Writing an Operating System with Modula-3

Emin Gün Sirer Stefan Savage Przemysław Pardyak Greg P. DeFouw
Brian N. Bershad

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

November 3, 1995

1 Introduction

We are using Modula-3 to write an operating system called *SPIN* at the University of Washington [BSP+95]. Modula-3 is an ALGOL-like typesafe, high-level programming language that supports interfaces, objects, threads, exceptions, garbage collection, and generic interfaces. When we began the design of the *SPIN* operating system, we sought a language that would allow us to integrate potentially untrusted extension code in large and small quantities into the kernel. The primary goal of the system is to allow applications that require high performance system services to customize the operating system for a particular task. The system provides a core service infrastructure that provides threads, virtual memory, and device management together with an extension facility that allows application-specific services to be dynamically downloaded into the kernel. We are using the system to develop internetworking services, HTTP servers, video servers, and a general purpose UNIX environment.

We use Modula-3 as the language for both extensions and the base system in order to catch programming errors, and decrease integration complexity and system support overhead. The *SPIN* kernel and system runtime libraries consist of about 40,000 lines of Modula-3 code. The kernel also contains C and assembly code which we have liberally borrowed from the sources for DIGITAL UNIX. Our borrowed sources implement platform specific services, such as device drivers, and are available to the Modula-3 component of the system through about 80 functions in a dozen interfaces.

Despite the fact that the primary reference for Modula-3 is entitled "Systems Programming with Modula-3," [Nel91] we have found that the general systems community has remained skeptical of the language. Instead, they hold to languages such as C and C++ which offer little more than an environment for advanced assembly language programming. We believe that this skepticism is due to some misplaced concerns and misunderstanding that surround Modula-3, rather than any limitations of the language. The purpose of this paper is to help clear up some confusion about developing software with Modula-3. In particular, we will concentrate on using Modula-3 to write an operating system, which is where our primary experience lies.

The key point with which we hope to leave the reader is that there is a fundamental difference between a programming language, and a particular implementation of that language. For example, there are many different versions of the C compiler and its runtime utilities. In the 80's though, BSD UNIX's *pcc* and `libc.a` essentially defined the C programming language that many people use today. Gradually, improvements in the compiler and the libraries allowed people to hold C up as the "language to be beat" in terms of expressiveness and performance. A similar evolution occurred with C++, for which the first implementation (ATT's *cfront*) substantially underperformed C. Over time, though, the compiler and runtime improved, together with people's understanding about what they could and could not do efficiently with the language and its default runtime environment. Today, commercial C++ compilers generate code with similar quality

to C compilers, and programmers rarely complain about the implementation of the language (although complaints about the language itself litter the Usenet bulletin boards).

Today, the Modula-3 programming language is widely considered synonymous with its primary reference implementation from Digital's Systems Research Center (DEC SRC). The DEC SRC implementation is a publicly available, highly portable Modula-3 system that consists of a compiler front-end, a code-generator, a set of runtime services, standard libraries, a debugger, and a distributed object library. The compiler front-end translates Modula-3 source code into GNU RTL intermediate representation[Sta90], and a `gcc` based code generator emits object code directly. The goals of the DEC SRC implementation have always been portability (the system runs on 12 different architectures and 25 different operating systems) and functionality (the system's runtime services consist of over 230 interfaces).

Although the DEC SRC environment compilation environment excels at many tasks, there are several at which it comes up (or at one time came up) short. Unfortunately, it has been these past and present shortcomings of the *implementation* that have caused many in the systems development community to perceive the *language* as inadequate for use in a serious development environment. Since we intended to develop seriously, these perceptions caused us concern. Consequently, before choosing Modula-3 as our system development language, we compiled a list of shortcomings based on the "community's collective wisdom." In other words, we scanned the discussion lists and tried to understand what people were saying. In particular, the most common concerns about we saw about Modula-3 were:

1. Modula-3 programs are slow.
2. Modula-3 programs take too long to compile.
3. Modula-3 programs require too much memory.
4. Modula-3 threads are slow.
5. Modula-3 checked runtime errors result in program termination.
6. Modula-3 can not be used in a mixed-language environment.
7. Modula-3's dynamic storage management is expensive.

In building *SPIN*, we came to understand that these were concerns primarily about the system's runtime environment, and secondarily about the DEC SRC compiler. In the rest of this paper, we address each concern and describe how we have dealt with it in our system.

We hope that the reader does not interpret our discussion as being a resounding endorsement for the Modula-3 language. Indeed, we have found a few places where the language is deficient in the construction of large, extensible, safe systems. Most notably, these deficiencies arise in the use of operations that "ought" to be safe (vis a vis the language's definition) but are not, or are safe but ought not to be. For example, the language does not allow for safe casting operations, whereby a data structure is represented as a union of possible types, even though the cast would not create a situation that might result in a possible unchecked runtime error. In a companion paper [HFC+96] we describe some of the changes that we have made to the language and its compiler in order to satisfy these types of problems.

2 Evaluating the concerns

We discuss the major concerns about Modula-3 from several angles. Where the concern is related to performance, we present data that shows the concern is not intrinsic to the language, but relevant only to the reference implementation. Where the concern is related to functionality, we describe how we have modified the standard reference compiler or the runtime to provide the functionality whose absence or unacceptability is implied by the concern.

2.1 Modula-3, code quality, and performance

There are two aspects to the concerns about the quality of code generated from Modula-3. The first has to do simply with the quality of the compiler, and the second has to do with the cost of ensuring safety within a compiled module. Although early implementations of the compiler were simple translators into C, the DEC SRC Modula-3 compiler is grafted on top of the GNU optimizing compiler's back-end and takes advantage of some of the optimizations that work at the intermediate representation level and below.

Modula-3 is a straightforward imperative language, and with few exceptions, there is little about the language that makes it more difficult to compile efficiently than other languages such as C. However, like other languages, the quality of the compiler has a direct impact on the quality of the generated code. Table 1, for example, illustrates the difference in code quality that results when using different compilers to generate semantically equivalent executables. The table shows the execution time of the MD-5 digital signature algorithm implemented in Modula-3 using the DEC SRC compiler and the Vortex research compiler [Gro95] being developed at the University of Washington. We also measured the execution time of the same program written in C and compiled with GCC. The C version included no explicit runtime array bounds checks, and we also compiled the Modula-3 version without bounds checking. The MD-5 algorithm was run 100 times against a block size of 1024 bytes.

Benchmark	C GCC	M3 DEC SRC	M3 Vortex
MD5	19.1	27.8	19.8

Table 1: *Comparison of GCC, the DEC SRC Modula-3 compiler and the Vortex based Modula-3 compiler on MD5. All versions are unsafe, as they execute without any runtime bounds checking. Times are in seconds and gathered on a SparcStation 20/61 60MHz with 128 MB main memory. We hope to present numbers from the DEC ALPHA for the Vortex compiler by the time this position paper is published. We will also include some non-array intensive benchmarks.*

2.1.1 Bounds Checking

Modula-3's safety guarantees require that the compiler ensure that there are enough runtime checks in place to prevent an unchecked runtime error from occurring within a module that is explicitly marked as `SAFE`. Most checks can be performed statically when the program is compiled. Ensuring the safety of array accesses, however, can require dynamic checks that can impact the performance of array intensive applications.

The DEC SRC Modula-3 compiler is extremely conservative about runtime checks for array bounds violations, prefacing every index operation with a check against the bounds. Moreover, the front end is not very aggressive in the RTL that it passes on to gcc, and so opportunities for removing some of these runtime checks through optimization are lost. As mentioned in the introduction, this is not an issue with the language, but with the implementation of the language. Techniques to eliminate unnecessary range checks are well-known within the compiler community[Gup90, Gup93, KW95], and are clearly applicable to Modula-3. For example, when we apply the bounds checking optimizations within the Vortex compiler, MD-5 execution time with bounds checking enabled goes from 27.6 seconds to 23 seconds.

In practice, we have not had incentive to write code fragments in any language other than Modula-3 strictly to avoid inefficiencies in generated code. In a few places, we have retreated to assembly language (the system's *bcopy bzero*, and checksum routines), but these functions are generally written in assembly even in systems built using C.

2.2 Modula-3 compiler execution time

The DEC SRC Modula-3 compiler consists of a platform-specific back end and a mostly platform-independent front end. The back end of the Modula-3 compiler is a slightly modified version of the GNU optimizing compiler (based on gcc 2.5.7). The front end of the compiler is written in Modula-3, and produces a textual

representation of the compiled program in GNU's *Register Transfer Language* (RTL). It then forks off the back end which reads the RTL from the file system to generate a standard COFF object file.

Because of the amount of machinery involved in each compile, we initially had concerns that the compile phase of the "edit/compile/reboot" cycle would be dominated by compile/link time. We have all had to work with systems where the turnaround time for a simple one-line kernel change was 10 to 20 minutes, and we did not want to repeat that experience. As it turns out, we are able to iterate through development cycle quite rapidly. Once our kernel has been built, changes to individual files can be compiled and linked into the kernel image in under a minute. Moreover, our support for dynamic linking allows us to safely load new code modules into the kernel without having to perform a complete relink and reboot. Although a full kernel build takes about 15 minutes, we do it rarely.

2.3 Modula-3 and code size

Our next concern had to do with the size of Modula-3 executables. Starting with the initial versions of the reference implementation, the language had a reputation of requiring massive executables. Indeed, using the DEC SRC reference implementation on the Alpha, the simplest client is almost 500 KB when statically linked with debugging enabled. This massive size is due to three factors: the language's requisite core services such as garbage collection and exception handling, the *extra* services that are thrown in "for free" such as streaming readers and writers, and the fact that the runtime does not support dynamic linking and sharing of code and data.

We were able to strip out 25% of the runtime simply by identifying those portions which were not needed by the operating system. We also created a dynamic linking facility [BSP+95, SFPB96] that eliminates multiple copies of the runtime by allowing clients access to shared code and data. As a case in point, our HTTP server extension consists of 392 lines of Modula-3 code and consumes 9KB of memory (5KB text + 4KB data).

2.4 Modula-3 and threads

A third concern of ours was the cost associated with using the Modula-3 thread and synchronization services. Since threads are a fundamental component of our operating system, and since synchronization is a critical service required within an operating system, we initially felt that it would be necessary to abandon the language's thread interface and introduce one of our own. We were reluctant to do this from a practical standpoint, since the Modula-3 threads interface is known to the compiler, and influences other language services such as exceptions. In addition, we wanted our kernel programming environment to be as "vanilla" as possible to shorten the learning curve of people trying to develop code for the system.

Indeed, it is true that the standard Modula-3 threads package that comes with the reference implementation performs poorly. For example, on the 133 Mhz DEC Alpha, to spawn and terminate a new thread takes over 700 *usecs*. While slow, this is the level of performance that can only be expected from any threads package that implements its services entirely at user-level on top of the UNIX process model. Intrinsically, performance has nothing to do with the thread interface, or the language's model of how threads interact with other system services.

As part of the initial development of our kernel, we reimplemented the Modula-3 thread, scheduler and synchronization services directly on top of a lightweight kernel threading interface called *strands*[BSP+95]. The strands interface allows thread packages and schedulers to be tightly integrated with each other as well as with their clients. In comparison to the 700 *usecs* required to create a thread on top of a UNIX process, a Modula-3 thread based on a *SPIN* strand can be created and terminated in 22 *usecs*. From this, and from other low-overhead thread operations that we perform using the Modula-3 threads interface, we find no evidence of any problem with the interface itself.

2.5 Modula-3 failure semantics

Modula-3 allows programmers to raise and catch exceptions during the course of program execution. If a thread raises an exception, the runtime walks through the raising thread's stack looking for a handler to catch the exception. If a handler is found, control is passed to it. If not, the thread is said to have committed a *checked runtime error*, for which the language leaves the system's behavior unspecified. In the reference implementation from DEC SRC, checked runtime errors result in program termination; there is no way for a thread to recover from its own, or another's checked runtime error.

For the *SPIN* operating system, as well as many other environments where programmers are willing to work hard to ensure liveness, the reference implementation's interpretation of the semantics of a checked runtime error is inadequate. Consequently, we changed the runtime system's implementation of exceptions, within the specifications of the language, such that users are notified of runtime failures through language exceptions. With this addition, code can implement failure recovery by installing an exception handler for runtime exceptions and taking remedial action.

2.6 Using Modula-3 in a mixed language environment

As previously mentioned, our kernel relies on some low level platform-specific services that we borrow from the sources of DIGITAL UNIX. These sources are written in C, and therefore directly highlight the concern that interfacing Modula-3 to other languages can be difficult. There are three aspects to this difficulty: safely calling foreign code from Modula-3, calling Modula-3 code from foreign code, and passing data between them.

In the case of calling out from Modula-3, the reference implementation defines a pragma that allows an interface to describe a function written in a foreign language such as C. In the reference implementation, this pragma is legal in both safe and unsafe interfaces, implying that a safe module could import a safe interface that provided direct access to a C function or data structure of arbitrary type. Since the type of the function or data structure may in fact be specified by the C implementation, we found that this flexibility introduced an unacceptable safety risk. Consequently, we modified the compiler so that the externalizing pragma could only be used within unsafe interfaces.

To call from C to Modula-3, we modified the front end of the compiler to generate C header files for Modula-3 interface files. In this way, any function exported via a Modula-3 interface can be called directly from C using "module dot method" syntax.

We believe that the concern about passing data between Modula-3 and other languages stems from the fact that the Modula-3 heap is automatically managed in ways that are not consistent with sharing memory between safe and unsafe languages. For example, the DEC SRC reference implementation uses a copying garbage collector. Consequently, if a Modula-3 program passes a collectible reference to C, and C stores the reference in it's own uncollectible heap, the collector might copy the object leaving C's reference dangling. Modula-3 provides both collectible and uncollectible heap space (traced and untraced), so this problem can be avoided by allocating shared objects from the appropriate heap. Sometimes, though, it is not possible to predict whether or not a reference will be passed across the language barrier at the time it is allocated. Indeed, this property is something that ought to be hidden within the modules that implement the pass through to the foreign language. For this reason, we introduced the notion of a *Strong Reference*, which is an object allocated from the traced heap that the collector should consider as temporarily uncollectible and immovable. In this way, an object can be "strong reffed" before it is passed to C.

To pass data from a foreign language (typically C, although occasionally assembly language) to Modula-3, we pass the data by reference from the foreign language, and follow one of two strategies on the Modula-3 side. We either declare the in parameter as a reference to a record stored on the uncollectible heap (untraced), or we declare it as variable parameter (for which the compiler generates code that automatically dereferences the in parameter). Neither approach has been particularly complicated to use.

2.7 Dynamic storage management

Lastly, we were concerned that the overhead of dynamic storage management in Modula-3 would be prohibitive. We anticipated two types of overhead here: allocation and collection. Allocation overhead is that incurred when a thread creates a new object and there is plenty of free space in the heap. Collection overhead is that incurred in order to ensure free space.

With respect to allocation, there is no fundamental reason why Modula-3's allocation overhead should be any larger than that of a standard C `malloc()` package, since both are implementing the same service: locating a block of memory having a specific size from a free list. Nevertheless, we have measured allocation overheads on the Alpha using the DEC SRC reference implementation that are roughly four times higher than for malloc. The reason for this is simple: the DEC SRC allocator was not built to be fast (for example, it maintains statistics about memory usage on its critical path). We were able to cut down allocation overhead by almost a factor of two simply by removing code that had nothing to do with allocation. We are presently working on a complete reimplementation of the allocator for which overhead will be comparable to C's. In any event, allocation overhead has not been a serious problem in the system we've built so far because we avoid allocation altogether on critical paths such as interrupt handling and thread management. In the few instances where allocation latency is an inherent component of the critical path, e.g. thread fork, we use the common technique of preallocation and caching of initialized objects to shift the performance penalty to less critical periods.

The second concern we had about dynamic storage management was collector overhead, specifically the pauses incurred by major collections. Currently, we use a single-threaded collector which requires that all other concurrent execution, including interrupt handlers, be suspended while it is activated to avoid mutations of the heap. A typical collection takes about 100 ms. on our platform, which introduces perceptible delays into the system.

We are dealing with the collector overhead in two ways. The first is to use a better collector. We are examining concurrent and incremental garbage collection techniques [SG95, AEL88] to reduce disruptive system pauses. While a better collector can reduce the pause times, it will not directly address the overhead problem. If garbage is created, there is going to be a penalty to clean it up. Consequently, we also adopt a "pro-recycling" attitude within the system, and encourage clients to reuse objects they have allocated without requiring that they be collected and reallocated. As long as the type of an allocated and discarded object remains unchanged, client code may recycle the object for reuse. High-throughput subsystems that deal with large quantities of data, such as devices and the network stack, implement a notification scheme by which they indicate when a particular item can be reused by their clients. For example, our video server allocates a buffer, fills it in with data from secondary storage, and invokes the network stack by calling `UDP.Send`. The buffer makes its way through the stack down to the device driver, which eventually raises the `MBuf.Freed` event to indicate that the buffer can be reused. The video server can then recycle this packet to hold other data, without having to wait for a collection and an allocation. Consequently, client working sets are stable in steady-state, decreasing pressure on the garbage collector.

Even though we reduce collector involvement in our system, the collector still serves two important purposes: it allows safe reuse of memory between unrelated applications, and it acts a safety net for extensions that are unwilling to follow the memory recycling protocol. Nevertheless, we realize that our current situation with respect to the collector is not ideal – it is controlling us, rather than we it. We intend to dedicate more effort to this problem with the expectation that we can reduce collection overhead to an acceptable level.

3 Summary

In this paper, we have discussed the use of Modula-3 in the construction of an operating system. We have shown that criticisms about a programming language can be misplaced, and are often better directed towards a particular implementation of that language or its runtime. Moreover, we have shown that it is possible to construct a high performance implementation of both. We conclude that the safety of Modula-3 combined with its support for systems programming make the language an ideal choice for an systems programming.

Not only does Modula-3 prevent most common programming errors by virtue of its typesafety, it offers a variety of powerful tools that allow the programmer to tackle a range of systems programming tasks. We have found the safety of the language, as well as its data hiding properties, generic interfaces and object support to be effective in developing a high-performance, modular system and its extensions.

References

[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of ACM SIGPLAN '88 Conf. on Programming Language Design and Implementation*, June 1988.

[BSP+95] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin G n Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995.

[Gro95] Cecil Group. UW Vortex Compiler, An Optimizing Compiler for Object-Oriented Languages. http://www.cs.washington.edu/research/projects/cecil/www/cecil-home.html, December 1995.

[Gup90] Rajiv Gupta. A fresh look at optimizing array bound checking. *SIGPLAN Notices*, 25(6):272–282, June 1990. In ¡italic¿Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation¡noitalic¿.

[Gup93] Rajiv Gupta. Optimizing array bounds checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1):135–150, March 1993.

[HFC+96] W.C. Hsieh, M.E. Fiuczynsko, C.Garrett, S.Savage, D.Becker, and B.N. Bershad. Language Support for Extensible Systems. Submitted to the First Workshop on Compiler Support for Systems Software, November 1996.

[KW95] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. *SIGPLAN Notices*, pages 270–278, June 1995. In ¡italic¿Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation¡noitalic¿.

[Nel91] Greg Nelson, editor. *System Programming in Modula-3*. Prentice Hall, 1991.

[SFPB96] E.G. Sirer, M. Fiuczynski, P. Pardyak, and B.N. Bershad. Safe Dynamic Linking in an Extensible Operating System. Submitted to the First Workshop on Compiler Support for Systems Software, November 1996.

[SG95] Jacob Seligmann and Steffen Grarup. Incremental mature garbage collection using the train algorithm. In *Proceedings of ECOOP'95, Ninth European Conference on Object-Oriented Programming*, volume 952, pages 235–252, 1995.

[Sta90] Richard M. Stallman. Using and porting GNU CC. Technical report, Free Software Foundation, Cambridge, MA, 1990.