The JX Operating System

Michael Golm, Meik Felser, Christian Wawersich, Jürgen Kleinöder
University of Erlangen-Nürnberg
Dept. of Computer Science 4 (Distributed Systems and Operating Systems)
Martensstr. 1, 91058 Erlangen, Germany
{golm, felser, wawersich, kleinoeder}@informatik.uni-erlangen.de

Abstract

This paper describes the architecture and performance of the JX operating system. JX is both an operating system completely written in Java and a runtime system for Java applications.

Our work demonstrates that it is possible to build a complete operating system in Java, achieve a good performance, and still benefit from the modern software-technology of this object-oriented, type-safe language. We explain how an operating system can be structured that is no longer build on MMU protection but on type safety.

JX is based on a small microkernel which is responsible for system initialization, CPU context switching, and low-level protection-domain management. The Java code is organized in components, which are loaded into domains, verified, and translated to native code. Domains can be completely isolated from each other.

The JX architecture allows a wide range of system configurations, from fast and monolithic to very flexible, but slower configurations.

We compare the performance of JX with Linux by using two non-trivial operating system components: a file system and an NFS server. Furthermore we discuss the performance impact of several alternative system configurations. In a monolithic configuration JX achieves between about 40% and 100% Linux performance in the file system benchmark and about 80% in the NFS benchmark.

1 Introduction

The world of software production has dramatically changed during the last decades from pure assembler programming to procedural programming to object-oriented programming. Each step raised the level of abstraction and increased programmer productivity. Operating systems, on the other hand, remained largely unaffected by this process. Although there have been attempts to build object-oriented or object-based operating systems (Spring [27], Choices [10], Clouds [17]) and many operating systems internally use object-oriented concepts, such as vnodes [31], there is a growing divergence between application programming and operating system programming. To close this semantic gap between the applications and the OS interface a large market of middleware systems has emerged over the last years. While these systems hide the ancient nature of operating systems, they introduce many layers of indirection with several performance problems.

While previous object-oriented operating systems demonstrated that it is possible and beneficial to use object-orientation, they also made it apparent that it is a problem when implementation technology (object orientation) and protection mechanism (address spaces) mismatch. There are usually fine-grained "language objects" and large-grained "protected objects". A well-known project that tried to solve this mismatch by providing object-based protection in hardware was the Intel iAPX/432 processor [37]. While this project is usually cited as a failure of object-based hardware protection, an analysis [14] showed that with a slightly more mature hardware and compiler technology the iAPX/432 would have achieved a good performance.

We believe that an operating system based on a dynamically compiled, object-oriented intermediate code, such as the Java bytecode, can outperform traditional systems, because of the many compiler optimizations (i) that are only possible at a late time (e.g., inlining virtual calls) and (ii) that can be applied only when the system environment is exactly known (e.g., cache optimizations [12]).

Using Java as the foundation of an operating system is attractive, because of its widespread use and features, such as interfaces, encapsulation of state, and automatic memory management, that raise the level of abstraction and help to build more robust software in less time.

To the best of our knowledge JX is the first Java operating system that has *all* of the following properties:

- The amount of C and assembler code is minimal to simplify the system and make it more robust.
- Operating system code and application code is separated in protection domains with strong isolation between the domains.
- The code is structured into components, which can be collocated in a single protection domain or dislocated in separate domains without touching the component code. This

reusability across configurations enables to adapt the system for its intended use, which may be, for example, an embedded system, desktop workstation, or server.

- Performance is in the 50% range of monolithic UNIX performance for computational-intensive OS operations. The difference becomes even smaller when I/O from a real device is involved.

Besides describing the JX system, the contribution of this paper consists of the first performance comparison between a Java OS and a traditional UNIX OS using real OS operations. We analyze two costs: (i) the cost of using a type-safe language, like Java, as an OS implementation language and (ii) the cost of extensibility.

The paper is structured as follows: In Section 2 we describe the architecture of the JX system and illustrate the cost of several features using micro benchmarks. Section 3 describes two application scenarios and their performance: a file system and an NFS server. Section 4 describes tuning and configuration options to refine the system and measures their effect on the performance of the file system. Section 5 concludes and gives directions for future research.

2 JX System Architecture

The majority of the JX system is written in Java. A small microkernel, written in C and assembler, contains the functionality that can not be provided at the Java level (system initialization after boot up, saving and restoring CPU state, low-level protection-domain management, and monitoring).

Figure 1 shows the overall structure of JX. The Java code is organized in components (Sec. 2.4) which are loaded into domains (Sec. 2.1), verified (Sec. 2.6), and translated to native code (Sec. 2.6). Domains encapsulate objects and threads. Communication between domains is handled by using portals (Sec. 2.2).

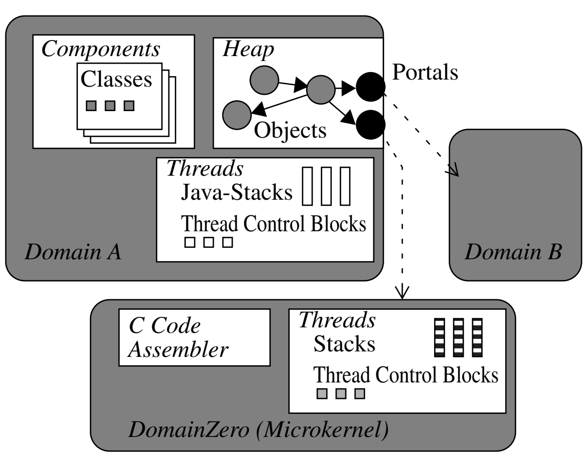

Figure 1: Structure of the JX system

The microkernel runs without any protection and therefore must be trusted. Furthermore, a few Java components must also be trusted: the code verifier, the code translator, and some hardware-dependent components (Sec. 2.7). These elements are the minimal *trusted computing base* [19] of our architecture.

2.1 Domains

The unit of protection and resource management is called a *domain*. All domains, except DomainZero, contain 100% Java code.

DomainZero contains all the native code of the JX microkernel. It is the only domain that can not be terminated. There are two ways how domains interact with DomainZero. First, explicitly by invoking services that are provided by DomainZero. One of these services is a simple name service, which can be used by other domains to export their services by name. Secondly, implicitly by requesting support from the Java runtime system; for example, to allocate an object or check a downcast.

Every domain has its own heap with its own garbage collector (GC). The collectors run independently and they can use different GC algorithms. Currently, domains can choose from two GC implementations: an exact, copying, non-generational GC or a compacting GC.

Every domain has its own threads. A thread does not migrate between domains during inter-domain communication. Memory for the thread control blocks and stacks is allocated from the domain's memory area.

Domains are allowed to share code - classes and interfaces - with other domains. But each domain has its own set of static fields, which, for example, allows each domain to have its own System.out stream.

2.2 Portals

Portals are the fundamental inter-domain communication mechanism. The portal mechanism works similar to Java's RMI [43], making it easy for a Java programmer to use it. A portal can be thought of as a proxy for an object that resides in another domain and is accessed using remote procedure call (RPC).

An entity that may be accessed from another domain is called *service*. A service consists of a normal object, which must implement a portal interface, an associated *service thread*, and an initial portal. A service is accessed via a *portal*, which is a remote (proxy) reference. Portals are capabilities [18] that can be copied between domains. The service holds a reference counter, which is incremented each time, the portal is duplicated. A domain that wants to offer a service to other domains can register the service's portal at a name server.

When a thread invokes a method at a portal, the thread is blocked and execution is continued in the service thread. All parameters are deep copied to the target domain. If a parameter is itself a portal, a duplicate of the portal is created in the target domain.

Copying parameters poses several problems. It leads to duplication of data, which is especially problematic when a large transitive closure is copied. To avoid that a domain is flooded with parameter objects, a per-call quota for parameter data is used in JX. Another problem is that the object identity is lost during copying. Although parameter copying can be avoided in a single address space system, and even for RPC between address spaces by using shared communication buffers [7], we believe that the advantages of copying outweigh its disadvantages. The essential advantage of copying is a nearly complete isolation of the two communicating protection domains. The only time where two domains can interfere with each other is during portal invocation. This makes it easy to control the security of the system and to restrict information flow. Another advantage of the copying semantics is, that it can be extended to a distributed system without much effort.

In practice, copying posed no severe performance problems, because only small data objects are used as parameters. Objects with a large transitive closure in most cases are server objects and are accessed using portals. Using them as data objects often is not intended by the programmer.

As an optimization the system checks whether the target domain of a portal call is identical to the current domain and executes the call as a function invocation without thread switch and parameter copy.

When a portal is passed as a parameter in a portal call, it is passed by-reference. As a convenience to the programmer the system also allows an object that implements a portal interface to be passed like a portal. First it is checked, whether this object already is associated with a service. In this case, the existing portal is passed. Otherwise, a service is launched by creating the appropriate data structures and starting a new service thread. This mechanism allows the programmer to completely ignore the issue of whether the call is crossing a domain border or not. When the call remains inside the domain the object is passed as a normal object reference. When the call leaves the domain, the object automatically is promoted to a service and a portal to this service is passed.

When a portal is passed to the domain in which its service resides, a reference to the service object is passed instead of the portal.

When two domains want to communicate via portals they *must* share some types. These are at least the portal interface and the parameter types. When a domain obtains a portal, it is checked whether the correct interface is present.

System	IPC (cycles)
L4Ka (PIII, 450MHz) [32]	818
Fiasco/L4 (PIII 450 MHz) [42]	2610
J-Kernel (LRMI on MS-VM, PPro 200MHz) [28]	440
Alta/KaffeOS (PII 300 MHz) [5]	27270
JX (PIII 500MHz)	650

Table 1: IPC latency (round-trip, no parameters)

Each time a new portal to a service is created a reference counter in the service control block is incremented. It is decremented when a portal is collected as garbage or when the portal's domain terminates. When the count reaches zero the service is deactivated and all associated resources, such as the service thread, are released.

Table 1 shows the cost of a portal invocation and compares it with other systems. This table contains very different systems with very different IPC mechanisms and semantics. The J-Kernel IPC, for example, does not even include a thread switch.

Fast portals. Several portals which are exported by DomainZero are *fast portals*. A fast portal invocation looks like a normal portal invocation but is executed in the caller context (the caller thread) by using a function call - or even by inlining the code (see also Sec. 4.2.2). This is generally faster than a normal portal call, and in some cases it is even necessary. For example, DomainZero provides a portal with which the current thread can yield the processor. It would make no sense to implement this method using the normal portal invocation mechanism.

2.3 Memory objects

An operating system needs an abstraction to represent large amounts of memory. Java provides byte arrays for this purpose. However, arrays have several shortcomings, that make them nearly unsuitable for our purposes. They are not accessed using methods and thus the set of allowed operations is fixed. It is, for example, not possible to restrict access to a memory region to a read-only interface. Furthermore, arrays do not allow revocation and subrange creation - two operations that are essential to pass large memory chunks without copying.

To overcome these shortcomings we developed another abstraction to represent memory ranges: *memory objects*. Memory objects are accessed like normal objects via method invocations. But such invocations are treated specially by the translator: they are replaced by the machine instructions for the memory access. This makes memory access as fast as array access.

Memory objects can be passed between domains like portals. The memory that is represented by a memory object is not copied when the memory object is passed to another domain. This way, memory objects implement shared memory.

Access to a memory range can be revoked. For this purpose all memory portals that represent the same range of memory contain a reference to the same central data structure in DomainZero. Among other information this data structure contains a valid flag. The revocation method invalidates the original memory object by clearing the valid flag and returns a new one that represents the same range of memory. Memory is not copied during revocation but all memory portals that previously represented this memory become invalid.

When a memory object is passed to another domain, a reference counter, which is maintained for every memory range, is incremented. When a memory object - which, in fact, is a portal or proxy for the real memory - is garbage collected, the reference counter is decremented. This happens also for all memory objects of a domain that is terminated. To correct the reference counts the heap must be scanned for memory objects before it is released.

ReadOnlyMemory. ReadOnlyMemory is equivalent to Memory but it lacks all the methods that modify the memory. A ReadOnlyMemory object can not be converted to a Memory object.

DeviceMemory. DeviceMemory is different from Memory in that it is not backed by main memory: It is usually used to access the registers of a device or to access memory that is located on a device and mapped into the CPU's address space. The translator knows about this special use and does not reorder accesses to a DeviceMemory. When a DeviceMemory is garbage collected the memory is not released.

2.4 Components

All Java code that is loaded into a domain is organized in components. A component contains the classes, interfaces, and additional information; for example, about dependencies from other components or about the required scheduling environment (preemptive, nonpreemptive).

Reusability. An overall objective of object orientation and object-oriented operating systems is code reuse. JX has all the reusability benefits that come with object orientation. But there is an additional problem in an operating system: the protection boundary. To call a module across a protection boundary in most operating system is different from calling a module inside the own protection domain. Because this difference is a big hindrance on the way to reusability, this problem has already been investigated in the microkernel context [22].

Our goal was a reuse of components in different configurations without code modifications. Although the portal mechanism was designed with this goal the programmer must keep several points in mind when using a portal. Depending on whether the called service is located inside the domain or in another domain there are a few differences in behavior. Inside a domain normal objects are passed by reference. When a domain border is crossed, parameters are passed by copy. To write code that works in both settings the programmer must not rely on either of these semantics. For example, a programmer relies on the reference semantics when modifying the parameter object to return information to the caller; and the programmer relies on the copy semantics when modifying the parameter object assuming this modification does not affect the caller.

In practice, these problems can be relieved to a certain extent by the automatic promotion of portal-capable objects to services as described in Section 2.2. By declaring all objects that are entry points into a component as portals a reference semantics is guaranteed for these objects.

Dependencies. Components may depend on other components. We say that component B has an *implementation dependence* on component A, if the method implementations of B use classes or interfaces from A. Component B has an *interface dependence* on component A if the method signatures of B use classes or interfaces from A or if a class/interface of B is a subclass/subinterface of a class/interface of A, or if a class of B implements an interface from A, or if a non-private field of a class of B has as its type a class/interface from A.

Component dependencies must be non-cyclic. This requirement makes it more difficult to split existing applications into components (Although they can be used as one component!). A cyclic dependency between components usually is a sign of bad design and should be removed anyway. When a cyclic dependency is present, it must be broken by changing the implementation of one component to use an interface from an unrelated component while the other class implements this interface. The components then both depend on the unrelated component but not on each other. The dependency check is performed by the verifier and translator.

We used Sun's JRE 1.3.1_02 for Linux to obtain the transitive closure of the depends-on relation starting with java.lang.Object. The implementation dependency consists of 625 classes; the interface dependency consists of 25 classes. This means, that each component that uses the Object class (i.e., every component) depends on at least 25 classes from the JDK. We think, that even 25 classes are a too broad foundation for OS components and define a compatibility relation that allows to exchange the components.

Compatibility. The whole system is build out of components. It is necessary to be able to improve and extend one component without changing all components that depend on this component. Only a component B that is compatible to component A can be substituted for A. A component B is binary compatible to a component A, if

- for each class/interface C_A of A there is a corresponding class/interface C_B in component B
- class/interface C_B is binary compatible to class C_A according to the definition given in the "Java Language Specification" [26] Chapter 13.

When a binary compatible component is also a semantic superset of the original component, it can be substituted for the original component without affecting the functionality of the system.

JDK. The JDK is implemented as a normal component. Different implementations and versions can be used. Some classes of the JDK must access information that is only available in the runtime system. The class Class is an example. This information is obtained by using a portal to DomainZero. In other words, where a traditional JDK implementation would use a native method, JX uses a normal method that invokes a service of DomainZero via a portal. All of our current components use a JDK implementation that is a subset of a full JDK and, therefore, can also be used in a domain that loads a full JDK.

Interface invocation. Non-cyclic dependencies and the compilation of whole components opens up a way to compile very efficient interface invocations. Usually, interface invocations are a problem because it is not possible to use a fixed index into a method table to find the interface method. When different classes implement the interface, the method can be at different positions in their method tables. There exists some work to reduce the overhead in a system that does not impose our restrictions [1]. In our translator we use an approach that is similar to selector coloring [20]. It makes interface invocations as fast as method invocations at the cost of (considerably) larger method tables.

The size of the x86 machine code in the complete JX system is 1,010,752 bytes, which was translated from 230,421 bytes of bytecode. The method tables consume 630,388 bytes. These numbers show that it would be worthwhile to use a compression technique for the method tables or a completely different interface invocation mechanism. One should keep in mind, that a technique as described in [1] has an average-case performance near to a virtual invocation, but it may be difficult to analyze the worst-case behavior of the resulting system, because of the use of a caching data structure.

2.5 Memory management

Protection is based on the use of a type-safe language. Thus an MMU is not necessary. The whole system, including all applications, runs in one physical address space. This makes the system ideally suited for small devices that lack an MMU. But it also leads to several problems. In a traditional system fragmentation is not an issue for the user-level memory allocator, because allocated, but unused memory, is paged to disk. In JX unused memory is wasted main memory. So we face a similar problem as kernel memory allocators in UNIX, where kernel memory usually also is not paged and therefore limited. In UNIX a kernel memory allocator is used for vnodes, proc structures, and other small objects. In contrast to this the JX kernel does not create many small objects. It allocates memory for a domain's heap and the small objects live in the heap. The heap is managed by a garbage collector. In other words, the JX memory management has two levels, a global management, which must cope with large objects and avoid fragmentation, and a domain-local garbage-collected memory. The global memory is managed using a bitmap allocator [46]. This allocator was easy to implement, it automatically joins free areas, and it has a very low memory footprint: Using 1024-byte blocks and managing about 128MBytes or 116977 blocks, the overhead is only 14622 bytes or 15 blocks or 0.01 percent. However, it should not be too complicated to use a different allocator.

To give up the MMU means that several of their responsibilities (besides protection) must be implemented in software. One example is the stack overflow detection, another one the null pointer detection. Stack overflow detection is implemented in JX by inserting a stack size check at the beginning of each method. This is feasible, because the required size of a stack frame is known before the method is executed. The size check has a reserve, in case the Java method must trap to a runtime function in DomainZero, such as *checkcast*. The null pointer check currently is implemented using the debug system of the Pentium processor. It can be programmed to raise an exception when data or code at address zero is accessed. On architectures that do not provide such a feature, the compiler inserts a null-pointer check before a reference is used.

A domain has two memory areas: an area where objects may be moved and an area where they are fixed. In the future, a single area may suffice, but then all data structures that are used by a domain must be movable. Currently, the fixed area contains the code and class information, the thread control blocks and stacks. Moving these objects requires an extension of the system: all pointers to these objects must be known to the GC and updated; for example, when moving a stack, the frame pointers must be adjusted.

2.6 Verifier and Translator

The verifier is an important part of JX. All code is verified before it is translated to native code and executed. The verifier first performs a standard bytecode verification [48]. It then verifies an upper limit for the execution times of the interrupt handlers and the scheduler methods (Sec. 2.8) [2].

The translator is responsible for translating bytecode to machine code, which in our current system is x86 code. Machine code can either be allocated in the domain's fixed memory or in DomainZero's fixed memory. Installing it in DomainZero allows to share the code between domains.

2.7 Device Drivers

An investigation of the Linux kernel has shown that most bugs are found in device drivers [13]. Because device drivers will profit most from being written in a type-safe language, all JX device drivers are written in Java. They use DeviceMemory to access the registers of a device and the memory that is available on a device; for example, a frame buffer. On some architectures there are special instructions to access the I/O bus; for example, the in and out processor instructions of the x86. These instructions are available via a fast portal of DomainZero. As other fast portals, these invocations can be inlined by the translator.

DMA. Most drivers for high-throughput devices will use busmaster DMA to transfer data. These drivers, or at least the part that accesses the DMA hardware, must be trusted.

Interrupts. Using a portal of DomainZero, device drivers can register an object that contains a handleInterrupt method. An interrupt is handled by invoking the handleInterrupt method of the previously installed interrupt handler object. The method is executed in a dedicated thread while interrupts on the interrupted CPU are disabled. This would be called a *first-level interrupt handler* in a conventional operating system. To guarantee that the handler can not block the system forever, the verifier checks all classes that implement the InterruptHandler interface. It guarantees that the handleInterrupt method does not exceed a certain time limit. To avoid undecidable problems, only a simple code structure is allowed (linear code, loops with constant bound and no write access to the loop variable inside the loop). A handleInterrupt method usually acknowledges the interrupt at the device and unblocks a thread that handles the interrupt asynchronously.

We do not allow device drivers to disable interrupts outside the interrupt handler. Drivers usually disable interrupts as a cheap way to avoid race conditions with the interrupt handler. Code that runs with interrupts disabled in a UNIX kernel is not allowed to block, as this would result in a deadlock. Using locks also is not an option, because the interrupt handler - running with interrupts disabled - should not block. We use the abstraction of an AtomicVariable to solve these problems. An AtomicVariable contains a value, that can be changed and accessed using set and get methods. Furthermore, it provides a method to atomically compare its value with a parameter and block if the values are equal. Another method atomically sets the value and unblocks a thread. To guarantee atomicity the implementation of AtomicVariable currently disables interrupts on a uniprocessor and uses spinlocks on a multiprocessor. Using AtomicVariables we implemented, for example, a producer/consumer list for the network protocol stack.

2.8 Scheduling

There is a common experience that the scheduler has a large impact on the system's performance. On the other hand, no single scheduler is perfect for all applications.

Instead of providing a configuration interface to the scheduler we follow our methodology of allowing a user to completely replace an implementation, in this case the scheduler. Each domain may also provide its own scheduler, optimized for its particular requirements.

The scheduler can be used in several configurations:

- First, there is a scheduler that is build into the kernel. This scheduler is only used for performance analysis, because it is written in C and can not be replaced at run time.
- The kernel can be compiled without the built-in scheduler. Then all scheduling decisions lead to the invocation of a scheduler implementation which is written in Java. In this configuration there is one (Java) scheduler that schedules all threads of all domains.
- The most common configuration, however, is a two-level scheduling. The global scheduler does not schedule threads, as in the previous configuration, but domains. Instead of activating an application thread, it activates the scheduler thread of a domain. This domain-local scheduler is responsible for selecting the next application thread to run. The global scheduler knows all domain-local schedulers and a domain-local scheduler has a portal to the global scheduler. On a multiprocessor there is one global scheduler per pro-

cessor and the domains posses a reference to the global schedulers of the processors on which they are allowed to run.

The global scheduler must be trusted by all domains. The global scheduler does not need to trust a domain-local scheduler. This means, that the global scheduler can *not* assume, that an invocation of the local scheduler returns after a certain time.

To prevent one domain monopolizing the processor, the computation can be interrupted by a timer interrupt. The timer interrupt leads to the invocation of the global scheduler. This scheduler first informs the scheduler of the interrupted domain about the pre-emption. It switches to the domain scheduler thread and invokes the scheduler's method preempted(). During the execution of this method the interrupts are disabled. An upper bound for the execution time of this method has been verified during the verification phase. When the method preempted() returns, the system switches back to the thread of the global scheduler. The global scheduler then decides, which domain to run next activates the domain-local scheduler using the method activated(). For each CPU that can be used by a domain the local scheduler of the domain has a CPU portal. It activates the next runnable thread by calling the method switchTo() at the CPU portal. The switchTo() method can only be called by a thread that runs on the CPU which is represented by the CPU portal. The global scheduler does not need to wait for the method activated() to finish. Thus, an upper time bound for method activated() is not necessary. This method makes the scheduling decision and it can be arbitrarily complex.

If a local scheduler needs smaller time-slices than the global scheduler, the local scheduler must be interrupted without being pre-empted. For this purpose, the local scheduler has a method interrupted() which is called before the time-slice is fully consumed. This method operates similar to the method activated().

Because our scheduler is implemented outside the microkernel and there are operations of the microkernel that affect scheduling, for example, thread handoff during a portal invocation, we face a similar situation as a user-level thread implementation on a UNIX-like system. A well-known solution are scheduler activations [3], which notify the user-level scheduler about events inside the kernel, such as I/O operations. JX uses a similar approach, although there are very few scheduling related operations inside the kernel. Scheduling is affected when a portal method is invoked. First, the scheduler of the calling domain is informed, that one thread performs a portal call. The scheduler can now delay the portal call, if there is any other runnable thread in this domain. But it can as well handoff the processor to the target domain. The scheduler of the service domain is notified of the incoming portal call and can either activate the service thread or let another thread of the domain run. Not being forced to schedule the service thread immediately is essential for the implementation of a non-preemptive domain-local scheduler.

This extra communication is not for free. The time of a portal call increases from 650 cycles (see Table 1) to 920-960 cycles if either the calling domain or the called domain is informed. If both involved domain schedulers are informed about the portal call the required time increases to 1180 cycles.

2.9 Locking and condition variables

Kernel-level locking. There are very few data structures that must be protected by locks inside DomainZero. Some of them are accessed by only one domain and can be locked by a domain-specific lock. Others, for example, the domain management data structures, need a global lock. Because the access to this data is very short, an implementation that disables interrupts on a uniprocessor and uses spinlocks on a multiprocessor is sufficient.

Domain-level locking. Domains are responsible for synchronizing access to objects by their own threads. Because there are no objects shared between domains there is no need for inter-domain locking of objects. Java provides two facilities for thread synchronization: mutex locks and condition variables. When translating a component to native code, an access to such a construct is redirected to a user-supplied synchronization class. How this class is implemented can be decided by the user. It can provide no locking at all or it can implement mutexes and condition variables by communicating with the (domain-local) scheduler. Every object can be used as a monitor (mutex lock), but very few actually are. To avoid allocating a monitor data structure for every object, traditional JVMs either use a hashtable to go from the object reference to the monitor or use an additional pointer in the object header. The hashtable variant is slow and is rarely used in today's JVMs. The additional pointer requires that the object layout must be changed and the object header be accessible to the locking system. Because the user can provide an own implementation, these two implementations, or a completely application-specific one, can be used.

Inter-domain locking. Memory objects allow sharing of data between domains. JX provides no special inter-domain locking mechanisms. When two domains want to synchronize, they can use a portal call. We did not need such a feature yet, because the code that passes memory between domains does it by explicitly revoking access to the memory.

3 Application Scenarios: Comparing JX to a Traditional Operating System

JX contains a file system component that is a port of the Linux ext2 file system to Java [45]. Figure 2 shows the configuration, where file system and buffer cache are cleanly separated into different components. The gray areas denote protection domains and the white boxes components. The file system uses the BufferCache interface to access disk blocks. To read and write blocks to a disk the buffer cache implementation uses a reference to a device that implements the BlockIO interface. The file system and buffer cache components do not use locking. They require a non-preemptive scheduler to be installed in the domain.

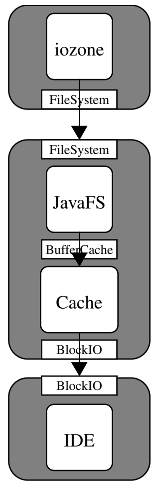

Figure 2: IOZone configuration

To evaluate the performance of JX we used two benchmarks: the IOZone benchmark [44] to assess file system performance and a home brewed *rate* benchmark to assess the performance of the network stack and NFS server. The rate benchmark sends *getattr* requests to the NFS server as fast as possible and measures the achievable request rate. As JX is a pure Java system, we can not use the original IOZone program, which is written in C. Thus we ported IOZone to Java. The JX results were obtained using our Java version and the Linux results were obtained using the original IOZone.

The hardware consists of the following components:

- The system-under-test: PIII 500MHz with 256 MBytes RAM and a 100 MBit/s 3C905B Ethernet card running Suse Linux 7.3 with kernel 2.4.0 or JX.
- The client for the NFS benchmark: a PIII 1GHz with a 100 MBit/s 3C905B Ethernet card running Suse Linux 7.3.
- A 100MBit/s hub that connects the two systems.

Figure 3 shows the results of running the IOZone reread benchmark on Linux.

Our Java port of the IOZone contains the write, rewrite, read, and reread parts of the original benchmark. In the following discussion we only use the reread part of the benchmark. The read benchmark measures the time to read a file by reading fixed-length records. The reread benchmark measures the time for a second read pass. When the file is smaller than the buffer cache all data comes from the cache. Once a disk access is involved, disk and PCI bus data transfer times dominate the result and no conclusions about the performance of JX can be drawn. To avoid these effects we only use the reread benchmark with a maximum file size of 512 KBytes, which means that the file completely fits into the buffer cache. The JX numbers are the mean of 50 runs of IOZone. The standard deviation was less than 3%. For time measurements on JX we used the Pentium timestamp counter which has a resolution of 2 ns on our system.

Figure 2 shows the configuration of the JX system when the IOZone benchmark is executed. Figure 4 shows the results of the benchmark. Figure 5 compares JX performance to the Linux performance. Most combinations of file size and record size give a performance between 20% and 50% of the Linux performance. Linux is especially good at reading a file using a small record size. The performance of this JX configuration is rather insensitive to the record size. We will explain how we improved the performance of JX in the next section.

Another benchmark is the rate benchmark, which measures the achievable NFS request rate by sending *getattr* requests to the NFS server. Figure 6 shows the domain structure of the NFS server: all components are placed in one domain, which is a typical configuration

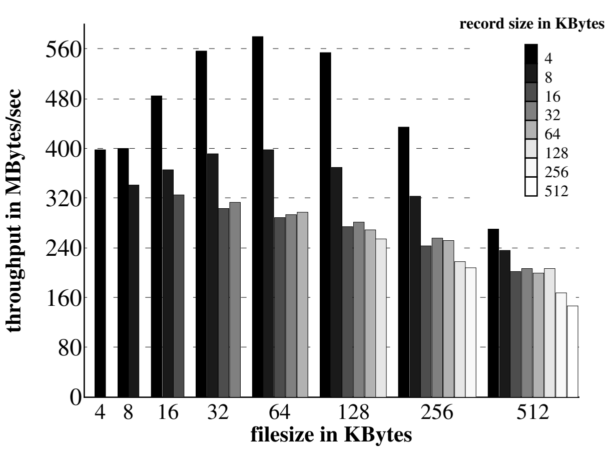

Figure 3: Linux IOZone performance

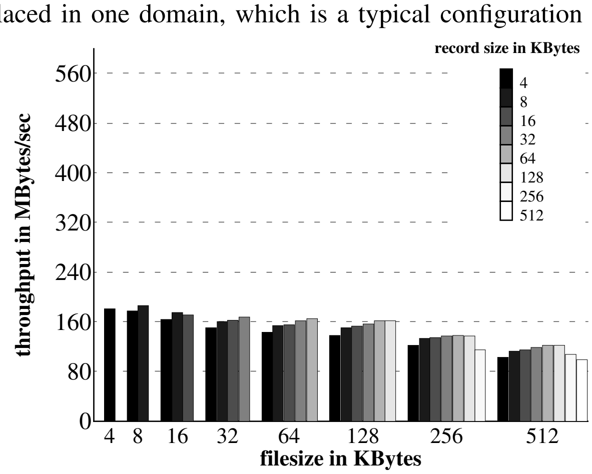

Figure 4: JX IOZone: multi-domain configuration

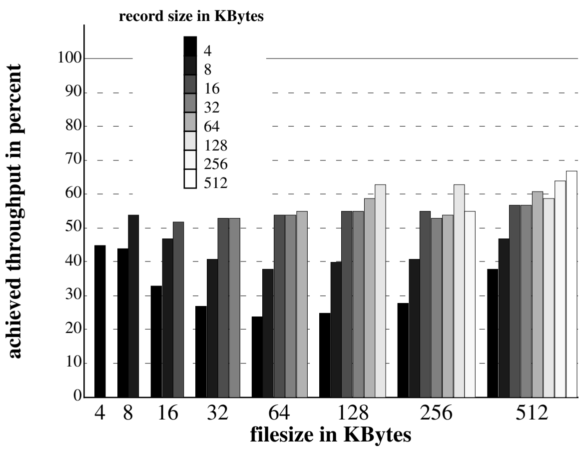

Figure 5: JX vs. Linux: multi-domain configuration

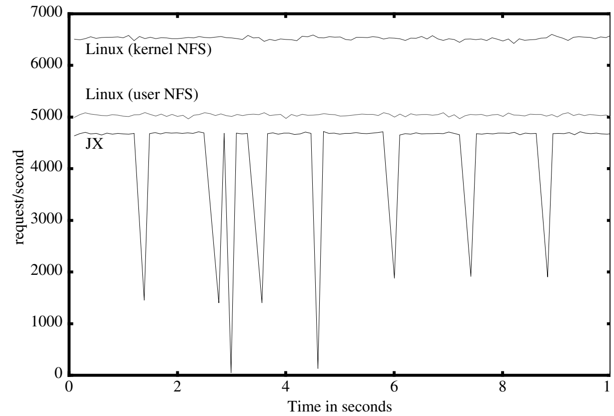

Figure 8: JX NFS performance (rate benchmark)

for a dedicated NFS server. Figure 8 shows the results of running the rate benchmark with a Linux NFS server (both kernel and user-level NFS) and with a JX NFS server. There are drops in the JX request rate that occur very periodically. To see what is going on in the JX NFS server, we collected thread switch information and created a thread activity diagram. Figure 7 shows this diagram. We see an initialization phase which is completed six seconds after startup. Shortly after startup a periodic thread (ID 2.12) starts, which is the interrupt handler of the real-time clock. But the important activity starts at about 17 seconds. The CPU is switched between "IRQThread11", "Etherpacket-Queue", "NFSProc", and "Idle" thread. This is the activity during the rate benchmark. Packets are received and put into a queue by the first-level interrupt handler of the network interface "IRQThread11" (ID 2.14). This unblocks the "Etherpacket-Queue" (ID 2.19), which processes the packet and finally puts it into a UDP packet queue. This unblocks the "NFSProc" (ID 2.27) thread, which processes the NFS packet and accesses the file system. This is done in the same thread, because the NFS component and the file system are collocated. Then a reply is sent and all threads block, which wakes up the "Idle" thread (ID 0.1). The sharp drops in the request rate of the JX NFS server in Figure 8 correspond to the GC thread (ID 2.1) that runs for about 100 milliseconds without being interrupted. It runs that long because neither the garbage collector nor the NFS server are optimized. Especially the RPC layer creates many objects during RPC packet processing. The GC is not interrupted, because it disables interrupts as a safety precaution in the current implementation. The pauses could be avoided by using an incremental GC [6], which allows the GC thread to run concurrently with threads that modify the heap.

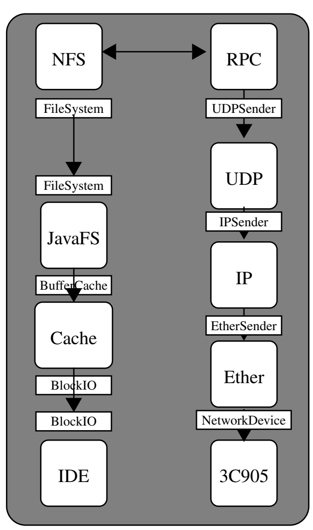

Figure 6: JX NFS configuration

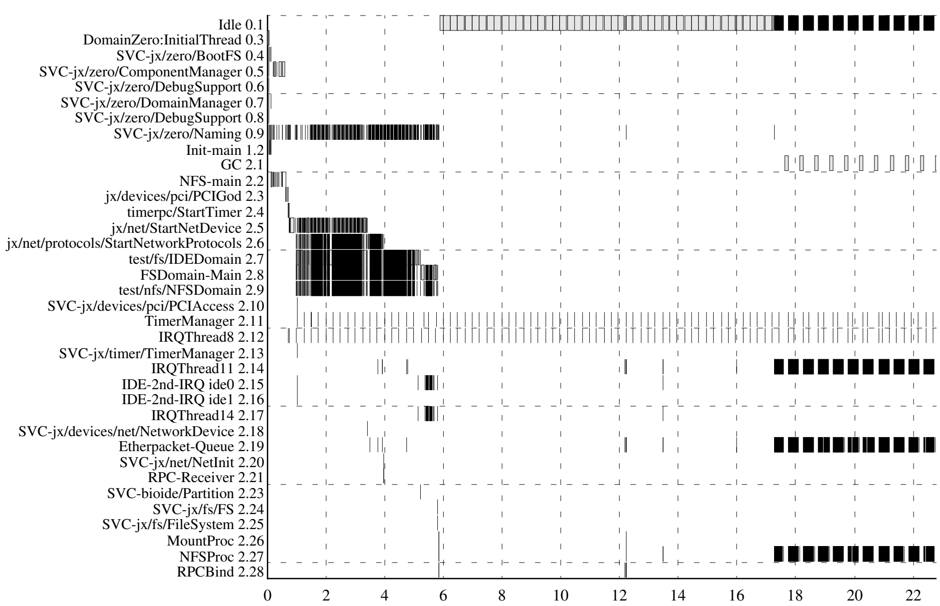

Figure 7: Thread activity during the rate benchmark

4 Optimizations

JX provides a wide range of flexible configuration options. Depending on the intended use of the system several features can be disabled to enhance performance.

Figures 9 through 14 show the results of running the Java IOZone benchmark on JX with various configuration options. These results are discussed in further detail below. The legend for the figures indicates the specific configuration options used in each case. The default configuration used in Figure 3 was MNNSCR, which means that the configuration options used were multi-domain, no inlining, no inlined memory access, safety checks enabled, memory revocation check by disabling interrupts, and a Java round-robin scheduler. At the end of this section we will select the fastest configuration and repeat the comparison to Linux.

The modifications described in this sections are pure configurations. Not a single line of code is modified.

4.1 Domain structure

How the system is structured into domains determines communication overheads and thus affects performance. For maximal performance, components should be placed in the same domain. This removes portal communication overhead. Figure 9 shows the improvement of placing all components into a single domain. The performance improvement is especially visible when using small record sizes, because then many invocations between the IOZone component and the file system component take place. The larger improvement in the 4KB file size / 4KB record size can be explained by the fact that the overhead of a portal call is relatively constant and the 4KB test is very fast, because it completely operates in the L1 cache. So the portal call time makes up a considerable part of the complete time. The contrary is true for large file sizes: the absolute throughput is lower due to processor cache misses and the saved time of the portal call is only a small fraction of the complete time. Within one file size the effect also becomes smaller with increasing record sizes. This can be explained by the decreasing number of performed portal calls.

4.2 Translator configuration

The translator performs several optimizations. This section investigates the performance impact of each of these optimizations. The optimizations are inlining, inlining of fast portals, and elimination of safety checks.

4.2.1 Inlining

One of the most important optimizations in an object-oriented system is inlining. We currently inline only non-virtual methods (final, static, or private). We plan to inline also virtual methods that are not overridden, but this would require a recompilation when, at a later time, a class that overrides the method is loaded into the domain. Figure 10 shows the effect of inlining.

4.2.2 Inlining of fast portals

A fast portal interface (see Sec. 2.2) that is known to the translator can also be inlined. To be able to inline these methods that are written in C or assembler the translator must know their semantics. Since we did not want to wire these semantics too deep into the translator, we developed a plugin architecture. A translator plugin is responsible for translating the invocations of the methods of a specific fast portal interface. It can either generate special code or fall back to the invocation of the DomainZero method.

We did expect a considerable performance improvement but as can be seen in Figure 11 the difference is very small. We assume, that these are instruction cache effects: when a memory access is inlined the code is larger than the code that is generated for a function call. This is due to range checks and revocation checks that must be emitted in front of each memory access.

4.2.3 Safety checks

Safety checks, such as stack size check and bounds checks for arrays and memory objects can be omitted on a per-domain basis. Translating a domain without checks is equivalent to the traditional OS approach of hoping that the kernel contains no bugs. The system is now as unsafe as a kernel that is written in C. Figure 12 shows that switching off safety checks can give a performance improvement of about 10 percent.

4.3 Memory revocation

Portals and memory objects are the only objects that can be shared between domains. They are capabilities and an important functionality of capabilities is revocation. Portal revocation is implemented by checking a flag before the portal method is invoked. This is an inexpensive operation compared to the whole portal invocation. Revocation of memory objects is more critical because the operations of memory objects - reading and writing the memory - are very fast and frequently used operations. The situation is even more involved, because the check of the revocation flag and the memory access have to be performed as an atomic operation. JX can be configured to use different implementations of this revocation check:

- **NoCheck**: No check at all, which means revocation is not supported.

Legend for all figures on this page:

record size in KBytes: 4, 8, 16, 32, 64, 128, 256, 512

Encoding of the measured configuration:

1. domain structure:	S (single domain), M (multi domain)
2. inlining:	I (inlining), N (no inlining)
3. memory access:	F (inlined memory access), N (no inlined memory access)
4. safety checks:	S (safety checks enabled), N (safety checks disabled)
5. memory revocation:	N (no memory revocation), C (disable interrupts), S (spinlock), A (atomic code)
6. scheduling:	C (microkernel scheduler), R (Java RR scheduler), I (Java RR invisible portals)

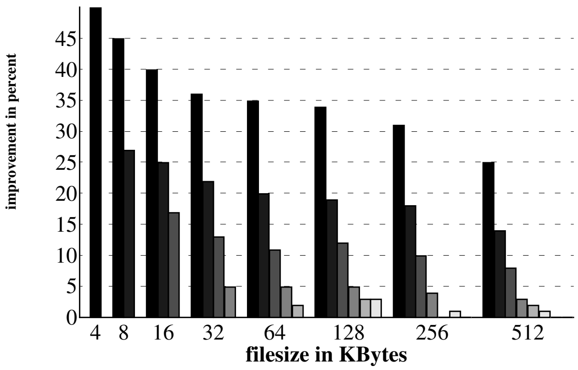

Figure 9: Domain structure: ***S***NNSCR vs. ***M***NNSCR

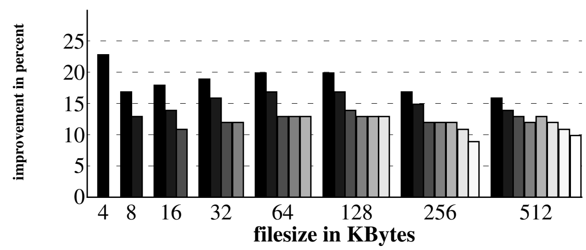

Figure 10: Inlining: S***I***NSCR vs. S***N***NSCR

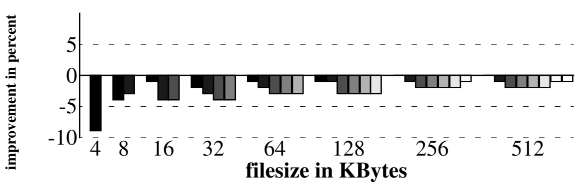

Figure 11: Memory access inlining:SI***F***SNR vs. SI***N***SNR

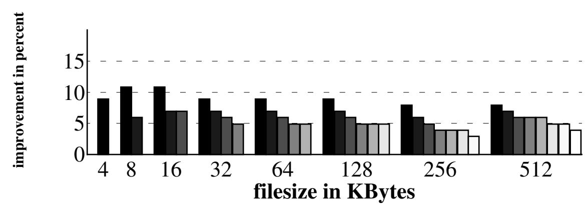

Figure 12: Safety checks: SIF***N***CR vs. SIF***S***CR

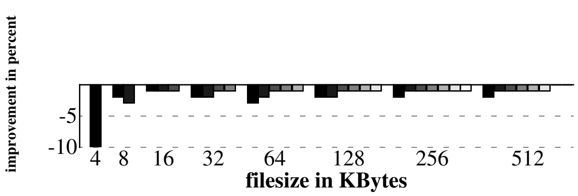

Figure 13a: No revocation: SIFS***N***R vs. SIFS***C***R

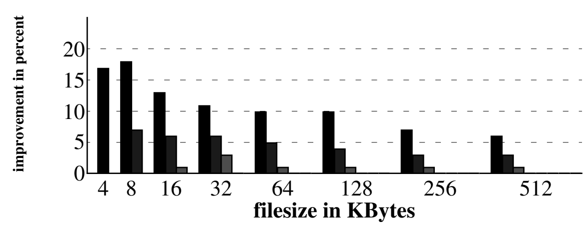

Figure 14a: Simple Java Scheduler: MIFSN***I*** vs. MIFSN***R***

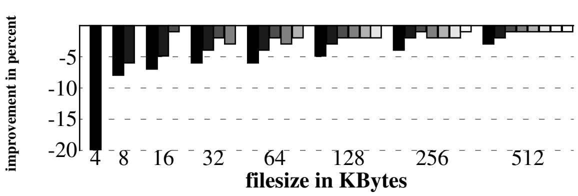

Figure 13b: SPIN revocation: SIFS***S***R vs. SIFS***C***R

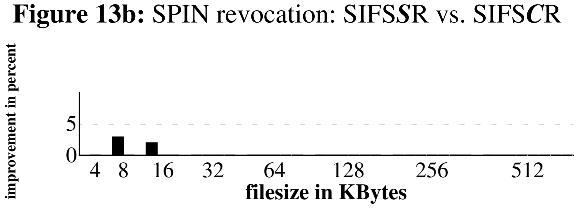

Figure 13c: ATOMIC revocation: SINS***A***R vs. SIFS***C***R

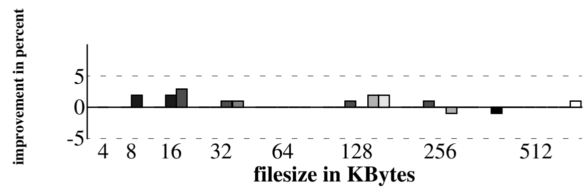

Figure 14b: Kernel scheduler: MIFSN***C*** vs. MIFSN***I***

• **CLI**: Saves the interrupt-enable flag and disables interrupts before the memory access and restores the interrupt-enable flag afterwards.
• **SPIN**: In addition to disabling interrupts a spinlock is used to make the operation atomic on a multiprocessor.
• **ATOMIC**: The JX kernel contains a mechanism to avoid locking at all on a uniprocessor. The atomic code is placed in a dedicated memory area. When the low-level part of the interrupt system detects that an interrupt occurred inside this range the interrupted thread is advanced to the end of the atomic procedure. This technique is fast in the common case but incurs the overhead of an additional range check of the instruction pointer in the interrupt handler. It increases interrupt latency when the interrupt occurred inside the atomic procedure, because the procedure must first be finished. But the most severe downside of this technique is, that it inhibits inlining of memory accesses. Similar techniques are described in [9], [36], [35], [41].

Figure 13a shows the change in performance when no revocation checks are performed. This configuration is slightly slower than a configuration that used the CLI method for revocation check. We can only explain this by code cache effects.

Using spinlocks adds an additional overhead (Figure 13b). Despite some improvements in a former version of JX using atomic code could not improve the IOZone performance of the measured system (Figure 13c).

4.4 Cost of the open scheduling framework

Scheduling in JX can be accomplished with user-defined schedulers (see Sec. 2.8). The communication between the global scheduler and the domain schedulers is based on interfaces. Each domain scheduler must implement a certain interface if it wants to be informed about special events. If a scheduler does not need all the provided information, it does not implement the corresponding interface. This reduces the number of events that must be delivered during a portal call from the microkernel to the Java scheduler.

In the configurations presented up to now we used a simple round-robin scheduler (RR) in each domain. The domain scheduler is informed about every event, regardless whether being interested in it or not. Figure 14a shows the benefit of using a scheduler which implements only the interfaces needed for the round-robin strategy (RR invisible portals) and is not informed when a thread switch occurred due to a portal call.

As already mentioned, there is a scheduler built into the microkernel. This scheduler is implemented in C and can not be exchanged at run time. Therefore this type of scheduling is mainly used during development or performance analysis. The advantage of this scheduler is that there are no calls to the Java level necessary. Figure 14b shows that there is no relevant performance difference in IOZone performance between the core scheduler and the Java scheduler with invisible portals.

4.5 Summary: Fastest safe configuration

After we explained all the optimizations we can now again compare the performance of JX with the Linux performance. The most important optimizations are the use of a single domain, inlining, and the use of the core scheduler or the Java scheduler with invisible portals. We configured the JX system to make revocation checks using CLI, use a single domain, use the kernel scheduler, enabled inlining, and disabled inlining of memory methods. With this configuration we achieved a performance between about 40% and 100% of Linux performance (Figure 15). By disabling safety checks we were even able to achieve between 50% and 120% of Linux performance.

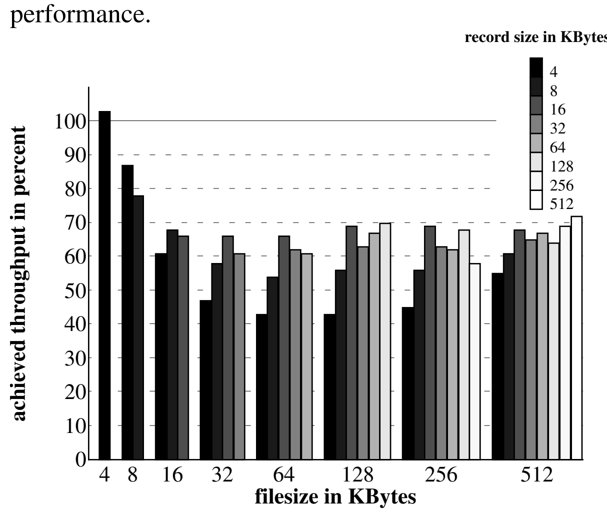

Figure 15: JX vs. Linux: Fastest configuration (SINSCC)

5 Related work

There are several areas of related work. The first two areas are concerned with general principals of structuring an operating system: extensibility and reusability across system configurations. The other areas are language-based operating systems and especially Java operating systems.

Extensibility. With respect to extensibility JX is similar to L4 [33], Pebble [25], and the Exokernel [24] in that it tries to reduce the fixed, static part of the kernel. It is different from systems like SPIN [8] and VINO [40], because these systems only allow a gradual modification of the system service, using spindles (SPIN) or grafts (VINO). JX allows its complete replacement. This is necessary in some cases and in most cases will give a better performance, because more suitable algorithms can be used inside the service. A system service with an extension interface will only work as long as the extensions fit into a certain pattern that was envisioned by the designer of the interface. A more radical change of the service is not possible.

An important difference between JX and previous extensible systems is, that in JX the translator is part of the operating system. This allows several optimizations as described in the paper.

Modularity and protection. Orthogonality between modularity and protection was brought forward by Lipto [22]. The OSF [15] attacked the specific problem of collocating the OSF/1 UNIX server, which was run on top of the Mach microkernel, with the microkernel. They were able to achieve a performance only 8% slower than a monolithic UNIX. The special case of code reuse between the kernel and user environment was investigated in the Rialto system [21]. Rialto uses two interfaces, a very efficient one for collocated components (for example the mbuf [34] interface) and another one when a protection boundary must be crossed (the normal read/write interface). We think that this hinders reusability and complicates the implementation of components, especially as there exist techniques to build "unified" interfaces in MMU-based systems [23], and, using our memory objects, also in language-based systems.

There is a considerable amount of work in single address space operating systems, such as Opal [11] and Mungi [29]. Most of these systems use hardware protection, depend on the mechanisms that are provided by the hardware, and must structure the system accordingly, which makes their problems much different from ours.

Language-based OS. Using a safe language as a protection mechanism is an old idea. A famous early system was the Pilot [38], which used a language and bytecode instruction set called Mesa [30], an instruction set for a stack machine. Pilot was not designed as a multi-user operating system. More recent operating systems that use safe languages are SPIN [8], which uses Modula3, and Oberon [47], which uses the Oberon language, a descendant of Modula2.

Java OS. The first Java operating system was JavaOS from Sun [39]. We do not know any published performance data for JavaOS, but because it used an interpreter, we assume that it was rather slow. Furthermore, it did only provide a single protection domain. This makes sense, because JavaOS was planned to be a thin-client OS. However, besides JX, JavaOS is the only system that tried to implement the complete OS functionality in Java. JKernel [28], the MVM [16], and KaffeOS [4] are systems that allow isolated applications to run in a single JVM. These systems are no operating systems, but contain several interesting ideas. JKernel is a pure Java program and uses the name spaces that are created by using different class loaders, as a means of isolation. JKernel concentrates on the several aspects how to implement a capability mechanism in pure Java. It relies on the JVM and OS for resource management. The MVM is an extension of Sun's HotSpot JVM that allows running many Java applications in one JVM and give the applications the illusion of having a JVM of their own. It allows to share bytecode and JIT-compiled code between applications, thus reducing startup time. There are no means for resource control and no fast communication mechanisms for applications inside one MVM. KaffeOS is an extension of the Kaffe JVM. KaffeOS uses a process abstraction that is similar to UNIX, with kernel-mode code and user-mode code, whereas JX is more structured like a multi-server microkernel system. Communication between processes in KaffeOS is done using a shared heap. Our goal was to avoid sharing between domains as much as possible and we, therefore, use RPC for interdomain communication. Furthermore, KaffeOS is based on the Kaffe JVM, which limits the overall performance and the amount of performance optimizations that are possible in a custom-build translator like ours.

These three systems do not have the robustness advantages of a 100% Java OS, because they rely on a traditional OS which is written in a low-level language, usually C.

6 Conclusion and future work

We described the JX operating system and its performance. While being able to reach a performance of about 50% to 100% of Linux in a file system benchmark in a monolithic configuration, the system can be used in a more flexible configuration with a slight performance degradation.

To deliver our promise of outperforming traditional, UNIX-based operating systems, we have to further improve the translator. The register allocation is still very simple, which is especially unsatisfactory on a processor with few registers, like the x86.

We plan to refine the memory objects. Several additional memory semantics are possible. Examples are copy-on-write memory, a memory object that represents non-continuous chunks of memory as one memory object, or a memory object that does not allow revocation. All these semantics can be implemented very efficiently using compiler plugins. The current implementation does not use an MMU because it does not need one. MMU support can be added to the system to expand the address space or implement a copy-on-write memory. How this complicates the architecture and its implementation remains to bee seen.

7 Acknowledgements

We wish to thank the anonymous reviewers and our shepherd Jason Nieh for the many comments that helped to improve the paper. Franz Hauck and Frank Bellosa read an earlier version of the paper and suggested many improvements.

8 References

[1] B. Alpern, A. Cocchi, S. J. Fink, D. P. Grove, and D. Lieber. Efficient Implementation of Java Interfaces: invokeinterface Considered Harmless. In *OOPSLA 01'*, Oct. 2001.

[2] M. Alt. *Ein Bytecode-Verifier zur Verifikation von Betriebssystemkomponenten*. Diplomarbeit, available as DA-I4-2001-10, Univ. of. Erlangen, Dept. of Comp. Science, Lehrstuhl 4, July 2001.

[3] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *ACM Trans. on Computer Systems, 10(1)*, pp. 53-79, Feb. 1992.

[4] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *Proc. of 4th Symposium on Operating Systems Design & Implementation*, Oct. 2000.

[5] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Techniques for the Design of Java Operating Systems. In *2000 USENIX Annual Technical Conference*, June 2000.

[6] H. G. Baker. List processing in real time on a serial computer. In *Communications of the ACM, 21(4)*, pp. 280-294, Apr. 1978.

[7] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. In *Operating Systems Review, 23(5)*, pp. 102-113, Dec. 1989.

[8] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. of the 15th Symposium on Operating System Principles*, pp. 267-284, Dec. 1995.

[9] B. N. Bershad, D. D. Redell, and J. R. Ellis. Fast Mutual Exclusion for Uniprocessors. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 223-233, Sep. 1992.

[10] R. Campbell, N. Islam, D. Raila, and P. Madany. Designing and Implementing Choices : An Object-Oriented System in C++. In *Communications of the ACM, 36(9)*, pp. 117-126, Sep. 1993.

[11] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and Protection in a Single Address Space Operating System. In *ACM Trans. on Computer Systems, 12(4)*, pp. 271-307, Nov. 1994.

[12] T. M. Chilimbi. *Cache-Conscious Data Structures - Design and Implementation*. Ph.D. thesis, University of Wisconsin-Madison, 1999.

[13] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating System Errors. In *Symposium on Operating System Principles 01'*, 2001.

[14] R. P. Colwell, E. F. Gehringer, and E. D. Jensen. Performance effects of architectural complexity in the intel 432. In *ACM Trans. on Computer Systems, 6(3)*, pp. 296-339, Aug. 1988.

[15] M. Condict, D. Bolinger, E. McManus, D. Mitchell, and S. Lewontin. *Microkernel modularity with integrated kernel performance*. Technical Report, OSF Research Institute, Cambridge, MA, Apr. 1994.

[16] G. Czajkowski and L. Daynes. Multitasking without Compromise: A Virtual Machine Evolution. In *Proc. of the OOPSLA*, pp. 125-138, Oct. 2001.

[17] P. Dasgupta, R. J. LeBlanc, M. Ahamad, and U. Ramachandran. The Clouds distributed operating system. In *IEEE Computer, 24(11)*, pp. 34-44, Nov. 1991.

[18] J. B. Dennis and E. C. Van Horn. Programming Semantics for Multiprogrammed Computations. In *Communications of the ACM, 9(3)*, pp. 143-155, Mar. 1966.

[19] Department of Defense. *Trusted computer system evaluation criteria (Orange Book)*. DOD 5200.28-STD, Dec. 1985.

[20] R. Dixon, T. McKee, P. Schweizer, and M. Vaughan. A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. In *Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 211-214, 1989.

[21] R. Draves and S. Cutshall. *Unifying the User and Kernel Environments*. Technical Report MSR-TR-97-10, Microsoft Research, Mar. 1997.

[22] P. Druschel, L. L. Peterson, and N. C. Hutchinson. Beyond micro-kernel design: Decoupling modularity and protection in Lipto. In *Proc. of Twelfth International Conference on Distributed Computing Systems*, pp. 512-520, 1992.

[23] P. Druschel and L. Peterson. Fbufs: A highbandwidth cross-domain transfer facility. In *14th ACM Symp. on Operating System Principles*, pp. 189-202, 1993.

[24] D. Engler, F. Kaashoek, and J. OToole.Exokernel:AnOperatingSystemArchitectureforApplication-LevelResourceManagement.In'*Proc. of the 15th Symposium on Operating System Principles*, pp. 251-266, Dec. 1995.

[25] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble Component-Based Operating System. In *USENIX 1999 Annual Technical Conference*, pp. 267-282, June 1999.

[26] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Aug. 1996.

[27] G. Hamilton and P. Kougioris. The Spring Nucleus: a Micro-kernel for objects. In *Proc. of Usenix Summer Conference*, pp. 147-159, June 1994.

[28] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. v. Eicken. Implementing Multiple Protection Domains in Java. In *Proc. of the USENIX Annual Technical Conference*, pp. 259-270, June 1998.

[29] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. The Mungi single-address-space operating system. In *Software: Practice and Experience, 28(9)*, pp. 901-928, Aug. 1998.

[30] R. K. Johnsson and J. D. Wick. An overview of the Mesa processor architecture. In *ACM Sigplan Notices, 7(4)*, pp. 20-29, Apr. 1982.

[31] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun Unix. In *USENIX Association: Summer Conference Proceedings*, 1986.

[32] L4Ka Hazelnut evaluation, http://l4ka.org/projects/hazelnut/eval.asp.

[33] J. Liedtke. Towards Real u-Kernels. In *CACM, 39(9)*, 1996.

[34] M. K. McKusick, K. Bostic, and M. J. Karels. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, May 1996.

[35] M. Michael and M. Scott. Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. In *Journal of Parallel and Distributed Computing, 54(2)*, pp. 162-182, 1998.

[36] D. Mosberger, P. Druschel, and L. L. Peterson. Implementing Atomic Sequences on Uniprocessors Using Rollforward. In *Software---Practice and Experience, 26(1)*, pp. 1-23, Jan. 1996.

[37] E. I. Organick. *A ProgrammersViewoftheIntel432System'*. McGraw-Hill, 1983.

[38] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell. Pilot: An operating system for a personal computer. In *Communications of the ACM, 23(2)*, pp. 81-92, ACM Press, New York, NY, USA, Feb. 1980.

[39] T. Saulpaugh and C. Mirho. *Inside the JavaOS Operating System* . Addison Wesley Longman, 1999.

[40] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *2nd Symposium on Operating Systems Design and Implementation*, 1996.

[41] O. Shivers, James W. Clark, and Roland McGrath. Atomic heap transactions and fine-grain interrupts. In *ACM Sigplan International Conference on Functional Programming (ICFP99)'*, Sep. 1999.

[42] Status page of the Fiasco project at the Technical University of Dresden, http://os.inf.tu-dresden.de/fiasco/status.html.

[43] Sun Microsystems. *Java Remote Method Invocation Specification*. 1997.

[44] Webpage of the IOZone filesystem benchmark, http://www.iozone.org/.

[45] A. Weissel. *Ein offenes Dateisystem mit Festplattensteuerung fuer metaXaOS*. Studienarbeit, available as SA-I4-2000-02, Univ. of. Erlangen, Dept. of Comp. Science, Lehrstuhl 4, Feb. 2000.

[46] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In. In *Proc. of International Workshop on Memory Management*, Sep. 1995.

[47] N. Wirth and J. Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley, 1992.

[48] F. Yellin. Low level security in Java. In *Proc. of the 4th World Wide Web Conference*, pp. 369-379, OReilly,1995.'