

Programming Language Challenges in Systems Codes

Why Systems Programmers Still Use C, and What to Do About It

Jonathan Shapiro, Ph.D.
Systems Research Laboratory
Department of Computer Science
Johns Hopkins University

October 22, 2006

Abstract

There have been major advances in programming languages over the last 20 years. Given this, it seems appropriate to ask why systems programmers continue to largely ignore these languages. What are the deficiencies in the eyes of the systems programmers? How have the efforts of the programming language community been misdirected (from their perspective)? What can/should the PL community do address this?

As someone whose research straddles these areas, I was asked to give a talk at this year's PLOS workshop. What follows are my thoughts on this subject, which may or not represent those of other systems programmers.

1 Introduction

Modern programming languages such as ML [16] or Haskell [17] provide newer, stronger, and more expressive type systems than systems programming languages such as C [15, 13] or Ada [12]. Why have they been of so little interest to systems developers, and what can/should we do about it?

As the primary author of the EROS system [18] and its successor Coyotos [20], both of which are high-performance microkernels, it seems fair to characterize myself primarily as a hardcore systems programmer and security architect. However, there are skeletons in my closet. In the mid-1980s, my group at Bell Labs developed one of the first large commercial C++ applications — perhaps *the* first. My early involvement with C++ includes the first book on reusable C++ programming [21], which is either not well known or has been graciously disregarded by my colleagues.

In this audience I am tempted to plead for mercy on the grounds of youth and ignorance, but having been an active advocate of C++ for so long this entails a certain degree of *chutzpah*.¹ There is hope. Microkernel developers seem to have abandoned C++ in favor of C. The book is out of print

¹ *Chutzpah* is best defined by example. *Chutzpah* is when a person murders both of their parents and then asks the court for mercy on the grounds that they are an orphan.

in most countries, and no longer encourages deviant coding practices among susceptible young programmers.

A Word About BitC Brewer *et al.*'s cry that *Thirty Years is Long Enough* [6] resonates. It really *is* a bit disturbing that we are still trying to use a high-level assembly language created in the early 1970s for critical production code 35 years later. But Brewer's lament begs the question: why has no viable replacement for C emerged from the programming languages community? In trying to answer this, my group at Johns Hopkins has started work on a new programming language: BitC. In talking about this work, we have encountered a curious blindness from the PL community.

We are often asked "Why are you building BitC?" The tacit assumption seems to be that if there is nothing fundamentally new in the language it isn't interesting. The BitC goal isn't to invent a new language or any new language concepts. It is to integrate existing concepts with advances in proven technology, and reify them in a language that allows us to build stateful low-level systems codes that we can reason about in varying measure using automated tools. The feeling seems to be that everything we are doing is straightforward (read: uninteresting). Would that it were so.

Systems programming — and BitC — are fundamentally about engineering rather than programming languages. In the 1980s, when compiler writers still struggled with inadequate machine resources, engineering considerations were respected criteria for language and compiler design, and a sense of "transparency" was still regarded as important.² By the time I left the PL community in 1990, respect for engineering and pragmatics was fast fading, and today it is all but gone. The concrete syntax of Standard ML [16] and Haskell [17] are every bit as bad as C++. It is a curious measure of the programming language community that nobody cares. In our pursuit of type theory and semantics, we seem to have forgotten pragmatics and usability. Transparency gave way to abstraction, and then to opacity. The systems programmer *requires* an intuitive model of what is happening on the machine at the source code level.

If indeed it turns out that BitC has nothing new, I think that my group will be very well pleased. Unfortunately,

² By "transparent," I mean implementations in which the programmer has a relatively direct understanding of machine-level behavior.

there *are* going to be innovations in BitC, and this talk will try to explore some of the factors that are driving us.

2 Defining Terms

Before proceeding further, it seems appropriate to define what I mean by “systems programming” — or more precisely, by “systems programs.” These programs seem to have several common properties:

Systems programs operate in constrained memory. This is obvious in embedded systems, but modern general-purpose kernels operate very near the limits of available virtual memory. On a 32-bit IA-32 class machine with a fully populated 64 gigabyte physical memory, it is *extremely* difficult to get the per-page book-keeping data structures to fit within a 0.5 gigabyte kernel virtual region. Two-space storage managers are simply unthinkable, and the majority of the reachable kernel working set cannot be mapped into the virtual address space at any given time. The implications for garbage collectors are unpleasant.

Systems programs are strongly driven by bulk I/O performance. In applications that move large amounts of data, the goal is to have *zero* copies of that data, even as headers and trailers are being added to that data. This is typically done by designing a data structure that reserves space for the headers and trailers in advance, which entails the creation of pointers that do not point to the beginning of an object. Worse, these objects may be multiply aliased in both the virtual and physical maps (e.g. by DMA). Those aliases that are held by hardware require explicit management. Data may be gathered or scattered in the course of handling. It is often desirable for these sub-objects to reference the storage of the original. The implications for property checking are distressing.

Performance matters. At least in the critical paths of systems programs, the addition of only a few cache references — or worse, cache misses — in an inner loop can mean the difference between winning and losing customers. It follows that **data representation matters.**

This is not especially true in client-side code, where languages like Java and C# are establishing success. On the desktop, computations are available in excess and the limiting factor in performance is generally the user, who is a surprisingly slow peripheral. The corresponding servers, however, must serve thousands of these clients simultaneously, and must do so without excessive memory requirements. They must also be cooled.

In systems code, the effect of representation and data placement can be extreme. Bonwick *et al.* discuss some of these effects [5], noting that the performance of system-level benchmarks can change by 50% through careful management of cache residency and collisions.

Systems Programs Retain State. The most common pattern in systems programs is event loops. While there is short-lived data within these processing loops, the pattern overall is that there are (relatively) large amounts of state that live for the duration of the event processing cycle. This tends to penalize the performance of automatic storage reclamation strategies. To make matters more in-

teresting, there are caches. These update frequently enough that “old space” techniques may not perform well, but are large enough that scanning them can be quite expensive. Our experiences with the Boehm collector in OpenCM were that storage leaked very liberally — enough so to crash the OpenCM server frequently. It follows that **user-managed storage is a requirement**, but perhaps not in fully general form.

3 Four Fallacies

Factors of 1.5 to 2 don’t matter. It is sometimes argued that a factor of 1.5 to 2 isn’t important. CPUs continue to increase in speed, and surely this will make up the gap.

The facts say otherwise. The annual cost to operate a large banking data center today is \$150,000 per square foot. It is by far the most expensive real estate in the world, and more than one third of that cost is the cost of cooling the data center. The general rule of thumb is that power is proportional to V^2F : the square of the voltage times the frequency. Most of this power is wasted as heat. To a system’s programmer, the cost of doubling the clock rate is \$50,000 per square foot per machine room. Raising the clock rate decidedly isn’t free, and walking into the network distribution closet at your business or school will quickly convince you that *current* power usage is excessive.

Boxed representation can be optimized away. Bagioni’s remarkable success with a TCP/IP implementation in Standard ML [3] is sometimes used to support the contention that the compiler can manage representation issues. Derby’s technical report [8] is more revealing. On interactive traffic, FoxNet achieves 0.00007 times the performance of the then-current UNIX network stack. That is not a typo. On *large* blocks, which is by far the cheapest case to implement, FoxNet imposes *11 times* the CPU load of the network stack implemented in C. Large block processing costs are dominated by memory bandwidth, not software overheads. As Blackwell discusses [4], processing overhead on smaller packets is necessarily much higher.

Further work will certainly improve on Bagioni’s results, but the UNIX stack is much faster now than it was then too. Ultimately, the reason for this is that the UNIX networking stack exploits three unsafe strategies: (1) the ability to directly express data representation that carefully respects that behavioral constraints of the data cache, (2) the ability to alias the insides of data structures, and (3) explicit storage management in the form of recycled buffer and header data structures — this is not merely a storage optimization. The buffer structures have alignment and virtual memory address binding constraints that enable them to exploit direct DMA.

There appears to be a fundamental expressiveness problem here: because DMA is involved, data structure alignment and virtual address binding constraints are not merely matters of convenience or performance. They are requirements of correctness. I am not aware of any managed storage approach that is likely to perform well while supporting this type of constraint.

The optimizer can fix it. Hypothetically, modern commodity processors can sustain a 3 to 4 instruction issue rate

per cycle. This may or may not be typical of applications codes in the wild. It certainly is *not* true of kernel code, where (a) a very large proportion of branches are data dependent, and (b) neither the compiler nor the hardware are able to establish any useful issue distance between the load of that data and the branch nor schedule other instructions between them. For embedded processors, of course, considerations of power still dominate, and out-of-order issue implementations remain quite rare.

For microkernels, an issue rate of 1.5 to 1.7 in the critical path is extremely hard to achieve, *and can generally be obtained only with carefully hand-optimized code*. This optimization frequently entails exploiting non-obvious conditions that the programmer knows, as a matter of application-specific invariants, correspond to the tests that would naturally have been written in a more “direct” implementation. In the microkernel IPC path, it is generally agreed that C code is not fast enough and carefully hand-tuned assembly code is needed. Writing this kind of code in a language like ML simply isn’t an option.

I emphasize that the problem here is not merely a matter of applying global optimization. Compilers have a difficult time generating code when instruction selection is tightly constrained, and the optimization strategy available to the compiler does not have access to application-specific invariants that are known to the programmer. In any case, the control paths of systems code are dominated by proximate data dependencies. There is relatively little automatable optimization to be had in these codes. The best way for the programming language designer to help is to ensure that the language can express algorithms that have a direct and transparent transformation into low-level assembly code.

The legacy problem is insurmountable. It is a popular assertion that so much code is written in C and C++ that we are stuck with them forever. People said the same thing about MVS, and later UNIX. From this axiom, we may directly derive the conclusion that Windows, Linux, Java, and C# do not exist. The reality, of course, is that each of these systems started from scratch and succeeded. While it *is* comforting to think of them as the products of mass delusion, life is what happens while logicians are making other plans.

It is okay to start from scratch.

4 Challenges

Application constraint checking The wonder of systems codes is that they work at all, and it may be instructive to speculate on why this is so. Operating system kernels and databases, for example, seem to be characterized by a rich invariant structure. Some examples from the EROS kernel (see [19]):

- **Atomicity** No path may make a semantically observable change to kernel state until all preconditions are satisfied. When preconditions are satisfied, a “commit point” is declared and semantically significant mutation can occur.
- **Fail-Fast** Following a commit point, a system call

may result in a valid response or an architected error. Any runtime error must be handled by an immediate system restart *without* allowing state to be committed to persistent store.

- **Caching Constraint** If a capability to an object is in the “prepared” state, the target object must be in memory.

Other systems have very different constraints. The point is that all operating systems *have* such constraints, and their developers maintain a mental list of these constraints while programming. This serves to impose a human-implemented check on the process of coding. It *should* be possible in principle to annotate programs in a way that lets these constraints be *automatically* checked. Hao Chen and I did some initial work in this direction [7] with promising success, but concluded that capture of application-specific invariants requires something much richer than a simple model checker. This ultimately motivated our work on BitC. SLAM [2] is an interesting step in the right direction, but it is not straightforward for end-developers to extend its properties. Ultimately, any tool that operates on C code faces an imprecise language, a weak type system, and undecipherable aliasing effects. C wasn’t designed as a language to facilitate the expression of global properties.

Idiomatic Manual Storage While systems programs use manual storage management, they do so in highly idiomatic ways. When combined with a rich application-defined constraint environment, much of the storage management behavior of a modern operating system kernel should be checkable. The challenge is to provide mechanisms in the programming language (more precisely: in a constraint metalanguage) that allow programmers to *express* these constraints. One difficulty is that global constraints are difficult to express using pre- and post-conditions, because these are constrained by what is lexically in scope. Finding richer ways to deal with this is one of the things that we will be experimenting with in BitC.

Representation It is hopefully clear by now that direct control over data structure representation is essential in systems programs. The semantics of unboxed datatypes is straightforward, but few modern language designers seem to feel that representation is significant. Related to this, there is a tendency to eliminate multiple fixed-precision integral types, which are the primary types used by systems codes. Diatchki *et al.* [9] have made an interesting start on the representation problem. BitC is a more comprehensive approach, and by and large we have not found that direct support for representation provides any interesting difficulties in of itself. The difficulty lies in simultaneous support for representation, state, and polymorphism, which is the subject of Sridhar’s paper in the current proceedings [23]. In email exchanges, Diatchki has reported that his enhancement to Haskell has had a significant impact on the implementation of their prototype operating system. Representation matters!

State For systems and kernel codes, state is never going away. These codes are *defined* by I/O and in-place transformation of data. While it is possible to express all of these operations in monadic form, as the House project [11] is doing, there are no examples of large, complex, and deeply

stateful codes that have been maintainably constructed in this fashion. My group has spent some time writing code in ACL2 using the STOBJ mechanism [14], which provides single-threaded state. Our conclusion is that it doesn't scale in the human sense. Kernel programmers already operate at (or slightly beyond) the outer limit of systemic complexity that can be maintained in a programmer's head.

This is a large part of why transparency of expression in systems programming languages is critically important to the success of the language. Subjectively, it seems likely that re-expressing algorithms in monadic form will add one layer of complexity too much — much as the combination of aggressive inlining, constructors, and virtual functions did in C++. As the developer loses their ability to build a clear and direct mental model for what is happening on the machine, the quality of systems code deteriorates rapidly. This is a usability and design problem rather than a science problem, and I remain interested to see what will emerge in the House effort.

I should acknowledge that BitC may suffer from an application driven bias in our assessment of state. The Coyotos kernel operates using semi-static memory allocation (a startup-time allocation of available memory is made, but is never changed), constant recursion bounds, and an atomicity discipline. Collectively, these have the effect of simplifying the state model to the point where it may be possible to reason about our system using state induction based reasoning. This may not generalize to user-mode components in Coyotos, many of which operate under less restrictive idioms.

5 Opportunities

The zero-copy idea in operating systems is a critical idea, but so is the concept of protection domains. In hardware-based protection, the basic mechanism for defining domains of protection is the address space. There are two problems with this: (1) hardware context switch times are proportionally slow and getting slower, and (2) the two most widely used embedded processors (Coldfire and ARM) have virtually indexed caches that are not anti-aliased. The first imposes a terrible penalty on secure system structure, and the second is all but fatal.

Because of the need to flush the data cache, the time to perform a full context switch on a 200Mhz StrongARM or XScale processor can be measured in *milliseconds*. For comparison, the corresponding context switch on a 90Mhz Pentium was between 1.35 and 2.4 μ s, and even at that speed was very slow in proportion to the surrounding machine. Despite the work on fast context switching that the L4 and EROS communities did in the 1990s and following, context switch performance remains the limiting factor in the design of well-structured, recoverable systems. There is a certain irony in this, since a primary reason for introducing memory management hardware in early machines was the desire to provide an efficient domain separation mechanism.

Recent results from the Singularity project [10, 1] are extremely thought provoking. If linear types prove to be a human-manageable mechanism for handling interprocess communication, it may really be true that the era of

hardware-based isolation is over (but for legacy support). Most of the performance unpredictability issues of garbage collection become irrelevant if collection occurs over a small heap. By increasing messaging performance, Singularity permits applications to be split into much smaller, independently collected domains. Robustness is simultaneously improved.

In fact, the Singularity project is probably the most serious challenge to the current Coyotos work in the secure embedded system space. There are some denial of resource issues that are inherent in the Singularity communications layer. These are fixable, and we gave serious thought to replacing the Coyotos system with a structurally similar system built on Singularity-like foundations. In the commercial Coyotos effort, we reluctantly concluded that the schedule risk to our current commitments was too high, but we think that there is an interesting research challenge here: enhance the linear type approach of Singularity messages with complete storage accountability during message exchange. See if a system can be successfully structured using language based mechanisms when the assumption that there is an infinite pool of storage is abandoned.

6 The Funding Challenge

Without wandering too far from a mostly technical discussion, it seems important to say a word about *where* and *how* the integration of language and OS research seems likely to happen.

Regrettably, DARPA has seen fit to abandon research funding for computer science in the United States. This is hardly news to anyone who reads the New York Times. While the NSF funding model is good at many things, it is not structured to support large scale, single investigator projects that exceed three years. Europe seems to be putting research funding into OS research, and that is promising. Because of this, it seems likely that leadership in both areas will shift to Europe over the next few years.

Another path to consider is *commercial* funding. While research funding has disappeared in the United States, there is active funding through various SBIR initiatives. Investigators may want to consider more active collaborations with small companies as a path to funding and deploying this type of work. One thing is clear: the days of unlimited funding for core computer science in the United States are over, and we must consider new and creative collaboration strategies if core computer science is to remain viable and relevant.

7 Conclusion

Programming language advances have a lot to offer the operating systems community, but only if they can be embodied in a language that preserves state, (idiomatic) manual storage management, and low-level representation control. BitC is one attempt at this, but we would frankly prefer to work on operating systems and let programming language experts deal with these issues.

Systems programmers *will* adopt a new language if it gives them greater ability to understand and maintain the very complex programs that we write. So far, modern programming languages have come at the cost of a prohibitive level of abstraction. Perhaps the programming language community will revisit these issues.

References

- [1] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. R. Lauris. “Deconstructing Process Isolation.” *Microsoft Technical Report MSR-TR-2006-43*. Microsoft, Inc. 2006
- [2] Thomas Ball and Sriram K. Rajamani. “The SLAM Project: Debugging System Software via Static Analysis.” *Proc. 2002 ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, 2002.
- [3] E. Biagioni. “A Structured TCP in Standard ML.” *Proc. SIGCOMM 1994*. pp. 36–45. 1994.
- [4] T. Blackwell. “Speeding up Protocols for Small Messages.” *Proc. ACM SIGCOMM '96*. pp. 85–95. Sep. 1996.
- [5] J. Bonwick and J. Adams. “Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources.” *Proc. 2001 USENIX Annual Technical Conference*, pp. 15–33. Boston, MA. 2001
- [6] E. Brewer, J. Condit, B. McCloskey, and F. Zhou. “Thirty Years is Long Enough: Getting Beyond C.” *Proc. Tenth Workshop on Hot Topics in Operating System (HotOS X)*, USENIX, 2005.
- [7] H. Chen and J. Shapiro, “Using Build-Integrated Static Checking to Preserve Correctness Invariants.” *Proc. 11th ACM Conference on Computer and Communications Security*. pp. 288–297. Washington, DC. 2004
- [8] H. Derby. *The Performance of FoxNet 2.0*. CMU Technical Report CMU-CS-99-137. 1999.
- [9] Iavor S. Diatchki, Mark P. Jones, and Rebekah Leslie. “High-level Views on Low-level Representations.” *Proc. 10th ACM Conference on Functional Programming* pp. 168–179. September 2005.
- [10] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Lauris, and S. Levi. “Language Support for Fast and Reliable Message-based Communication in Singularity OS.” *Proc. EUROSYS 2006*, Leuven Belgium. 2006
- [11] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. “A Principled Approach to Operating System Construction in Haskell.” *Proc. International Conference on Functional Programming (ICFP'05)*, Sep. 2005. Tallinn, Estonia. pp. 116–128.
- [12] ISO, *International Standard ISO/IEC 8652:1995 (Information Technology — Programming Languages — Ada)* International Standards Organization (ISO). 1995.
- [13] ISO, *International Standard ISO/IEC 9899:1999 (Programming Languages - C)* International Standards Organization (ISO). 1999.
- [14] M. Kaufmann, J. S. Moore. *Computer Aided Reasoning: An Approach*, Kluwer Academic Publishers, 2000.
- [15] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988
- [16] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised* The MIT Press, May 1997.
- [17] Simon Peyton Jones (ed.). *Haskell 98 Language and Libraries: The Revised report*. Cambridge University Press. 2003.
- [18] J. S. Shapiro, J. M. Smith, and D. J. Farber. “EROS, A Fast Capability System” *Proc. 17th ACM Symposium on Operating Systems Principles*. Dec 1999. pp. 170–185. Kiawah Island Resort, SC, USA.
- [19] J. S. Shapiro and N. Hardy. “EROS: A Principle-Driven Operating System from the Ground Up.” *IEEE Software*, **19**(1). Jan. 2002.
- [20] J. S. Shapiro, Eric Northup, M. Scott Doerrie, and Swaroop Sridhar. *Coyotos Microkernel Specification*, 2006, available online at www.coyotos.org.
- [21] J. S. Shapiro. *A C++ Toolkit*, Prentice Hall, 1999.
- [22] **NOT USED** J. S. Shapiro, S. Sridhar, and M. S. Doerrie. *BitC Language Specification, version 0.3+* <http://coyotos.org/docs/bitc/spec.html>
- [23] S. Sridhar and J. Shapiro. “Type Inference for Unboxed Types and First Class Mutability” *Proc. 3rd ECOOP Workshop on Programming Languages and Operating Systems (PLOS 2006)* San Jose, CA. 2006.