can be performed with a minimal impact on the Erlang system. There will be a few extra calls from the Erlang interpreter to the library OS, but these can be conditionally compiled in for only the exokernel version of Erlang.

5.3.3 Opaque changes

Finally, we have changes that will require more or less substantial changes to the Erlang system. In some cases, the information needed to optimize is not kept, and retaining it would require a redesign of part of the Erlang system.

Using memory mapping instead of copying, handling cache alignment of data and code, dealing with what to page out when we need to release physical memory are all examples of strategies that could increase performance, but are likely to require a major overhaul of the Erlang system to exploit.

Attempting this type of changes may give additional performance, but at the very probable cost of splitting the development into a "standard" OS version and an exokernel version.

5.4 Suggestions for Erlang annexes

A second strategy to exploit exokernel features is to provide annexes. An annex is an extension to Erlang so that low-level features can be exported directly to an Erlang program. An excellent example of this would be protocol implementation features. Erlang programs could be given direct access to the packet filter mechanism and receive/send calls that will place network packets into the Erlang buffers without requiring copies and send network packets directly from Erlang buffers. This will allow Erlang programmers to implement fast and efficient protocols.

Annexes can be provided for other low-level functionality such as direct disk access, interrupt handling and memory management.

These annexes can, from an Erlang perspective, be seen as an extension to the already wide array of function libraries. The small difference is that such an extension will not only link new functions to the Erlang program, but will also link new kernel functionality to the library OS.

5.5 Exokernels and traditional operating systems

The suggestions and strategies presented are possible to realize on non-exokernel systems. The C-FFS file system and the packet filtering mechanisms have been used on traditional operating system. However, doing this involves modifying the operating system to provide the new abstractions. The advantage of working on an exokernel system is that it gives an environment where it is simpler to experiment with new abstractions. Their very architecture encourages experimentation.

Open source operating systems such as Linux and the many BSD versions have become more widely accepted in the last years, and the availability of the source to these operating system enables programmers a fairly simple way to implement new kernel features without having to rely on an experimental operating system architecture.

References

[1] Bengt Ahlgren, Mats Björkman, and Per Gunningberg. Towards predictable ILP performance — controlling buffer cache effects. *The Australian Computer Journal*, 28(2):66–71, May 1996.

[2] Bengt Ahlgren, Mats Björkman, and Kjersti Moldeklev. The performance of a no-copy API for communication. In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, August 1995.

[3] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler activations: Effective kernel support for the user level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 95–109, October 1991.

[4] J.L. Armstrong, B.O. Däcker, S.R. Virding, and M.C. Williams. Implementing a functional language for highly parallel realtime applications. In *Proc. Software Engineering for Telecommunication Switching Systems*, 1992.

[5] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang.* Prentice Hall, 1993.

[6] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. PATHFINDER: A pattern-based packet classifier. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 115–123, November 1994.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[8] A. Chankhunthod, P.B. Danzig, C. Neerdaels, M.F. Schwartz, and K.J. Worrell. A hierarchical internet object cache. In *Proceedings of 1996 USENIX Technical Conference*, pages 153–163, January 1996.

[9] D. R. Cheriton. An experiment using registers for fast message-based interprocess communication. *Operating Systems Review*, October 1984.

[10] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP: Design, Implementation, and Internals*, volume II. Prentice-Hall, second edition, 1991. ISBN 0-13-472242-6.

[11] Intel Corporation. *Intel386 Family Binary Compatibility Specification 2.* McGraw-Hill, March 1992. ISBN 0-07-031219-2.

[12] D. R. Engler, M. F. Kaashoek, and J. O'Toole. The operating system kernel as a secure programmable machine. In *Proceedings of the Sixth SIGOPS European Workshop*, pages 62–67, September 1994.

[13] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. Exokernel: An operating system architecture for application-level resource managerment. In *Proceedings of the Fifteenth Symposium on Operating System Principles*, pages 251–266, December 1995.

[14] D.R. Engler and M.F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of ACM SIGCOMM 1996*, pages 53–59, August 1996.

[15] Linus Torvalds et al. The linux kernel. `http://www.kernel.org/`.

[16] Bryan Ford, Godmar Back, Greg Benson, Jay Leprau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the Sixteenth Symposium on Operating System Principles*, October 1997.

[17] Bryan Ford, K. Van Maren, Jay Lepreau, S. Clawson, B. Robinson, and Jeff Turner. The FLUX OS toolkit: Reusable components for OS implementation. In *Proc. of Sixth Workshop on Hot Topics in Operating Systems*, pages 14–19, May 1997.

[18] G. Ganger and M.F. Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *Proceedings of the 1997 USENIX Technical Conference*, pages 1–18, 1997.

[19] D. Goloub, R. Dean, A. Forin, and R. Rashid. UNIX as an application program. In *USENIX 1990 Summer Conference*, pages 87–95, June 1990.

[20] Bogdan Hausman. Turbo erlang: Approaching the speed of C. In *Implementations of Logic Programming Systems*, pages 119–135. Kluwer Academic Publishers, 1994.

[21] Pekka Hedqvist. A parallel and multithreaded Erlang implementation. Master's thesis, Uppsala University, Computing Science Department, June 1998.

[22] W.C. Hsieh, M.F. Kaashoek, and W.E. Weihl. The persistent relevance of IPC performance: New techniques for reducing the IPC penalty. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 186–190, October 1993.

[23] Intel Corporation. *Pentium II Processor at 233 MHz, 266 MHz, 300 MHz, and 333 MHz*, January 1998. Order number 243335-003.

[24] Erik Johansson, Christer Jonsson, Thomas Lindgren, Johan Bevemyr, and Håkan Millroth. A pragmatic approach to compilation of Erlang. UPMAIL Technical Report 136, Computing Science Department, Uppsala University, July 1997. ISSN 1100–0686.

[25] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Sixteenth Symposium on Operating System Principles*, October 1997.

[26] M.F. Kaashoek, D.R. Engler, D.H. Wallach, and G. Ganger. Server operating systems. In *SIGOPS European Workshop*, pages 141–148. ACM, September 1996.

[27] K. Mackenzie, J. Kubiatowicz, M. Frank, W. Lee, V. Lee, A. Agarwal, and M.F. Kaashoek. UDM: User direct messaging for general-purpose multiprocessing. Technical memo MIT/LCS/TM-556, Massachusetts Institute of Technology, March 1996.

[28] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Technical Conference Proceedings*, pages 259–269, 1993.

[29] Marshall Kirk McKusic, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System.* Addison-Wesley, 1996. ISBN 0-201-54979-4.

[30] Sun Microsystems. Sun collaborates with Linux community to make Linux available for UltraSPARC systems. Press release, December 1998.

[31] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, November 1987.

[32] The Open Telecom Platform Project. *Erlang System/OTP 4.5: Development Environment Reference Manual.* Ericsson Software Technology AB, Erlang Systems, second edition, 1997. EN/LZ 151 248 R2.

[33] Héctor Manuel Briceño Pulido. Decentralizing UNIX abstractions in the exokernel architecture. Master's thesis, Massachusetts Institute of Technology, February 1997.

[34] Ronald Joe Record, Michael Hopkirk, and Steve Ginzburg. Linux emulation for SCO. In *Proceedings of the 1998 USENIX Annual Technical Conference*, June 1998.

[35] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 213–228, October 1996.

[36] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts.* Addison-Wesley, fourth edition, 1994. ISBN 0-201-59292-4.

[37] William Stallings. *Operating Systems: Internals and Design Principles.* Prentice-Hall, third edition, 1998.

[38] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.

[39] C. A. Waldspurger and W. E. Weihl. Stride scheduling: deterministic proportional-share resource management. Technical Memorandum MIT/LCS/TM528, Massachusetts Institute of Technology, June 1995.

[40] M. Yahara, B. Bershad, C. Maeda, and E. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Proceedings of the Winter 1994 USENIX Conference*, 1994.