Writing an Operating System with Modula-3

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

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

1 Introduction

We are using Modula-3 to write an operating system called *SPIN* at the University of Washington [Bershad et al. 95]. 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.

When we began the design of the *SPIN* operating system, we sought a language that would help us integrate untrusted extension code into the kernel. We converged on Modula-3 as the implementation language for both extensions and the core system. Modula-3 is an ALGOL-like, typesafe, high-level programming language that supports interfaces, objects, threads, exceptions and garbage collection. There is a well-defined, safe subset of Modula-3 which allows untrusting clients to securely share the same address space. Most of the *SPIN* kernel and extensions, which compromise about 40,000 lines of Modula-3 code, are written in this safe subset. 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 titled "Systems Programming with Modula-3," [Nelson 91] 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*) [Stroustrup 83] substantially underperformed C. Over time, though, the compiler and runtime improved,

[0] This research was sponsored by the Advanced Research Projects Agency, the National Science Foundation (Grants no. CDA-9123308 and CCR-9200832) and by an equipment grant from Digital Equipment Corporation. Bershad was partially supported by a National Science Foundation Presidential Faculty Fellowship. Sirer was supported by an IBM Graduate Student Fellowship.

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[Stallman 90], 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 excels at many tasks, there are several at which it comes up short. Unfortunately, it has been these 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 seriously intended to develop a production quality operating system, 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 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 violate typesafety. In a companion paper [Hsieh et al. 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 with the cost of ensuring safety within a compiled module, and the second has to do simply with the quality of the compiler.

2.1.1 Safety and Compiler-Generated Dynamic Checks

Modula-3's safety guarantees require that the compiler ensure that there are enough runtime checks in place to prevent a violation of typesafety 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, 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[Gupta 90, Gupta 93, Kolte & Wolfe 95], and are clearly applicable to Modula-3. The Vortex research compiler [Chambers et al. 95], being developed at the University of Washington, performs redundant range check elimination among other optimizations and typically eliminates 30-90% of range checks in our benchmarks.

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.

2.1.2 Compiler Quality

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.

Benchmark	C	M3 Unsafe		M3 Safe	
	GCC	DEC SRC	Vortex	DEC SRC	Vortex
MD5	17.0	17.5	17.8	21.4	18.5
hotlist	15.3	15.3	15.4	15.3	15.4
lld	43.5	47.0	50.7	51.1	53.7
Richards	99.0	87.8	88.6	88.8	91.3

Table 1: *Comparison of GCC, the DEC SRC Modula-3 compiler and the Vortex based Modula-3 compiler. Times are in seconds and were gathered on a dedicated DECStation 4000/400 133 MHz with 64 MB main memory. The mean of 10 timings is reported.*

Table 1 illustrates the difference in code quality that results when different compilers generate semantically equivalent executables. The C versions of the benchmarks offer no safety guarantees. Safe Modula-3 measurements, on the other hand, include compiler generated dynamic checks wherever necessary which ensure that all memory accesses are legitimate. The unsafe Modula-3 measurements, which omit all dynamic checks, are included to demonstrate the baseline performance of the compilers in relation to GCC.

Our benchmarks consist of common operating system tasks and typical operating system extensions [Bershad et al. 95, Campbell & Tan 95, Small & Seltzer 94]. *MD5* is a digital signature algorithm [Rivest 92] implemented in Modula-3. It is very array intensive which results in the DEC SRC compiler placing 10 range checks in its inner loop. However, redundant bounds check elimination in Vortex is able to eliminate all but one of these checks and reduce the performance difference with C to less than 10% for safe code. *Lld* is a logical disk simulator that is similarly array intensive. Despite this fact, the overhead of dynamic bound checks due to safety is roughly 5-8% of the total execution time. We found that the quality of code generated for arithmetic operations such as modulus had a much greater impact on performance than dynamic checks. *Hotlist* is a page eviction simulation, where the benchmark keeps track of an LRU list of eviction candidates and traverses the list repeatedly in response to page fault events. In this instance, Modula-3 can guarantee safe behaviour without recourse to any dynamic checks. Hence the C and Modula-3 versions of the benchmark have essentially identical performance. *MD5*, *hotlist* and *lld* were contributed by Small [Small & Seltzer 96]. *Richards* is an operating system simulator with a synthetic workload of process creation and termination. The coding style encouraged by Modula-3 allows the compiler to produce better code for a commonly executed switch statement, resulting in higher performance than C, even for safe code.

From these results, other measurements [Small & Seltzer 96], and our experience with the *SPIN* kernel, it seems evident that Modula-3 compiler technology is currently comparable to that of C, yet offers stronger guarantees about program safety.

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 compilation, 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 the development cycle quite rapidly due to Modula-3's support for incremental recompilation. 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 [Bershad et al. 95, Sirer et al. 96] 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 fourth 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 part of the language definition. 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 a 133 Mhz DEC Alpha, to spawn and terminate a new thread takes over 700 $\mu secs$. 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, thread performance is tied much more to the particular implementation method rather than the interface that is exported to the clients.

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* [Bershad et al. 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 $\mu secs$ 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 $\mu secs$. 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. In the event that a thread fails to do so, another thread can collect the exception delivered to the failed thread.

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, we found that type mismatch errors could occur on the language boundary, since type information is typically not propagated between languages. Such a mismatch could potentially be used to circumvent the typesafety of Modula-3. In particular, the reference implementation defines a pragma that allows an interface to refer to a function written in a foreign language such as C. In the reference implementation, this pragma is permissible in safe interfaces, which allows untrusted code to provide direct access to C functions or data structures of arbitrary type. Since the compiler cannot enforce type-checking across the language boundary and has to trust the user-supplied type definition for external symbols, there is 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.

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 the explicit storage management of C. 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 communicate with 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 [Seligmann & Grarup 95, Appel et al. 88] 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 some protocol and trust is established between cooperating modules, objects of a given type can be shared and reused without collector or allocator involvement. High-throughput subsystems that deal with large quantities of data, such as device drivers 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 by the application. 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 remain bounded 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 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

[Appel et al. 88] Appel, A. W., Ellis, J. R., and Li, K. Real-time concurrent collection on stock multiprocessors. In *Proceedings of ACM SIGPLAN '88 Conf. on Programming Language Design and Implementation*, June 1988.

[Bershad et al. 95] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., and Chambers, C. 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.

[Campbell & Tan 95] Campbell, R. H. and Tan, S.-M. μChoices: An Object-Oriented Multimedia Operating System. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, pages 90–94, Orcas Island, WA, May 1995.

[Chambers et al. 95] Chambers, C., Dean, J., and Grove, J. Vortex Compiler, An Optimizing Compiler for Object-Oriented Languages. University of Washington. http://www.cs.washington.edu/research/projects/cecil/www/cecil-home.html, December 1995.

[Gupta 90] Gupta, R. A Fresh Look at Optimizing Array Bound Checking. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 272–282, June 1990.

[Gupta 93] Gupta, R. Optimizing Array Bounds Checks Using Flow Analysis. *ACM Letters on Programming Languages and Systems*, 2:135–150, March 1993.

[Hsieh et al. 96] Hsieh, W. C., Fiuczynski, M. E., Garrett, C., Savage, S., Becker, D., and Bershad, B. N. Language Support for Extensible Systems. In *First Annual Workshop on Compiler Support for System Software*, January 1996.

[Kolte & Wolfe 95] Kolte, P. and Wolfe, M. Elimination of Redundant Array Subscript Range Checks. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 270–278, June 1995.

[Nelson 91] Nelson, G., editor. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[Rivest 92] Rivest, R. The MD5 Message-Digest Algorithm. Request for Comments 1321, Internet Engineering Task Force, April 1992.

[Seligmann & Grarup 95] Seligmann, J. and Grarup, S. 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.

[Sirer et al. 96] Sirer, E. G., Fiuczynski, M., Pardyak, P., and Bershad, B. N. Safe Dynamic Linking in an Extensible Operating System. In *First Annual Workshop on Compiler Support for System Software*, January 1996.

[Small & Seltzer 94] Small, C. and Seltzer, M. VINO: An Integrated Platform for Operating System and Database Research. Technical Report TR-30-94, Harvard University, 1994.

[Small & Seltzer 96] Small, C. and Seltzer, M. A Comparison of OS Extension Technologies. In *Proceedings of the 1996 Winter USENIX Conference*, San Diego, CA, January 1996.

[Stallman 90] Stallman, R. M. Using and Porting GNU CC. Technical report, Free Software Foundation, Cambridge, MA, 1990.

[Stroustrup 83] Stroustrup, B. Adding Classes to the C Language, An Exercise in Language Evolution. *Software—Practice and Experience*, 13(2):139–161, February 1983.