

A Java Embedded Micro-kernel Infrastructure

Mark Baker and Hong Ong

The Distributed Systems Group

University of Portsmouth

Portsmouth, PO1 2EG, UK

Mark.Baker@computer.org and Hong.Ong@port.ac.uk

Abstract

In this paper we outline a prototype implementation of a Java Embedded Micro-kernel (JEM) infrastructure that is capable of dynamically loading service modules at runtime onto devices with limited system resources, such as intelligent network cards or PDAs. The JEM system allows for a range of Java-based modules to be developed and utilized to, for example, dramatically improve the communications performance of Java applications, move operating system or application level services down into the network layer, or provide other specialized services.

Keywords: Java, JNI, Micro-kernel, Modules, Communication API,

1 Introduction

Today, increasing numbers of microprocessors, and other computer-based resources, are being embedded in the modern generation of devices. Embedded devices span a growing variety of consumer products, such as mobile phones, pagers, PDAs, set-top boxes, and process controllers. According to research reported by International Data Corporation (IDC) [1], revenue from embedded operating environment sales is expanding at an annual growth rate of 30%. This growth highlights the importance and interest in this area. In particular, there is a great demand for software that can easily incorporate new functions and “smart” features for a wide range of embedded products. However, designing reliable software for the increasingly complex and sophisticated embedded devices has become a major challenge.

In the past, the hardware resources available on these small and embedded devices were very limited. Therefore,

software engineers were forced to cram logic into tens of Kbytes of RAM and squeeze every available cycle out of the processor. The common programming language used in the software development cycle was assembly (machine) language or a specialist language and more recently in C. Obviously, this software development style had its disadvantages, not least of which was portability, lack of software reuse, and a limited ability to use modern software design techniques. Nevertheless, this software development style was appropriate due to the limited capabilities and performance of the target devices.

This situation is changing rapidly, however. Devices now have significant hardware resources. In particular, the current and emerging generation of devices are being put together with increasingly powerful processors and significant amounts of memory. For example, the latest generation of network card from Myricom [2] comes with 8 Mbytes of memory and a 200 MHz RISC processor (PCI64C). Equally, PDAs can now be found with 64 Mbytes of memory and a 206 MHz processor [3]. Consequently, the necessity to squeeze software into limited memory and write fast and terse codes is not as crucial as before. The presence of additional hardware capabilities not only provides software engineers with the scope to develop and implement increasingly sophisticated software to manage and run on these devices, but also, in some cases, provides the possibility for these devices to off-load functionalities normally undertaken by a traditional computer, for example with intelligent network cards.

Now, software engineers can turn towards modern programming techniques and languages, such as object orientation and Java, for the current generation of embedded devices. Additionally, software engineers can also explore and develop dynamic applications that are capable of downloading, customize and/or migrating code.

1.1 Java for Embedded Devices

Clearly, software for these modern embedded devices is harder to design. Some of the essential design issues are heterogeneity, code-reusability, concurrency, and modules

interaction. Therefore, it is necessary to have an infrastructure for developing and testing software that specifically caters to embedded devices. By modules, we mean a distributed object that is capable of performing a specific task with a set of constraints; and, by infrastructure, we mean a run-time environment that will provide the necessary mechanisms by which the modules can be executed and perhaps interact. Ideally, the modules will be reusable, platform independent, and embody valuable domain-specific expertise.

Java is a good choice for the implementation of the infrastructure and it is now well established in most areas of information technology. Using Java for embedded systems has many advantages, including:

- Platform independence allows code reuse across processors and product lines. This enables developers to deploy the same applications to a range of devices.
- Applications development can begin on a variety of desktop environments well before the targeted deployment hardware is available.
- Java provides a framework for secure networking, and thus adds the possibility of remote access for configuration and managements of devices.
- The Java class loader permits dynamic loading of classes that will facilitate the dynamic evolution of the application embedded in the devices.

The simplicity of the Java programming language – with its absence of pointers and its automatic garbage collection – eliminates many programming errors through type checking and the risk of memory leaks that affect reliability of applications.

1.2 Project Objectives

It is the aim of the JEM project to design and implement a generic infrastructure that can be used in a range of embedded devices – from network cards to PDAs and from mobile phones to printers. For this reason, one of the design goals is to minimize the device dependent code and thus maximize portability. Central to this infrastructure is a Java-based micro-kernel, which will provide the necessary mechanisms for the various runtime aspects of the system. In addition, the Java-based micro-kernel will allow a range of service modules, written in Java, to be dynamically loaded at runtime. The Java-based micro-kernel together with the service modules forms the Java Embedded Micro-kernel (JEM) infrastructure.

The rest of this paper is organized as follows. In Section 2 we detail our motivation for undertaking this project. In Section 3 we describe the JEM Architecture and discuss some related works. In Section 4 we discuss a variety of

implementation issues. Finally in Section 5 we conclude and review our current progress.

2 Project Motivation

Our motivation for undertaking this project is driven by several needs:

- An ongoing project, MPJ [4], and its needs for fast and reliable communication (discussed in Section 2.1).
- The ability to move services from system and user-space down into the network layer, for example, packet filtering or daemon services (discussed in Section 2.2)
- The ability to test the design and implementation of services for embedded devices (see Section 2.3).

2.1 Java Message Passing

The Message-Passing Working Group of the Java Grande Forum [5] was formed late 1998 as a response to the appearance of several prototype Java bindings for MPI-like libraries. An initial draft for a common API specification was distributed at Supercomputing '98. The present API is now called MPJ [7]. Currently there is no complete implementation of the draft specification. One message-passing interface, `mpiJava` [6][8], is moving towards the “standard”. `mpiJava` currently relies on the availability of platform-dependent native MPI implementation for the target computer. While this is a reasonable basis in many cases, the approach has some disadvantages; for example, the need to manage issues related to portability over variety of platforms (UNIX and windows). An ongoing effort is focusing on producing a “pure” Java reference implementation of the MPJ [4] environment, which solves issues related to portability.

Even though Java has many advantages, several aspects of its computational and communications performance are generally regarded as poor. Even though the computational performance of Java applications is improving with the likes of Just-In-Time (JIT) compiler technologies and the optimizations of the JVM, Java communications performance is still regarded as poor. Inter-process communications in Java are generally undertaken via the RPC-like semantics of a Remote Method Invocations (RMI) method calls or via Java implementation of BSD Sockets.

Both `mpiJava` and MPJ, however, still suffer from the problems associated with traditional inter-computer communications bottlenecks, related to multiple data copies and management via the operating system. These bottlenecks can be minimized with the use of intelligent network cards and communication techniques that by-pass

the operating system, for example, the Virtual Interface Architecture [9].

Besides bypassing the operating system, the MPJ collective communication operations such as gathering and scattering of data, and broadcasting could also be implemented in the intelligent network card. Since collective communications are widely used in most MPI applications, it would be beneficial to reduce the latency of these operations as much as possible.

2.2 Moving Services into the Network Layer

Communication-oriented applications desire the highest bandwidth and lowest latency possible. The latency is primarily comprised of the time required to process a message and the time spent on the wire. Modern high-speed networks such as Myrinet and Gigabit Ethernet have shifted the bottleneck in communication from the network layer to the messaging software at the sending and receiving endpoints. Traditionally message processing by the operating system caused multiple data copies and required many context switches. This increases the overall end-to-end latency. This led to the development of user-level network protocols where operating system intervention was removed from the critical path of the message.

There has been substantial research work in offloading network protocol processing onto network adaptors. However, there has not been much discussion about offloading service functionality, as opposed to protocol support. Services such as firewalls, Web proxies, multicast routers, mobile proxies, and video gateways, that act upon user data could be offloaded onto the network adaptors. For example, a firewall not only implements packet filtering but also application specific functions. Currently, the firewalls need to be updated manually in order to enable the use of new applications. However, this process could be automated by allowing a new application to authenticate itself to the firewall and load the appropriate modules. Another example is the application specific multicast for video conferencing. The code could be loaded onto the network adaptor and provide the mechanisms to fine-tune the quality of service of video packets.

It is our belief that offloading service functionality onto the network will improve overall service response time and enable richer interactions.

2.3 Services for Embedded Devices

Operating system designers have adopted many software methodologies, such as modules and object-oriented

approaches. A popular move has also been to adopt micro- and pico-kernel architectures, where services are moved from the kernel into user-space in order to separate each sub-system's complexity [10]

The characteristics of application environments for which operating systems are built are radically changing, and this is leading to a new set of operating system requirements involving the management of change, mobility, quality of service and configuration in network architecture. These requirements are driven by innovation stemming from the Internet, such as streaming multimedia, or other advances like portable low-powered computing devices, wireless networks, visualization, smart rooms and smart devices. Future operating systems must support the emerging anytime/anywhere consumer usage [10][11].

To provide the services needed for anytime/anywhere consumer usage, fast, flexible and extensible operating systems need to be designed and developed. We believe that a Java-based micro-kernel will provide an ideal candidate platform to design, develop and implement a variety of dynamic services that are needed for current and emerging applications.

2.4 Summary

This project is motivated by a number of diverse drivers of which the major one is for producing optimized application services. In particular, these application services would bypass the normal performance bottlenecks found when implemented as system or user level services. These drivers have led us to believe that we need to develop and implement a generic JEM infrastructure for accommodating a wide-range of embedded products. The design and implementation of a set of service modules for the JEM infrastructure is also essential. In the subsequent sections, we will describe the design of the JEM architecture.

3 The Java Embedded Micro-Kernel Architecture

3.1 Introduction

Micro-kernels are not new. System architects have been designing and using micro-kernels since the mid-80s. A micro-kernel attempts to decompose the conventional monolithic kernel horizontally by migrating the kernel functionality into servers running in user space. This decomposition allows the micro-kernel to be small, fast, and does not burden each CPU with facilities, such as a complete file system, that it does not need [12]. As a consequence, the micro-kernel provides the foundation for modular, scalable and portable extensions. Typically,

micro-kernels have been used where there are limited system resources; the ability to load only essential services at any one time has been the best means of reducing kernel overheads [13].

Recently, there has been increasing interest in using Java-based micro-kernels [14] due to the advantages of Java briefly outlined in Section 1.1. Using Java, however, is not without problems. Some drawbacks of Java are that the Java Virtual Machine (JVM) tends to have a large memory footprint, is computationally slower than alternatives such as C and C++, has non-deterministic treatment of asynchronous exception events, and non-bounded resource usage, which are normally needed to meet the constraints of embedded software.

Although some solutions have been proposed to solve these shortcomings of Java regarding its use for embedded software programming [15][16], they have by no means completely eliminated the problems. Meanwhile, modern devices, which have significant memory and a reasonably fast processor, along with technologies such as optimizing compilers, have made Java a realistic option for embedded devices. However, with Java-based systems there is still a need to minimize memory and other resources used. Consequently, we feel that a micro-kernel based system, at this time, is the only viable alternative for providing the services as well as an experimental and research platform that we require.

3.2 Generic JEM Architecture

The design of JEM architecture is based largely, but not entirely, on L4 [17] that was originally proposed by Liedtke in the mid 90's. This design decision was made mainly due to the fact that most existing micro-kernels are not "tiny" (with the exception of QNX micro-kernel [18]). Additionally, there are several reference implementations of L4, for example L4/x86 and Fiasco [1], have reported overcoming the IPC performance problems that most micro-kernels experienced in the early 80's up till present. Consequently, we felt that instead of re-inventing the wheel of micro-kernel design, it would be more sensible to start a Java-based micro-kernel design based on existing one, i.e., L4.

Figure 1 shows the generic JEM architecture. This consists of the following parts:

- The Abstract Hardware Layer (AHL) is the JEM interface to the underlying device hardware. It is the means by which JEM interacts with the device processor registers and memory.
- The Java Embedded Micro-kernel (described more fully next).

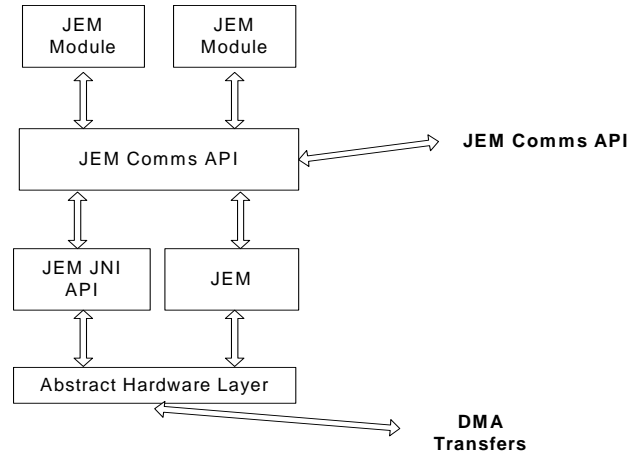


Figure 1: Generic JEM Architecture

- The JEM JNI API is the fast by-pass route for the JEM communications API to interact with the device AHL.
- The JEM Communications API is a messaging API that provides JEM models with data I/O services. This API interacts with its peer on other devices.
- The JEM Modules provide application level services for a Java application.

JEM supports three abstractions – modules and threads, address space, and messages. Modules abstraction arises from separating the traditional notion of a process into two sub-concepts. A module contains the resources associated with a process such as address space and I/O or file descriptors. A module does not perform computation itself but serves as a framework in which thread(s) can operate. Hence, a thread is a control unit, which contains the minimal processing state associated with a computation for example a program counter, a stack pointer, and other hardware register state information.

Address space is a mapping associated with each virtual page to a physical page frame. Each module has a virtual address space within which its thread executes. Like L4, JEM supports the recursive construction of address space concept. Depending on the capabilities of the target device, the module could construct and maintain further address spaces on top of the initial space. In order to support this concept, JEM provides three primitive operations: grant, map, and flush. The grant operation allows the owner of the address space to *grant* any of its pages to another space if and only if the recipient agrees. The map operation enables the owner of the address space to *map* any of its pages into another address space; again, the recipient has to agree. The flush operation allows the owner to de-map any of its pages. By supporting these three primitives, a higher-level memory server could be implemented on top of JEM to offer various memory protection schemes. A more detail explanation of the concept of recursive construction of address space could be found in [17].

All inter and intra communications among modules is carry out in the form of messages. A message is considered as a stream of bytes and contains the data to be communicated.

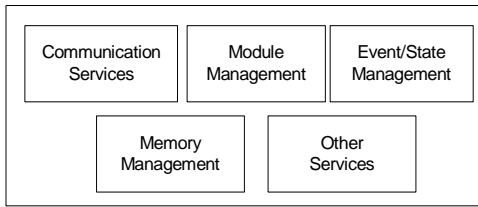


Figure 2: The Java Embedded Micro-Kernel (JEM)

Figure 2 shows the essential services that makeup the JEM:

- The Communication Services component is used for activities like maintaining descriptors, and transmitting/receiving and routing of messages.
- The Module Management component will maintain JEM-modules, and possibly load and unload them.
- The Memory Management component will undertake DMA/PIO to and from host hardware. This component will supports the concept of address spaces mentioned earlier.
- The Event/State Management component will monitor the state of the kernel; perform actions when receiving signals either from the host, remote NIC or switch.
- The Other Services component for now is not used but specialized services may be required to facilitate hardware specific capabilities (e.g., 2 CPUs).

Figure 3 shows the interaction of a JEM enabled network card and its host workstation. This particular example

shows the communications layers that would be used by a MPJ application send/receiving messages to and from another JEM enabled workstation. It should be noted that the MPJ message passing API would be built up from the lower-level JEM Communications API.

3.3 Related Work

As mentioned earlier, this work is largely based on the design concepts of L4 micro-kernel. However, our implementation is being carried out in Java as opposed to the original assembly language and C. In addition, unlike the L4 argument that micro-kernels need different implementations for different platforms, this work intends to show that JEM could be easily ported by carefully “drawing the red line” between Java code and hardware dependent code and by taking advantages of Java language construct.

To the best of our knowledge, the two research projects that are based on Java and provide direct user level access to system resources are Jaguar [20] and PANIC [21][22]. Jaguar accomplished this by translating the Java bytecode to machine code at runtime, while PANIC offloads protocol processing by splitting the protocol components across host and network interface. JEM is different in that it is not only aimed to off-load protocol processing but also to off-load application specific services. This requires JEM to address other design issues, mentioned earlier, than just by passing operating systems calls. However, JEM draws experiences from these two works (especially from Jaguar) in terms of providing support for user-level networking [23][24].

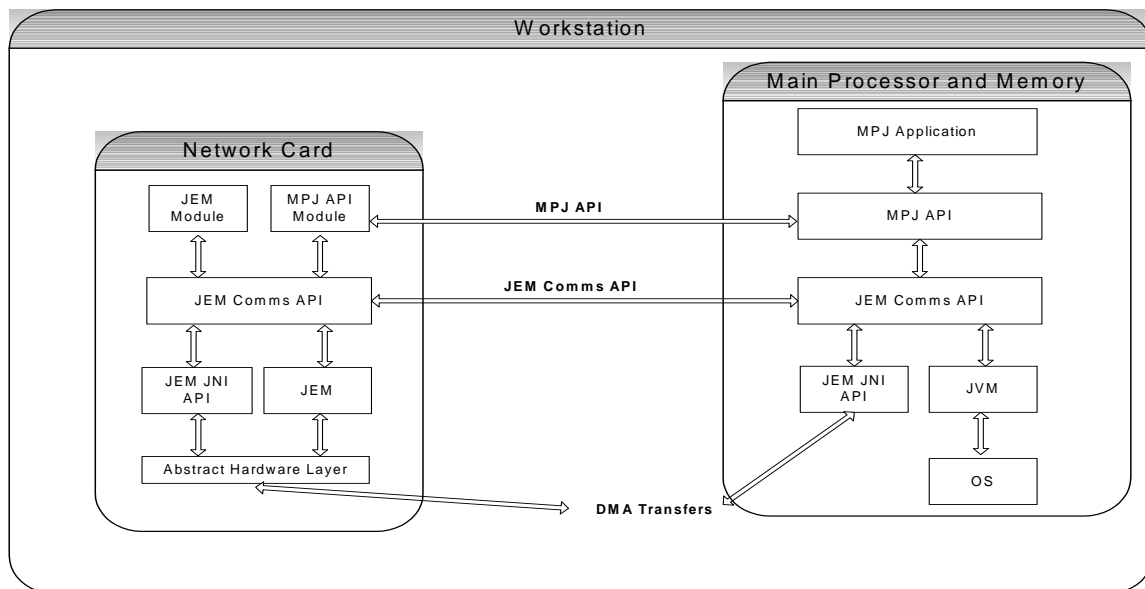


Figure 3: The Relationship between a JEM enabled network card and host workstation.

There are many research efforts on Java-based operating systems. The two projects that are closest to JEM goals are U-Net/SLE (Safe Language Environment) [25] and SPINE [26]. The U-Net/SLE project implemented a subset of the JVM for the LANai4; thereby allowing customized packet processing to occur on the network adaptor. Because of the limitation in both processor speed and memory size, the primary focus of their work was on the performance of the JVM. Their preliminary results indicated that the LANai4 processor lacked sufficient power to support the JVM interpretation efficiently. The SPINE project extended the ideas of SPIN [27][28] to the network interface. Underlying SPINE is a Modula-3 runtime executing on the network interface. The common aspects of these two approaches are the use of a kernel-integrated compiler to generate safe-code inside the kernel space. This enabled them to reduce the address-space switch and thus reduce the IPC costs. However, the kernel level compiler might impose overhead on the kernel. On the other hand, JEM is based on the L4 address space abstraction, which does not incur the cost of an address space switch and the JVM could be implemented as a service.

Other Java-based operating systems efforts are J-Kernel [29], KaffeOS [30], Joust [31] and Sun's JavaOS [32]. J-Kernel is structured like a microkernel-based system with automatic stub generation for inter-task communication. KaffeOS is based on the Kaffe JVM that provides mechanisms to support a process model. Both of these systems require a separate JVM for each Java application, and all run in supervisor mode. The Joust project integrated a JVM into the Scout operating system [33]. Joust uses Scout's path abstraction to provide control over CPU time and network bandwidth. All these approaches are sometime also known as language-based extensible systems [34][35]. These systems are either limited by their monolithic structure or are built upon a full features operating system and JVM. Although J-Kernel and KaffeOS try to overcome the monolithic structure, their research is mainly concerned with introducing the traditional concept of a process and a "red line" [36]. Sun's original JavaOS was a standalone operating system written almost entirely in Java. It is described as a "first class operating system for Java applications", but appears to provide a single JVM with little separation between applications. It is being replaced by a new implementation termed "JavaOS for Business", that also only runs Java applications [32]. JavaOS for consumers is built on the Chorus micro-kernel in order to achieve real-time properties needed in embedded systems.

JEM takes a different approach by emphasizing flexibility and customizability. In particular, JEM takes the pure micro-kernel approach in which basic mechanisms are implemented only to support implementation of higher-

level services, such as application protection with domain specific policy. In addition, like J-kernel and JavaOS, JEM is written in Java and the use of native code is avoided wherever possible.

4 JEM Implementation Issues

Work is currently underway to provide the initial, base-level implementation of JEM that runs directly on intelligent network interfaces, for example, Myrinet hardware. In this section, we highlight the various aspects of the implementation framework.

It is the intention of this implementation to investigate and abstract all the necessary APIs required to support the more generic JEM framework. The current implementation assumes that the intelligent network interface contains:

- A micro-controller,
- Static RAM,
- A re-programmable EPROM.

The JEM framework consists of the following parts:

- JEM: This is the core of the proposed infrastructure that provides the runtime environment for JEM modules.

The three major components in the design of the JEM are:

- Memory Management:

The SRAM memory will be partitioned into the run-time environment, the module code and data. Since the memory on the embedded device is limited, it is important to keep track of the available memory for modules and their data. The host's memory will also serve as a secondary memory area.
- Memory Protection:

Each module will be given its own memory segment. This ensures that modules are not writing to each other's data segment.
- Communication:

In general, a PCI based intelligent network interface provides either PIO or DMA data transfer. We have chosen the DMA approach. The messaging system will be an event-based one.
- JEM module guidelines and API:

The code is generally loaded as a module. A module is defined as the basic unit of program development in JEM. Each module will provide well-defined and independent services. Typical examples are an IP router, a Web server, a module that implements the VIA protocol, a video decoder. It is necessary that a set of rules and guidelines to be defined that allow

“standard” modules to be developed so that they can interact with the JEM through a set of low-level primitives correctly. Since JEM modules will be developed using Java, this will ensure portability. A JEM Module API will be developed for using the JEM communication, and other, services. Currently, the JEM module API is being drafted.

- **JEM Module compiler and loader:**
It will be necessary to compile JEM source code into compiled JEM modules and then load these JEM modules onto the JEM. This implies that a modified Java compiler is needed to translate the JEM source code into a suitable machine code to run on devices.
- **Abstract Hardware Layer API:**
The AHL API will be a native interface that provides services to the JEM. This will need to be loaded first before JEM.
- **JEM Communication API:**
A low level messaging system needs to be developed to move data from the host to device and from the device to the network link. A well-defined set of primitives is required so that the JEM module could use the primitives to build a higher-level communication module. In particular, the JEM module may implement IP, VIA, collective communications such as gather, scatter, and barrier operations.
- **JEM configuration and loading tool:**
The JEM needs to be loaded at runtime to host the JEM modules. This tool is used to configure and load the JEM at runtime.

5 Summary and Conclusions

In this paper we have detailed and discussed our proposed Java Embedded Micro-kernel Infrastructure. The purpose of this infrastructure is to provide application services on small devices, such as network cards or mobile phones. In order to provide these services a Java-based micro-kernel environment is being developed, alongside a general-purpose development and runtime infrastructure to support the JEM infrastructure.

Currently the work on this project is at an embryonic stage. The authors are presently working on the Abstract Hardware Layer and the Java Embedded Micro-kernel. Greater detail of these and other JEM components will be reported in the final paper and at the conference if the paper is accepted for publication.

Acknowledgements

The authors would like to thank Myricom for the loan of various network components, which are being used to prototype and test the JEM framework and code.

References

- [1] IDC, <http://www.windriver.com/press/html/jeode.html>
- [2] Myricom, <http://www.myri.com/>
- [3] Linux devices, <http://www.linuxdevices.com/articles/AT4936596231.html>
- [4] M.A. Baker and D.B. Carpenter, MPJ: A Proposed Java Message-Passing API and Environment for High Performance Computing, the 2nd Java Workshop at IPDPS 2000, Cancun, Mexico, LNCS, Springer Verlag, Germany, pp 552 - 559, May 5th 2000, ISBN 3-540-67442-X
- [5] JavaGrande, <http://www.javagrande.org/>
- [6] M.A. Baker, D.B. Carpenter, et al., mpiJava: An Object-Oriented Java interface to MPI, the 1st Java Workshop at the 13th IPPS & 10th SPDP Conference, Puerto Rico, April 1999, LNCS, Springer Verlag, Heidelberg, Germany, ISBN 3-540-65831-9
- [7] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox: MPJ: MPI-like message passing for Java. *Concurrency - Practice and Experience* 12(11): 1019-1038 (2000)
- [8] mpiJava, <http://aspen.ucsf.edu/pss/HPJava/mpiJava.html>
- [9] VIA, <http://www.viaarch.org>
- [10] R. Campbell, 2K: A Distributed OS for the New Millennium, 2nd ECOOP Workshop on Object-Oriented and Operating Systems. June, 1999, Lisbon, Portugal.
- [11] A. Gokhale, D. Schmidt, T. Harrison, and G. Parulkar, Operating System Support for High-Performance, Real-Time CORBA. 5th International Workshop on Object Orientation in Operating Systems. October 1996, Seattle, USA.
- [12] Andrew S. Tanenbaum, A Comparison of Three Microkernels, *The Journal of Supercomputing*, vol. 9, No 1/2, 1995
- [13] A. Messer and T. Wilkinson, Components for Operating System Design, 5th International Workshop on Object Orientation in Operating Systems. October 1996, Seattle, USA
- [14] Real-time Java Specification, <http://www.rti.org>
- [15] L. Carnahan and M. Ruak. Requirements for real time extensions for the Java platform, *Technical Reports, RTJWG, September 1999.*
- [16] H. McGhan and M. O'Connor. PicoJava: a direct execution engine for Java bytecode. *IEEE Computer*, 31(2), October 1998.
- [17] J. Liedtke, On Micro-kernel Construction, *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*. ACM Press, New York, 1995, pp. 2-11.
- [18] D. Hildebrand, An Architectural Overview of QNX, *Proceedings of the USENIX Workshop on Micro-*

- kernel and Other Kernel Architecture*. Seattle, WA,
- [19] Michael Hohmuth, The Fiasco Kernel: Requirement Definition, *Technische Universita Dresden, Technical Report, December 1998*.
- [20] M. Welsh and D. Culler, Jaguar: Enabling Efficient Communication and I/O in Java, *Concurrency: Practice and Experience, vol. 12, pp. 519-538, Special issue on Java for High Performance Application, Dec. 1999*.
- [21] PANIC,
http://www.cs.tamu.edu/faculty/bettati/panic_overview.html
- [22] C. Beauduy and R. Bettati, Protocols Aboard Network Interface Cards (PANIC), Proceedings of the IASTED International Conference Parallel and Distributed Computing and Systems, Cambridge Massachusetts, USA, November 3-6, 1999
- [23] M. Welsh, A. Basu, and T. von Eicken, Incorporating Memory Management into User-Level Network Interfaces, In Proceedings of Hot Interconnects V, Stanford, August 21-23, 1997
- [24] M Welsh, Safe and Efficient Hardware Specialization of Java Applications, UC Berkeley Technical Report, May 2000.
- [25] M. Welsh, D. Oppenheimer, and D. Culler, U-Net/SLE: A Java-based User-Customizable Virtual Network Interface, In Scientific Programming, Vol. 7, No. 2, 1999, Special Issue on High Performance Java Compilation and Runtime Issues.
- [26] M.E. Fiuczynski and B. N. Bershad, SPINE – A Safe Programmable and Integrated Network Environment, In Proceedings of the 8th ACM SIGOPS European Workshops, 1998
- [27] B.N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, S. Eggers, and C. Chambers, Extensibility, Safety and Performance in the SPIN Operating System, In Proceedings of the Fifteenth 1992.
ACM Symposium on Operating Systems Principles (SOSP), 1995
- [28] N. Wirth and J. Gutknecht, Project Oberon. ACM Press, New York, NY, 1992
- [29] C. Hawblitzel, T. von Eicken, A case for language-based protection, *Technical Report TR-98-1670, Cornell University, March 1998*.
- [30] G. Back, W. C. Hsieh, J. Lepreau, Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. *Proceedings of the 4th OSDI, pp. 333-346, Oct 2000*.
- [31] J. Hartman, L. Peterson, A. Bavier, P. Bigot, P. Bridges, B. Montz, R. Piltz, T. Proebsting, and O. Spatscheck. Experiences building a communication-oriented JavaOS. *Software--Practice and Experience, Vol. 30, Issue 10, April 2000*.
- [32] JavaOS for Business,
<http://developer.java.sun.com/developer/products/JavaOS/>
- [33] A.B. Montz, D. Mosberger, S.W. O'Malley, L.L. Peterson, and T.A. Proebsting, Scout: A Communications-Oriented Operating System. *Hot OS, May 1995*.
- [34] 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, *Communication of ACM, 23(2), pp. 81-92, 1980*.
- [35] D. C. Swinehart, P. T. Zellweger, R. J. Beach, and R. B. Hagmann, A structural view of the cedar programming environment, *ACM Transactions on Programming Languages and Systems, 8(4), pp. 419-490, Oct. 1986*.
- [36] D. R. Cheriton. Low and High Risk Operating System Architectures. *Proceedings of OSDI, pp. 197, Nov. 1994*.