HEY… You got your *Paradigm* in my *Operating System*!

Chris Matthews, Owen Stampflee, Yvonne Coady
University of Victoria

Jonathan Appavoo
IBM Research

Marc E. Fiuczynski
Princeton University

Robert Grimm
New York University

ABSTRACT

> "*People like to live in denial; thinking that programming shouldn't be *this* hard right? There must be an easier way, if only those pesky developers followed $fashionable_methodology_of_the_day...*"
>
> Pantelis Antoniou[1]

Linus Torvalds has gone on the record stating is dislike of C++ kernel code many times, "*In fact, in Linux we did try C++ once already, back in 1992. It sucks. Trust me – writing kernel code in C++ is a BLOODY STUPID IDEA...*"[8]. But today's mainstream operating systems are failing to scale. They are not only unable to unleash the power of true concurrency in multi-processor systems, but they are equally incapable of effectively supporting variability in today's feature-rich kernels. We believe the inability of these systems to respond to hardware/software evolution is a direct result of the lack of linguistic support for higher-level paradigms. This position paper provides examples of concrete ways in which traditional approaches are no longer sustainable, and suggests an incremental solution that stands to fit within current practices in systems development. Through incremental adoption, we believe this approach will achieve buy-in from the systems community, otherwise skeptical of the *$fashionable_methodology_of_the_day*.

1. INTRODUCTION

Given the complexities of systems code in general, it is natural to ask if modern programming methodologies and languages can help address some of the challenges facing system developers. Though a number of research systems have attempted to implement an operating system in languages other than C, despite their demonstrated software engineering advantages, most mainstream operating systems have rejected modern linguistic support in their construction. The conventional wisdom in the systems community seems to be that, though object/aspect orientation and domain specific languages may be beneficial, paradigms and runtimes are often cumbersome and inefficient – rendering them inappropriate for operating system construction. As a consequence, though a number of systems have been progressively restructuring services to leverage higher-level paradigms, it is intentionally done without language support.

On the surface, this resistance can be attributed to the practical challenges of trying to incrementally mix modules of differing languages. But we believe that the crux of the problem goes beyond that. Intentional decoupling of paradigms and language mechanisms appear to suggest a fear that opening the door to linguistic mechanisms will impose unnecessarily heavyweight manifestations of paradigms, potentially exposing the system to severe hidden run-time penalties. Given that the majority of any mainstream operating system will retain its current incarnation in hand-optimized C code, we advocate incremental adoption of critical paradigms reunited with linguistic support, albeit with modern, customizable runtimes.

It is important to note that, though current linguistic support has not gained favour in the systems community, adherence to higher-level paradigms is evident in key services, such as file systems and networking. Given that the systems community relies heavily on elegant C and Assembler used in crafty ways to ensure that the code is 100% optimized, the question becomes:

> *Should system developers buy-in to language mechanisms when they can selectively apply the paradigms for free?*

We believe that some of the newest challenges in kernel development – such as scalability in both hardware (multiprocessor systems) and software (variability in feature-rich kernels) motivate the need for incremental adoption of critical paradigms coupled with linguistic support. The costs of maintaining handcrafted code that only loosely leverages high-level paradigms will finally outweigh the skepticism of adopting new mechanisms if we can adequately address performance/adoption concerns in a community that collectively feels that they have been there, done that, and moved on.

1.1 A Blast from the Past: Bell Bottoms and Fat Pointers?

A direct descendent of C, Alef [9], is a block structured concurrent programming language designed for network application development under Plan 9 from Bell Labs (Lucent). It supports abstract data types, parameterized (generic) ADTs and functions, error handling, parallel task synchronization, an explicit tail recursion statement, and *fat* (typed) pointers. Arguably, Alef brought together the best of many linguistic features to support systems programming, though it never appears to have thrived outside of Bell's walls. Since very little can be found on the runtime characteristics of Alef, perhaps the lesson learned from the past is more about the politics of programming languages, as opposed to an indication of a design disaster. If the systems community were to start to look upon language support in a more favourable light, who knows – maybe the constructs of Alef could finally become fashionable and have their day in the sun?

[1] IRC chat, in reference to aspect-oriented programming, Apr/05

2. OSes FAIL TO SCALE

Recent trends in processor hardware show us that processor speed is no longer increasing, but more processing cores are being added. Intel's Hyper-Threading is already commonplace, and both Intel and AMD seem to be pushing to make true dual core processors commonplace[1, 2]. It does indeed seem, two cores are better than one. More processors are becoming more common; however, mainstream operating systems are failing to scale in terms of both hardware and software support. Hardware advancements are hidden behind an infrastructure that makes it difficult to unleash the power of true concurrency in multi-processor systems. Section 2.1 details the nature of this problem, and how an object-oriented approach could help to alleviate the inherent complexities of multi-processor environments. Software advancements are equally disadvantaged – variants force the system to lurch awkwardly into a weakened future state, instead of smoothly evolving into an enlightened state. Section 2.2 overviews the problems with variant evolution, and how an aspect-oriented approach could aid developers in an unobtrusive way.

2.1 True Concurrency

The primary mechanism used to deal with the challenges of synchronization and safety (integrity and consistency) in multiprogrammed systems is hardware support for disabling interrupts. Disabling interrupts makes the execution of a code path atomic, since interrupts are the only events that can cause current execution to be preempted.

With the introduction of multi-processors, disabling interrupts alone is not sufficient to ensure safety. The key feature of a multi-processor is the presence of multiple CPUs, concurrently executing instructions. Rather than simply interleaving instructions in response to interrupts, multiple applications and system requests can be executing in a ``truly concurrent'' fashion.

Shared Memory Multi-Processor (SMP) machines offer a programming model which is a natural extension of a typical uniprocessor; a single shared address space. This has resulted in most general purpose multi-processors today being SMPs. Given the familiar programming model, it was natural to develop operating system for SMPs as an incremental extension to uniprocessor OSes. Although this approach was the natural course of development, research has shown it is not necessarily the best approach with respect to yielding high performance SMP OSes.

The first and foremost challenge in porting a uniprocessor OS to an SMP is to ensure mutual exclusion. The most common approach has been to introduce a synchronization primitive into the uniprocessor code paths, serializing execution while requiring minimal changes. Initially, most solutions focused on correctness, that is to say that the techniques proposed ensured that only one process could be executing the critical section at a given time. Solutions then progressed to take into account notions of forward progress, attempting to provide guarantees about how the processes attempting to execute a critical section would progress. Later solutions also attempted to account for the performance of the primitives, ensure efficient execution on typical hardware platforms.

Most modern operating systems have settled on the semantics of a *lock* for synchronization. The implementation of locks to achieve mutual exclusion on general purpose SMP's typically synchronizes processors via shared variables. A great deal of effort has been spent in studying the performance of SMP locking techniques. Some of the aspects that were studied include: the effects of busy waiting and blocking when a lock is contended; the effects of how waiters on a lock are notified of released; the length of the critical sections associated with a lock; the use of multiple locks and the associated potential for deadlocks; the interaction of locks and scheduling; and the impact of hardware characteristics. There are increased costs associated with read write sharing of variables due to the attendant rise in communications.

Despite decades of research and development into SMP operating systems, achieving good scalability for general purpose workloads across a wide range of processors has remained elusive. ``Sharing'' lies at the heart of the problem. By its very nature, sharing introduces barriers to scalability.

2.1.1 An Abstraction for Sharing: Clustered Objects

Dealing with sharing is not a trivial task. The fine-grain locking used in traditional systems results in complex and subtle locking protocols. Also in traditional systems, adding per-processor data structures leads to obscure code paths that index per-processor data structures in ad-hoc manners. *Clustered Objects* were developed as a model of *partitioned objects* to simplify the task of designing high-performance SMP systems software [3]. In the partitioned object model, an externally visible object is internally composed of a set of distributed *Representative* objects. Each Representative object locally services requests, possibly collaborating with one or more other Representatives of the same Clustered Object. Cooperatively, all the Representatives of the Clustered Object implement the complete functionality of the Clustered Object. To the clients of the Clustered Object, the Clustered Object appears and behaves like a traditional object.

Techniques for constructing SMP friendly OSes have been around for nearly two decades [?]. Both Microsoft and Linux system developers have gone out of their way to create scheduling data structures that avoid cache conflicts[?]. The distributed nature of Clustered Objects make them ideally suited for the design of multi-processor system software, which often requires a high degree of modularity and yet benefits from the sharing, replicating and partitioning of data on a per-resource (object) basis. Clustered Objects are conceptually similar to design patterns such as façade [4]; however, they have been carefully constructed to avoid any shared front end, and are primarily used for achieving data distribution. IBM Research's K42 operating system[5] has taken advantage of these benefits with its Clustered Objects kernel facility to provide scaleable kernel data structures.

2.1.2 Asking the Right Questions

We offer some initial thoughts about a project to increase the scalability of the Linux operating system by introducing a Clustered Object facility. This project is a systems project; however, it has been acknowledged that taking to heart a more holistic programming philosophy could improve the quality of the resulting system if performance is not compromised [3].

Clustered Objects require information from the underlying operating system to function efficiently [6]. Whether the Clustered Object runtime system best belongs in the Linux kernel is still an open question; but there have been clustered object

systems imbedded in kernels before. IBM Research's K42 operating system [5] is one example of this. K42's Clustered Object facility is the template from which our new Clustered Object facility will be derived.

The K42 Clustered Object facility from which this project's was derived already had some interesting traits. Unlike most systems software it was written in C++, the facility makes use of a simple inheritance hierarchy to represent the variants of the core Clustered Object entities (representatives and miss handlers) [6]. There are many different types of representatives (policies) to deal with different methods of sharing and different hardware topologies [4]. The K42 creators were able to leverage C++ inheritance to make these policies consistent and the code much more understandable and well organized.

There are however some aspects of the system that do not nicely fit into the hierarchy. One example of this is memory allocation. There are some system rules that may be considered cross cutting concerns. For example, some representatives are not allowed to have their constructors called, but rather a custom create( ) function is called to create a new instance of a Clustered Object representative [7]. This is for memory allocation reasons and is currently enforced with macros. This style of crosscutting concern might be better implemented as an AspectC++ [10] aspect that deals with custom memory allocation than scattered through out the different representative's code.

2.1.3 A Case for Customization?

Some areas of the Clustered Object facility have not had the same trickle in of object-oriented ideas. For instance, the Clustered Object translation and dispatch routines that are responsible for performing runtime resolutions of Clustered Objects. These routines use assembly code and vtable manipulation to operate in an efficient manner outside the rules of regular C++. In a more "safe" programming environment this might not even be possible.

The use of C++ inheritance did help with the structure of the system; but the fact remains there was very little that could not have been done with regular C and some language parlor tricks. It may only have been necessary for a small C++ front end, and have the rest of the code in C. This would have the obvious performance and control benefits; as well, it would ease the interface with the Linux kernel that is also written in C. The performance gains may be tangible, but the losses in other areas may not be. The C++ version has an inherent structure; the C version may lack some of this structure. Macros may be applied where aspects might have; however, this process is done by hand and error prone. All these will lead to less understandability and more bugs in the system.

The questions become: was the extra structure worth it? Will another paradigm help or even be able to do the things required?

2.2 Managing Kernel Variants

Developers of kernel variants often start with a mainline kernel from kernel.org and then apply patches for their application domain. Many of these patches represent crosscutting concerns in that they do not fit within a single program module and are scattered throughout the kernel sources—easily affecting over a hundred files. It requires nontrivial effort to maintain such a crosscutting patch, even across minor kernel upgrades due to the variability of the kernel proper. It is exceedingly difficult to ensure correctness when integrating multiple variants. To make matters worse, developers use simple code merging tools that are limited to textual substitution (e.g., diff and patch), with the result that patch maintenance is error prone and time consuming.

2.2.1 An Abstraction for Crosscutting: Aspects

Our position is that a better method is needed—beyond diff and patch—that reduces the amount of work it takes to maintain and review a crosscutting kernel extension in Linux. We propose a semantic patch system called c4 for CrossCutting C Code [11]. A semantic patch basically amounts to a set of transformation rules that precisely specify the conditions under which changes need to be made and the means for rewriting the affected code. Its compact yet human readable form lets a community of developers easily understand and discuss a crosscutting kernel extension, thereby helping reduce the time and effort required to evolve the kernel.

2.2.2 Asking the Right Questions

Aspects in the kernel are not a new idea, but what will it take to get the system community to buy-in to this controversial paradigm? Much like object-orientation, runtime costs must be known in advance. Similarly, the ability to customize the linguistic support, perhaps only providing a limited set of features relative to application level programming, is a must.

2.2.3 A Case for a Customization?

We aim to reduce developers' exposure to c4 as much as possible. In particular, we are exploring how to support simple annotations of the form aspect(Name){...}, which can be added inline at the beginning or end of system functions and are then automatically extracted and converted into fully-featured aspects by the c4 compiler.

3. WHY BUY THE COW?

Given these concrete ways in which traditional approaches are no longer sustainable, we now advocate an incremental solution that stands to fit within current practices – and perhaps political climate–-of systems development. Through incremental adoption, we believe this approach will achieve buy-in from the systems community, otherwise skeptical of the *$fashionable_methodology_of_the_day*.

Primarily, we need to lay down our swords and think in nonpartisan terms about problems that are not met by current mechanisms, and what the important abstractions in the kernel really are.

We believe there is compelling evidence to suggest that if clustered objects and crosscutting concerns were first class citizens in kernel code, complete with first class linguistic support, kernels would be better equipped to cope with both hardware and software evolution.

Given that clustered objects have already been implemented in C++, this begs the question as to whether a module of this nature is viable and of interest to the community. That is, could the current implementation of clustered objects be extracted from K42 and transplanted into a mainstream OS, and if so, what are the barriers to its acceptance?

We believe that a first step in this direction is to get a more accurate accounting of maintenance and runtime costs. If they are prohibitive, then a lighter-weight runtime can be customized using a tool such as `xtc` [12]. Such an approach could most

flexibly support objects, aspects, and other necessary constructs. With kernels being restructured according to object-based paradigms, grafting this implementation into a mainstream operating system may be as easy as leveraging a polymorphic dispatch along the page fault path. Given runtime costs can be shown to be acceptable, then we return to the original question:

> *Should system developers buy-in to language mechanisms when they can selectively apply the paradigms for free?*

In an environment where *selectively* can mean inconsistently, incompletely, and inefficiently, and *free* can mean minimal microbenchmark performance penalty but maximal system compromise (for example, in terms of true concurrency) and maintenance fees, we believe the evidence we have gathered to date shows the answer is unequivocally, YES!

Linguistic support, such as that provided by aspect-oriented programming, has shown to present an important foundation upon which more powerful programming tools can be bolted-on. The fact that tools can leverage information explicitly declared in pointcuts and help developers visualize both internal invariants and external interaction comprehensively. IVY [13] is one such example that suggests a new, better way of support metaprogramming (macros), but they provide a tool (macroscope) to support the translation of existing cpp macros to IVY macros.

4. CONCLUSIONS

If the gap between the Programming Language and Systems communities can be bridged, kernel code may one day scale to accommodate evolution in both hardware and software. To date, the systems community focuses on using solutions that address a particular point of pain fully, while the former often focuses on a disjoint problem set. Arguably, there is currently too much work and trust required by the systems community to adopt solutions from the languages community. To bridge the gap, it is necessary for the languages community to build a set of tools that not only introduce a new paradigm, but also help in refactoring existing code from its current state to fit such new paradigms.

REFERENCES

[1] Intel, "Dual-Core Server Processors," Intel Corporation, http://www.intel.com/business/bss/products/server/dual-core.htm. 2005.

[2] AMD, "Welcome to AMD Multi-Core Technology," Advanced Micro Devices, Inc, http://multicore.amd.com/Global/. 2005.

[3] G. Yilmaz and N. Erdogan, "Partitioned Object Models for Distributed Abstractions," presented at 14th International Symp. on Computer and Information Sciences (ISCIS XIV), Kusadasi, Turkey, 1999.

[4] E. Gamma, R. Helm, and R. Johnson, *Design Patterns. Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.

[5] J. Appavoo, M. Auslander, D. DaSilva, D. Edelsohn, O. Krieger, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis, "K42 Overview," IBM TJ Watson Research 2002.

[6] J. Appavoo, "Clustered Objects: Initial Design, Implementation and Evaluation," in *Computer Science*. Toronto: University of Toronto, 1998, pp. 1-103.

[7] J. Appavoo, "phone conversations," C. Matthews, Ed. Victoria, 2005.

[8] Kerneltrap.org, http://kerneltrap.org/node/2067?PHPSESSID=3cf6f33dc7f40cedfe6014fc385fe239

[9] Alef, http://lists.cse.psu.edu/archives/9fans/1995-October/004370.html

[10] AspectC++, http://www.aspectc.org/

[11] Marc Fiuczynski,; Robert Grimm, Yvonne Coady, David Walker, "patch (1) Considered Harmful", HotOS X, 2005.

[12] xtc, Extensible C, http://www.cs.nyu.edu/rgrimm/xtc/

[13] Eric Brewer, Jeremy Condit, Bill McCloskey, Feng Zhou, "Thirty Years is Long Enough: Getting Beyond C", HotOS X, 2005.