Managing Variability in Systems: Oh What a Tangled OS We Weave

Stuart Bray, Marco Yuen ,Yvonne Coady
University of Victoria
{sbray, marcoy}@csc.uvic.ca, ycoady@cs.uvic.ca

Marc E. Fiuczynski
Princeton University
mef@cs.princeton.edu

This position paper outlines challenges associated with managing OS variants, key ways in which AOP could help, and criteria for adoption that we believe AOP must meet.

1. INTRODUCTION

Major variants to a mainline Linux kernel are typically represented in terms of high level frameworks and low level patches. Frameworks are typically vetted several times in the operating systems community before becoming mainlined into the kernel. Patches that quietly sprinkle the necessary hooks for the framework throughout the existing system receive less attention. Developers trying to manage variants manifested as patches are given no support when trying to understand the code.

Frameworks that have recently been mainlined include Linux Security Modules (LSM) [10], Advanced Linux Sound Architecture (ALSA) [1], and Video4Linux [13]. There are many others possibly soon to be mainlined, such as Class-Based Kernel Resource Management (CKRM) [5] and Linux Trace Toolkit (LTT) [11]. These are just a few of the many frameworks that resort to rudimentary patching schemes.

Though aspect-oriented programming (AOP) [9, 4] is poised to help, criteria for its adoption must ensure smooth integration with current approaches and tools within a systems context. To this end, we examine a concrete example, describe how AOP could help, establish criteria of adoption, and finally describe an experiment we plan to explore.

2. EXAMPLE: CKRM

Class-based Kernel Resource Management (CKRM) [5] project is developing new kernel mechanisms to provide differentiated service to shared system resources. These resources include CPU, memory, I/O and network bandwidth. At a low level, changes associated with this variant have been organized into several patches, as outlined in Table 1. The table shows that some patches are self-contained, involving no changes to the existing code, whereas others change collections of files.

In more detail, these patches dictate where "hunks" of code are applied, at specific line numbers or relative offsets within specific files, throughout the system. For example, this portion of a CKRM patch (*00-core.ckrm-E12.patch*):

```
@@ -619,6 +623,9 @@
   }
   else
                return -EPERM;
+
+  ckrm_cb_gid();
+
   return 0;
}
```

instruments an existing file with a call to the *ckrm_cb_gib()* function. Line numbers are represented as relative offsets indicated by @@<*line-info*>@@. Precisely which function is instrumented is not clear looking at the patch. But upon closer inspection some patterns are clear – all calls to *ckrm_cb_gid()*/*uid()* (6 in total) directly precede *return* statements, and *ckrm_cb_exit()/newtask()* are at the beginning and end of functions in exit/fork.c.

Leveraging these clues to trace upwards to higher level design intent is impaired in this representation. Developers are faced with minimal support for reasoning about interaction between the variant and the mainline kernel, prolonging the community consensus phase (roughly 2 years for LSM) for incorporating otherwise simple hooks. Moreover, as the mainline kernel evolves, the points during the execution of the kernel when the hooks should fire may also evolve. The problem is compounded as more variants are introduced, further bogging down the overall process of mainline decisions.

Table 1: CKRM patches, new files and files changed.

Kernel Patch	New Files	Files Changed
00-core.ckrm-E12.patch	*kernel/ckrm/Makefile* *kernel/ckrm/ckrm.c* *kernel/ckrm/ckrmutils.c* *include/linux/ckrm.h* *include/linux/ckrm_ce.h* *include/linux/ckrm_rc.h*	*include/linux/sched.h* *init/Kconfig* *init/main.c* *kernel/Makefile* *kernel/exit.c* *kernel/fork.c* *kernel/sys.c* *fs/exec.c*
01-rcfs.ckrm-E12.patch	*fs/rcfs/Makefile* *include/linux/rcfs.h* *fs/rcfs/dir.c* *fs/rcfs/inode.c* *fs/rcfs/magic.c* *fs/rcfs/rootdir.c* *fs/rcfs/super.c*	*fs/Makefile*
02-taskclass.ckrm-E12.patch.	*include/linux/ckrm_tc.h* *kernel/ckrm/ckrm_tc.c* *fs/rcfs/tc_magic.c*	none
03-numtasks.ckrm-E12.patch	*include/linux/ckrm_tsk.h* *kernel/ckrm/ckrm_tasks.c*	none
04-socketclass.ckrm-E12.patch	*include/linux/ckrm_net.h* *kernel/ckrm/ckrm_sockc.c* *fs/rcfs/socket_fs.c*	none
05-socketaq.ckrm-E12.patch	*kernel/ckrm/ckrm_listenaq.c*	*include/linux/tcp.h* *include/net/sock.h* *include/net/tcp.h* *net/ipv4/Kconfig* *net/ipv4/tcp.c* *net/ipv4/tcp_ipv4.c* *net/ipv4/ tcp_minisocks.c* *net/ipv4/tcp_timer.c*
cpu.ckrm-E13.patch	*include/linux/ckrm_sched.h* *include/linux/ckrm_cpu_class.h*	*include/linux/ circularqueue.h* *include/linux/sched.h* *kernel/Makefile* *kernel/circularqueue.c* *kernel/sched.c* *init/Kconfig*

AOP promises an elegant, language level solution to this problem. Instead of instrumenting mainline code with patches and line offsets, an aspect would use a few simple language extensions to attach hooks to principled points in the execution of the kernel. For example, the following CKRM aspect would instrument the kernel with: (1) a call to *ckrm_cb_newtask()* **before** the **execution** of *f1()*, exposing the parameter *arg* to CKRM code, and (2) calls to *ckrm_cb_uid()* **after** functions *f2(), f3()* or *f4()* respectively.

```
aspect CKRM { // modularizes CKRM hooks

   before(task_t* arg): execution(void f1(arg))
   {
      ckrm_cb_newtask(arg);  //accesses parameter
   }


  after(): execution(void f2() || void f3() || void f4())
   {
      ckrm_cb_uid();
   }
   ...
}
```

The aspect thus structures the modifications to the mainline code, which are "woven" automatically at buildtime (or, in recent AOP approaches, at runtime). The interaction with the kernel becomes explicit at the level of functions and parameters involved; hence,

code becomes more amenable to semantic analysis and developers can reason about the interaction at a higher level. Related work has shown the ways in which aspects reduce complexity of crosscutting concerns [7, 6, 12].

3. CRITERIA FOR ADOPTION

For developers to buy-in to AOP as a viable solution to this problem of managing variability in systems code, we must definitively answer three critical questions:

(1) how will it better support reasoning about interactions?
(2) how will it impact mainline structure and evolution?
(3) how will it impact runtime and buildtime performance?

Clarity of Interactions: Since AOP would make interaction between variants and a mainline kernel explicit at the level of the interfaces and parameters involved, the ability to reason about these interactions would be improved. The assumption here is that the aspect-oriented implementation of the variants could better represent the high level intent behind the hooks. That is, the principled ways in which variants interact with the mainline code would become clear. Tool support would be an added benefit, but ultimately, the ability to view the woven code may arguably be more natural within the minimal development environments typically available to system programmers.

Mainline Structure and Evolution: In order to compose aspects with mainline kernel code, existing mainline functions may have to be refactored. Ensuring that this refactoring is reasonable and causes no harm (performance or otherwise) is a necessary preliminary step before AOP can be considered a viable solution in a systems context.

Runtime and Buildtime Performance: Compiler optimizations, such as inlining, are an important part of code generation when AOP is applied to systems code. Though an in-depth study would be required before results are conclusive, we believe static AOP mechanisms will be acceptable if inlined throughout the kernel, whereas dynamic mechanisms (such as *cflow*) may have to be avoided in certain contexts due to performance constraints. Additionally, tighter integration with the compiler, including compiler warnings associated with changes to mainline code that could impact aspects, may ultimately enhance/replace tool support in this context.

4. AN EXPERIMENT

We are designing an experiment to test AOP's ability to meet this criteria; specifically, to manage variants in Linux kernel design and implementation. In the design portion of the experiment, we will look at points of interaction for several kernel patches. The goal is to trace upwards to some high level design criteria which would explain the significance of their points of interaction.

Table 2: Linux patches of various sizes.

Patch	New Files	Modified	#Line Adds(+)	#Line Subs(-)
ALSA	200	540	324672	148419
LSM	123	85	40355	415
LTT	9	71	5546	17
LLA[1]	1	39	619	60

Preliminary results in Table 2 show the following coarse assessment of four different patches of various sizes. Notice that there are two types of additions: new files added, and modifications to existing files. We also show the number of line additions and removals for each variant.

Further analysis of each patch will expose the points of interaction; that is, where the line additions and removals are actually taking place. If it is determined that these hooks lie at the boundaries of a function, they can be woven into the mainline code with an aspect. In the cases where the hooks are in the middle of functions, we will have to determine exactly why they are there. That is, we have to further establish if these hooks: (1) can be moved or (2) are preceded simply by local data structure initialization and argument validity checks or (3) may actually provide hints as to where otherwise bloated kernel functions should be refactored, because they perform more than one functional task. Ultimately, we expect that a clear separation of concerns within the kernel will improve its long-term scalability and ease of maintenance.

In the implementation portion of the experiment, we will assess the available AOP prototypes, such as AspectC/C++ [2, 3]. Evaluation will be based on the ability to support AOP within a systems context. The runtime and buildtime performance of each prototype will be measured; however, we do not believe these metrics to be of paramount importance at this stage of adoption, because inadequate performance in these areas can be improved with code optimizations and re-engineering. If the available prototypes are not sufficient, we may either work to improve their utility or consider redevelopment with eXTensible C (xtc) [8].

5. DISCUSSION

Variability in open source projects introduces several software engineering challenges – variants might break other devices in the system or not follow existing conventions, and this code is hard to analyze effectively once it is hidden after merges. A structural separation of variants, engineered with aspects, would save kernel programmers from having to compose and scatter more code across the kernel to resolve such issues. Though assessment of Aspect-Oriented Software Development (AOSD) [4] in general is still arguably in its early days, we believe AOP is particularly well suited to deliver the structural support necessary to manage variability in large open source software projects such as Linux. Improved management will accelerate advancement by clarifying variant merges that are typically complex and time consuming. Using AOP, the functionality and structure of a variant can be modularized, and its principled interaction with the kernel exposed.

6. References

[1] Advanced Linux Sound Architecture, http://www.alsa-project.org/
[2] AspectC, http://www.cs.ubc.ca/labs/spl/projects/aspectc.html
[3] AspectC++, www. aspectC.org
[4] Aspect-Oriented Software Development, www.aosd.net.
[5] Class-based Kernel Resource Management, http://ckrm.sourceforge.net/
[6] Yvonne Coady, Gregor Kiczales, Mike Feeley and Greg Smolyn, Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code**.** FSE-9, 2001.
[7] J. Gray, T. Bapty and S. Neema, "Aspectifying Constraints in Model Integrated Computing". In Proceedings of OOPSLA 1999.
[8] Robert Grimm, eXTensible C, (xtc) http://www.cs.nyu.edu/rgrimm/xtc/
[9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin, *Aspect-Oriented Programming*, ECOOP, 1997.
[10] Linux Security Modules (LSM), http://lsm.immunix.org/
[11] Linux Trace Toolkit (LTT), http://www.opersys.com/LTT/
[12] Gail Murphy, Robert Walker, and Elisa Baniassad. Evaluating Emerging Software Development Technologies: Lessons Learned from Evaluating Aspect-oriented Programming. In IEEE Transactions on Software Engineering 25, 4, 1999.
[13] Video4Linux, http://linux.bytesex.org/v4l2/

[1] Low Latency Audio, http://www.linuxdj.com/audio/lad/resourceslatency.php3