Movitz: Using Common Lisp for kernel-level programming on commodity hardware

Frode Vatvedt Fjeld[*]

Department of Computer Science, University of Tromsø, Norway.

April 17, 2004

Abstract

We describe the motivation and design of a software platform for kernel-level programming in Common Lisp on the x86 PC architecture. This platform supports the full range of Common Lisp's dynamic and introspective features, and aims to offer interactive and exploratory programming to operating system kernels and stand-alone systems, while remaining architecturally small and policy free so as not to obstruct any experimentation or system designs. The platform offers a cross-compiler, and aims to provide a viable alternative to the C and assembly-based development that is prevalent in kernel-level programming today. We present our experiences and on-going work with Movitz, its current status and potential as a development platform, and our directions for future work.

1 Introduction

Common Lisp and dynamic, interactive programming environments are established as useful platforms for exploratory programming, general purpose applications, and also for operating systems programming with the lisp machines' special purpose hardware. We wish to investigate whether we can have a similar platform for kernel-level programming on contemporary commodity hardware, and whether it is possible to construct a *development platform*—as opposed to a complete operating system architecture—such that it is still possible to experiment with fundamental (operating) system concepts and designs. In other words, the platform should support any number of widely different kernels or applications, just as e.g. the `gcc` compiler provides for numerous kernels and operating systems. However, in contrast to `gcc`-based development, where for example the various implementations of the `libc` library tend to be specific to some extensive software architecture, we would like to see substantial parts of the Common Lisp language library and system libraries provided with only minimal architectural dependencies. This would be similar in spirit to the C based Flux OSKit [5], while taking advantage of the flexibility and dynamicity provided by the Common Lisp language. We believe such a development platform could open up new possibilities in the realm of operating system programming, in terms of e.g. flexibility and performance. The Movitz project is an attempt to construct such a platform.

The Movitz development platform is a target-side run-time environment for the x86 PC hardware platform (the 386 generation and newer) running without any operating system support, and a host-side cross-compiler and associated development tools that run under any Common Lisp. Movitz can simultaneously be considered to

[*]`frodef@cs.uit.no`

be a Common Lisp implementation in its own right, a deployment option for Common Lisp applications as stand-alone systems, the core of a future general-purpose, Lisp-centric operating system, or as a tool for exploration and development of OS kernel concepts and systems. It is a goal for Movitz to commit to only a minimum of architectural design choices, in order to serve as a viable platform for a wide range of experiments and systems.

Movitz is *not* primarily an effort to write an operating system in Common Lisp in place of e.g. Unix. If this was the goal, the most rational approach would probably be to shoe-horn an existing Common Lisp system into running directly on hardware. We believe however that such an approach would severely limit the space of designs and concepts that might be explored. Therefore, we have designed and implemented Movitz from scratch, taking care to minimize dependencies of every kind, and in particular those that represent forms of policies. For example, Movitz contains no complete implementation of the SLEEP function, because it is impossible to implement SLEEP without making far-reaching assumptions about how to measure time, and how to suspend execution. Therefore, Movitz can be used as a platform for systems with any sleeping policy, ranging from the most primitive busy-waiting to e.g. an application-defined scheduler with threads and blocking based on timer interrupts.

In short, the Movitz project investigates whether Common Lisp can be appropriately used for kernel-level programming, and how feasible it is to provide a Common Lisp implementation whith only minimal architectural dependencies.

In this paper we discuss our design considerations, experiences, and Movitz' potential as a tool for exploring concepts and building systems in the operating system kernel or similar application domains. We first present a technical overview and some details of Movitz in its current incarnation.

2 The Movitz run-time environment

The run-time environment is designed specifically for the x86 PC platform running in 32-bit mode. This design does not take portability across CPU architectures into consideration. This decision is partly inspired by Liedke's argument that efficient low-level system abstractions are inherently not portable [10], but is also motivated by the goal of avoiding any hindrances to performing experiments with very platform-specific concepts.

Conceptually, Movitz' run-time environment consists of three parts: the basic execution environment, the ANSI-specified Common Lisp library [2], and a library specific to Movitz, the x86 CPU, and the x86 PC architecture.

2.1 The basic execution environment

This section describes some aspects of Movitz' run-time environment from a low-level technical perspective, and may be skipped without consequences for the understanding of the rest of this text.

The most fundamental execution environment is of course the CPU itself. We try to use the CPU as conventionally as possible. For example, we use the stack and frame-pointer registers just as e.g. C compilers would, and ensure that `call` and `ret` CPU instructions are matched pairwise, because recent CPUs optimize on this assumption. At the same time, we try to avoid utilizing features of the CPU that are typically used for building OS services, in order to keep these free for experimentation and use by applications in general.

Movitz extends the basic CPU state (i.e. the general purpose registers, the instruction pointer and status flags) with a structure in main memory called the

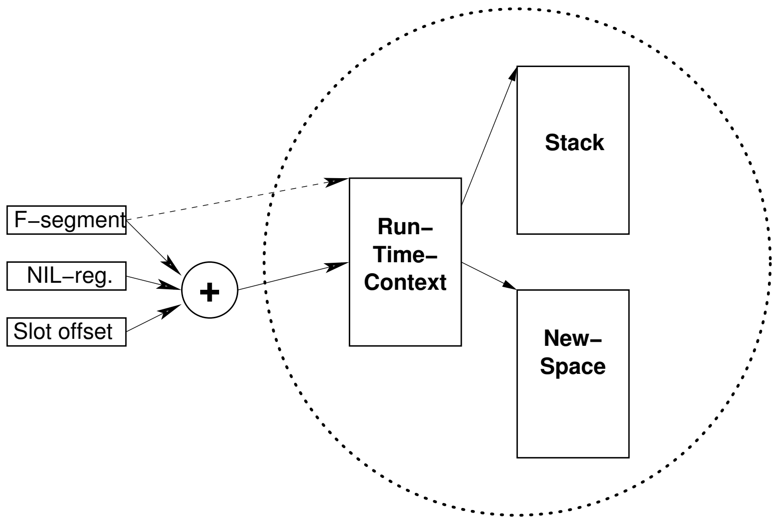

Figure 1: The run-time-context thread-local variables, including the current stack and new-space. It also holds numerous other thread-local variables and data. The dotted circle demarcates the conceptual notion of an execution context. The figure also shows how a thread-local slot in the run-time-context is typically referenced, where the slot's offset is a compile-time, system-wide constant.

"run-time-context". This structure holds dynamic state such as the current dynamic variable environment (a pointer to a linked list on the control stack), but also any other state that might be considered thread-local, such as storage for function return values beyond the secondary, and a reference to the new-space from which to allocate memory. One CPU register is assigned to always point to this structure. This register's pointer value also serves as the globally constant NIL value, making this frequently used value immediately accessible. Another CPU register is dedicated to point to the currently active function object. A function's references to pointer objects are stored in the function object, so that no object pointers need exist in the code-vectors as such, thus making code-vectors completely opaque to e.g. a copying garbage collector. The remaining four CPU registers are free for general purposes.

There is no concept of threads or processes as such in the Movitz platform. However, the run-time-context data structure is very much designed to be a building block for such abstractions, and aims to support concurrency. For dynamic variables (and similar dynamic environment elements), deep binding is employed, so that it is possible to have truly concurrent execution, i.e. systems with multiple CPUs. The compiler can be instructed to reference thread-local storage through a designated x86 segment register, and by this mechanism each thread can have a unique run-time-context structure while at the same time keeping the NIL-valued register globally constant. While using the x86 segmentation architecture this way runs counter to our goal of minimizing dependencies on special CPU features, we believe this is a fair trade-off between efficiency and architectural intrusiveness.

Figure 1 shows a sketch of the run-time-context, and how this structure together with a stack and new-space constitute the major elements of a program execution context. We also intend for this concept of the execution context, the dotted circle in the figure, to demarcate a boundary inside which to "make the common case fast". That is, when crossing this boundary we accept increased overhead so as to improve the flexibility of the platform. What this general principle can mean in practice, is that the common case of writing to a memory storage cell that is inside

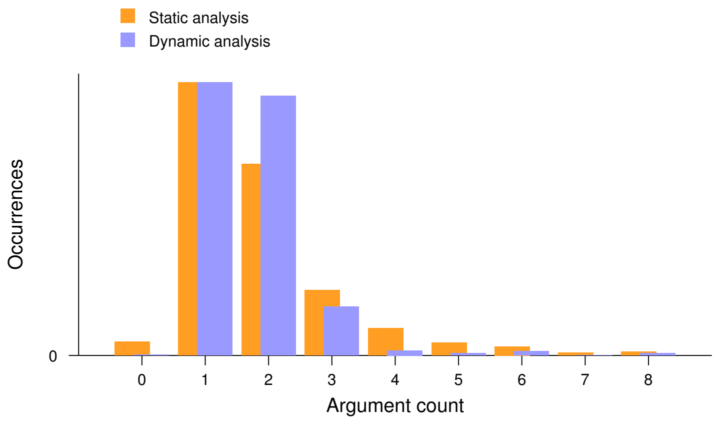

Figure 2: Two histograms of function argument count occurrences by static analysis of compiled function calls across a Movitz kernel, and dynamic analysis of the same kernel booting up and running for a few seconds. The 1, 2, and 3 argument occurrences account for 88% in the static case, and 97% in the dynamic case. The number of occurrences beyond 8 are negligible.

the program's execution context (i.e. its new-space, stack, or run-time-context) can be inlined to a primitive operation, whereas writes outside the execution context might invoke some protocol that can for example be used to implement the write barrier for some GC architecture. These ideas remain to be investigated more closely.

The function call protocol is a central part of any Lisp implementation. A design goal of the Movitz function call protocol is to allow efficient calls to the small functions that tend to be prolific in Lisp environments. This is important, because we want to minimize the use of efficiency-motivated inlining, and thus maximize the system's dynamicity. To this end, each function object contains four entry-points: one default entry-point, and three entry-points for optimized calls. The latter are used for calls with one, two, or three arguments respectively, and allow such calls to proceed without conditional branching, even when the callee specifies optional arguments. As evidenced in Figure 2, calls of one, two, or three arguments can make up about 97% of all function calls in typical code. In contrast, the default entry-point protocol carries the number of arguments in a designated register, requiring at least one register load and one conditional branch for checking the argument count in the callee function. The Movitz compiler also allows the programmer to specify separate implementations (i.e. function bodies) for each entry-point. This mechanism enables us to provide highly optimized implementations of the basic uses of many standard operators, without resorting to inlining of any kind. For example, the separate two-argument implementation of FIND can bypass all argument parsing and assume the default values for each of its six keyword arguments.

Another important aspect of a Common Lisp implementation is the dynamic typing regime. Movitz employs a two-stage tagged word strategy [7]. The lower three bits of a 32-bit word carries type information, while the upper 29 bits represent either immediate data or an eight-byte aligned pointer. Two of the eight tag values are allocated to immediate integers, so that Movitz' FIXNUM class is equivalent to the type (SIGNED-BYTE 30). A tag value is also allocated to the CHARACTER immediate class, and to each of the CONS, NULL, and SYMBOL pointer classes. Of the remaining two tag values, one is the second-stage marker used for every other kind of pointer class, which carry additional type information in the object pointed to. The last tag value is left free for experimental use.

There is no garbage collection implemented in the Movitz platform. Again, this is because we do not want to restrict the freedom of experimentation, even with GC architectures. However, the dynamic typing, stack, and CPU disciplines in Movitz are designed with garbage collection in mind. We do provide operators to map a function over the potential pointer references of heap and stack areas, and using this mechanism we have successfully implemented a simple stop-and-copy garbage collector in about 150 lines of Lisp code.

2.2 ANSI Common Lisp support

We aim with Movitz in principle to implement the Common Lisp language [2]. The aspects of Common Lisp that have yet to be implemented fall into two categories. Firstly, as we have already exemplified in the introduction in Section 1 with the SLEEP function, not all operators can be fully implemented by Movitz, because this would limit Movitz as a platform for experimentation with system concepts. This is for the most part the operators related to time, file, and pathnames, as well as those related to dynamic memory management. In such cases, we aim to provide the hooks and mechanisms necessary for implementing this functionality in the context of a particular application system. Secondly, there are operators and classes that are only partly implemented, or not implemented at all, because this has not been prioritized yet. Notable members of this set are numbers other than fixnums, multi-dimensional arrays, readtables, pretty-printing, and the related operators. And, except while cross-compiling, there is no real COMPILE function available, and EVAL is missing some standard special operators and almost all macros. However, many operators that are expectedly useful for interactive use are present, e.g. LET, SETF, LAMBDA, DEFUN, and FUNCTION.

Turning our attention to what *does* exist in Movitz at present, there is most of CLOS, with the major exception being non-standard method-combinations. Our CLOS implementation mostly adheres to the meta-object protocol, and it was also originally based on the Closette design [9]. Other notable features in Movitz are conditions, restarts, packages, hash-tables, LOOP, much of the FORMAT sub-language, and the beginnings of an implementation of streams based on Franz's simple-streams specification [8]. However, when enumerating Movitz' features, it is necessary to differentiate between what is available in the cross-compiling environment, and in the actual Movitz run-time. As indicated by Table 1, Movitz is currently skewed towards more complete support in the cross-compiling environment, although we expect the target side to catch up as Movitz grows closer to becoming self-hosting.

2.3 The Movitz-specific library

The Movitz-specific library is concerned with two things: The implementation-dependent aspects of Movitz as a Common Lisp implementation, and interfacing with the hardware environment. Because Movitz is work in progress, this library is not complete, or even very well specified in terms of the set of operators nor the operators' exact semantics.

The Movitz library provides access to all features of the hardware, often by wrapping hardware mechanisms in a thin abstraction layer so as to make them fit in naturally with the Lisp environment. For example, accessors are provided for the CPU's segment registers, control registers, various descriptor table registers, x86 model-specific registers[1], I/O ports, and untyped memory access by pointer

[1]These mechanisms are documented e.g. in the Intel and AMD x86 reference manuals.

Operator	Host	Target
DEFCLASS	✓	
DEFCONSTANT	✓	
DEFGENERIC	✓	
DEFINE-COMPILER-MACRO	✓	
DEFINE-CONDITION	✓	
DEFINE-METHOD-COMBINATION		
DEFINE-MODIFY-MACRO	✓	
DEFINE-SETF-EXPANDER	✓	
DEFINE-SYMBOL-MACRO	✓	
DEFMACRO	✓	
DEFMETHOD	✓	
DEFPACKAGE	✓	
DEFPARAMETER	✓	
DEFSETF	✓	
DEFSTRUCT	✓	
DEFTYPE	✓	
DEFUN	✓	✓
DEFVAR	✓	✓

Table 1: A feature checklist for the Common Lisp operators whose name starts with "DEF". The "host" column refers to the cross-compiler environment, while the "target" column represents the Movitz run-time environment proper.

reference. In short, all the mechanisms that are to be juggled by operating system kernels are available through Lisp syntax.

The operators that interface hardware mechanisms do not try to enforce safety or consistency in any way. This is because the users of such functions are expected to know what they are doing, and because these same mechanisms are expected to be used to *implement* hardware-based protection in a system. This is in accordance with a general design principle in Movitz, that Common Lisp's dynamic safety features apply to Lisp programs and objects, but not to the system as such. Consequently, it is very easy to crash the system e.g. by loading a segment register with an illegal value. But at the same time it is possible this way to inspect and experiment with the most fundamental hardware features from Lisp programs, and even interactively from a read-eval-print-loop.

2.4 The compiler and development tools

The Movitz compiler is a Common Lisp cross-compiler, in the sense that this compiler and development environment runs e.g. on a regular desktop system and an implementation of ANSI Common Lisp, while it targets the Movitz run-time environment. The Movitz compiler can produce for example bootable disk images, or dynamically and incrementally modify running images.

The cross-compiler and related tools uses the notion of a "symbolic image", which is a representation of the Movitz target Lisp world using the data-structures native to the host Lisp system. For example, a cons-cell in the target image is represented by an instance of the `movitz-cons` standard-class in the symbolic image. A two-way mapping between this class and its binary representation in a bootable image is declared by means of the Binary-types library [4], and this mapping is invoked when "dumping" the symbolic image to a bootable kernel image. A modified lisp-mode in the Emacs editor installs key-bindings such that e.g. the `M-C-x` key-stroke invokes the Movitz compiler, operating on the current symbolic image.

There are more uses for the symbolic representation of images than to dump bootable images. Symbolic objects, for example representing compiled function objects, can be printed into textual forms that can be transported to, read, and evaluated by a running Movitz kernel. We have successfully experimented with this

Benchmark	CMUCL	Movitz	Factor	Movitz†
`traverse`	449	515	1.15	496
`boyer`	45	77	1.71	71
`triang`	493	2,336	4.74	2,303
`triang-lex`	483	608	1.26	559

Table 2: Some of the Gabriel benchmarks measuring Movitz against CMUCL. The units are 10^6 CPU cycles on a 466 MHz Intel Celeron CPU. The Movitz† column shows Movitz' performance when the stack boundary check in function preambles is removed, indicating an average overhead of about 3%.

strategy to provide true incremental compilation, transporting the textual forms from the development machine to the Movitz target over UDP/IPv6. Furthermore, we use the automatic mapping between symbolic and binary representations to interface Movitz kernels running in the Bochs x86 PC emulator [1], accessing the emulated memory and CPU state through the Unix procfs mechanism. This serves as a useful debugging tool, allowing inspection of memory and CPU state, and can provide stack backtraces in situations where this is otherwise impossible.

3 Experiences with Movitz and exploratory kernel-level programming

Does the interactive programming mode work when it comes to kernel-level programming, without the safety features enforced by typical operating systems? We first note that the Lisp machines of the 70s and 80s proved that Lisp can work well as an operating system implementation language. However, these systems often had the benefit of Lisp-specific micro-code and other hardware mechanisms that helped with safety and efficiency issues. Such luxuries are not available to Movitz.

Because it is difficult to "prove" that something is a viable software platform, we offer here only some anecdotal evidence.

It is our experience so far that Common Lisp and Movitz can work adequately as a kernel-level platform, even on commodity hardware. For example, we have developed a driver for an Ethernet network interface whose exact operation is non-obvious, and therefore required many cycles of testing, editing, and re-compiling. Rarely, if ever, did the system lock up in the event that something went wrong. Rather, we would see an error condition, or simply produce an erroneous Ethernet frame.

It does happen that the system locks up irreversibly. The reasons for this tend to be one of the following:

- Something goes wrong during boot-up, before the system has reached a sound state. For example, if it is tried to allocate memory before dynamic memory buffers have been initialized. This would trigger a condition, causing a new memory allocation that fails, and so on.

- There are sometimes unexpected errors in the debugger. The debugger is a very critical part of the system, because if for example an error occurs while printing the debugger's prompt, the debugger is called recursively, and we have another recursive error situation going.

As a consequence of the latter point, we continue to try to improve the safety of the basic debugger functionality, by checking that the environment is consistent in various ways. The compiler inserts stack-pointer boundary checks in the preamble

```
Bochs BIOS, $Revision: 1.13 $ $Date: 2004/04/16 14:04:55 $

Booting from Floppy...
Loading Movitz 3020..
()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()
()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()
Enter..Ok.
CPU features: FPU, TSC and APIC.
Movitz image Los0 build 3020.
INIT> (setf (interrupt-handler 0)
> (lambda (n frame)
>   (warn "Division by zero!")
>   (backtrace)
>   (abort)))
#<function :ANONYMOUS-LAMBDA>
INIT> (truncate 42 0)
Warning: Division by zero!
 <5 (:ANONYMOUS-LAMBDA ...)
 <= {Interrupt 0 in TRUNCATE at PC offset 42.}
 <3 (EVAL (TRUNCATE #x2a #x0))
 <= (READ-EVAL-PRINT)
 <= (GENESIS) --|
INIT> _
```

Figure 3: A screen-shot of a Movitz session from boot-up. First, a division-by-zero exception-handler is installed interactively. Then, this exception is triggered.

to all functions that are not obvious call-graph leaves, so that unbounded recursion in general is not a safety problem. We would also like to point out that should we compare Movitz against traditional C or assembly-based systems, these are expectedly much less likely to catch run-time failures and offer a stack-backtrace, and they are we believe less likely to offer a general mechanism for recovering the system to a consistent state, other than a re-boot.

Regarding program execution speed, we believe there is little difference between kernel-level applications and other applications in this respect, so the normal considerations about the efficiency of Lisp programs apply. While the quality of the compiler is important, we believe it has been demonstrated by others that Common Lisp compilers can produce code of comparable quality to any other system. Also, some of the initial rationale and motivation for the Movitz project was the observation that in many situations now, CPU cycles are abundant, and other parameters, such as code flexibility, safety, and ease of development are more important. Table 2 compares the Movitz compiler with that of CMUCL running on identical hardware, using some of the Gabriel benchmarks [6]. Our poor result in the `triang` benchmark is due to this benchmark's extensive use of special variables for temporary values, where currently Movitz naively performs a deep binding search at each such variable reference. The `triang-lex` benchmark is a modified version of `triang` altered to predominately use lexical variables. We chose to compare against CMUCL because it has a mature compiler, and provides a CPU-cycle-counting TIME macro, as does Movitz. Table 2 also gives an indication of the cost of pervasively performing safety checks of the stack pointer.

As an example of how Movitz can be used for system experimentation and programming, even interactively, we show in Figure 3 a screen-shot from a session where an exception-handler for the division-by-zero CPU exception (i.e. exception number zero) is installed interactively. The exception-handler is in this case an interpreted anonymous lambda function that emits a warning, uses a Movitz library function to display a stack backtrace, and performs an ABORT, which transfers control to the current REPL. Normally, the default exception-handler for the division-by-zero CPU exception signals a DIVISION-BY-ZERO error condition, which if not otherwise handled results in a debugger prompt.

4 Future directions

Our short-term goal is to fill in the most obvious gaps and generally improve Movitz as a Common Lisp implementation and application platform. In particular, we plan to include support for bignums and ratios, and to improve and complete the arithmetic operators. On the development tools side, we aim to improve support for "live" interactive development, having the emacs lisp-mode communicate and integrate with running Movitz images.

Our medium-term goal is to include support for the 64-bit extensions to the x86 architecture. We believe that under 64-bit architectures many of the comparative disadvantages of dynamically typed languages diminish, and that the current technology shift from 32 to 64 bit hardware represents an opportunity to increase the visibility of dynamically typed languages and systems.

Our longer-term goals is to investigate some of the following problems, using Movitz as the research platform:

- How can we design Movitz so as to be minimally intrusive with respect to application experimentation with hardware and software mechanisms, while at the same time providing abstractions and mechanisms that are useful and elegant? What are the relevant trade-offs, and can the results carry over to other kinds of platforms than Movitz?

- Can we find system abstractions such that we can exploit Lisp's strong introspective and dynamic capabilities so as to recognize situations where we can provide specialized algorithms and code-paths, and perhaps by such means outperform other systems?

- Can we realize the benefits of the emulated PC and procfs interface described in section 2.4 in real[2] hardware? And how can we enhance the run-time environment (and possibly the programming language) to support "outside" debugging, visualization, and program steering better? We suspect there is an untapped potential of dynamic systems in this area, and hope to use Movitz as a platform for such research.

- Can we design a mechanism to provide hardware-based safety (presumably based on paging) while keeping the single-address-space paradigm? In other words, is it viable to have programs that only have read access to system objects, and otherwise reduced privileges, and this way provide a multi-user Lisp system?

- Can we implement a trusted compiler and a system that can compile untrusted programs so as to produce trusted or otherwise "more safe" code, e.g. in style with SPIN[3]?

We believe that answers to these and similar questions can also carry over to other systems, not based on Movitz or Lisp.

5 Conclusion

Although Movitz might not currently be quite ready for prime-time production use, we believe we have already demonstrated that Common Lisp can serve quite well as implementation language for kernel-level applications and systems on commodity

[2] "Real hardware" can mean for example a serial cable and an appropriate interrupt handler on the target, much like what e.g. the GNU debugger `gdb` supports, or a bus-mastering PCI card with a micro-controller and high-speed interconnect that operates more independently of the target system.

hardware. Concerning the feasibility of a very policy free and architecture independent platform, we find we are making good progress. We look forward to continuing research based on the Movitz platform, as outlined in Section 4.

The Movitz platform's source code and other information is hosted at `http://www.common-lisp.net/project/movitz/`.

References

[1] The Bochs IA-32 emulator project. http://bochs.sf.net.

[2] ANSI X3.226:1994 programming language Common Lisp, 1994. Also available in hypertext form at www.lispworks.com/reference/HyperSpec/.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, 1995.

[4] Frode Vatvedt Fjeld. Binary-types. www.cs.uit.no/˜frodef/sw/binary-types/. Software library.

[5] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In *Symposium on Operating Systems Principles*, pages 38–51, 1997.

[6] R. P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press, Cambridge, Massachusetts, 1986.

[7] D. Gudeman. Representing type information in dynamically-typed languages. Technical Report TR93-27, Tucson, Arizona, 1993.

[8] Franz Inc. Streams in Allegro CL. www.franz.com/support/documentation/6.2/doc/streams.htm.

[9] G. Kiczales, J. des Rivires, and D.G. Bobrow. *The Art of the Meta-Object Protocol.* MIT Press, 1991.

[10] Jochen Liedtke. On micro-kernel construction. In *Symposium on Operating Systems Principles*, pages 237–250, 1995.