Brittle Systems will Break – Not Bend: Can Aspect-Oriented Programming Help?

Yvonne Coady, Gregor Kiczales, Joon Suan Ong, Andrew Warfield and Michael Feeley
University of British Columbia

Abstract

As OS code moves to new settings, it must be continually reshaped. Kernel code however, is notoriously brittle – a small, seemingly localized change can break disparate parts of the system simultaneously. The problem is that the implementation of some system concerns are not modular because they naturally crosscut the system structure.

Aspect-oriented programming proposes new mechanisms to enable the modular implementation of crosscutting concerns. This paper evaluates aspect-oriented programming in the context of two crosscutting concerns in a FreeBSD 4.4 kernel – page daemon activation and disk quotas. The ways in which aspects allowed us to make these implementations modular, the impact they have on comprehensibility and configurability, and the costs associated with supporting a prototype of an aspect-oriented runtime environment are presented.

1. Introduction

Simple system rules such as avoiding deadlock by ensuring functions that block are not called while interrupts are disabled, are difficult to verify when inspecting kernel source code [5]. Not surprisingly, the coordination of more complex concerns, such as paging and quotas, are even more challenging to reason about and dependably manipulate. Successfully modifying the implementation of such elements requires both manual inspection of the multiple places in the system where they have impact, and derivation of otherwise implicit semantic information to understand the structural relationships involved. A key part of the problem is that their implementation is not modular.

The goal of aspect-oriented programming (AOP) [11] is to better modularize crosscutting concerns. Ideally, when modularized as aspects, the structure of crosscutting concerns becomes explicit and hence more comprehensible and configurable. AOP thus has the potential to make OS code less brittle and more amenable to change.

This paper provides an early assessment of our application of AOP to kernel code. After a brief introduction to AOP, the semantics of specific linguistic mechanisms are examined within the context of two practical examples, page daemon activation and disk quotas. An analysis of these implementations then highlights the specific ways in which these aspects provide leverage for reasoning about and configuring crosscutting implementation more comprehensively. Next, the implementation of AspectC runtime support is described, and microbenchmarks of our prototype are provided. Finally, this work is put in context with other research aimed at improving system structure.

1.1. AOP background

The AOP community has proposed linguistic mechanisms intended to allow implementation of crosscutting concerns as first class modules called *aspects*. To enable a range of experiments for operating systems written in C, we are developing AspectC [3]. Conceptually and in syntax, AspectC is a simple subset of AspectJ [10]. Aspect code, known as *advice*, interacts with other code at function call boundaries and can run `before`, `after` or `around` the call to, or execution of the function. The central elements of the language are a means for designating particular function calls, for accessing parameters of those calls, and for declaring advice on those calls. Capturing dynamic control flow context is done using the `cflow` construct, which enables advice to determine the calling context of a function's execution, and access arguments to functions higher up in the call path.

1.2. AOP mechanisms

The following portion of a example aspect, called `profile_low_and_high_level_params`, provides a brief

syntactic and functional introduction to mechanisms advice use to specify when they run, and what values they access.

```
aspect profile_low_and_high_level_params {

  before(int x, int y):
    execution(void low_level(x))
    && cflow(execution(void high_level(y, char)))
  {
    printf(``low_level runs with %d when
              high_level runs with %d'',
           x, y);
  }
}
```

This aspect defines one advice. Looking at the first line of the advice in detail, this line specifies that the advice executes `before` certain points in the execution of the system, and that it has two integer parameters, `x` and `y`.

The second and third lines – syntactically between the ':' and the '{' – define the execution points to be the intersection of all executions of the `low_level` function, and all functions that execute in the dynamic context of an execution of `high_level`. In other words, executions of `low_level` dynamically within executions of `high_level`. The second line also says that the value for `x` comes from the first argument to `low_level`, and the value for `y` comes from the first argument to `high_level`. The second argument to `high_level` is ignored.

The body of the advice is regular C code. The effect of this advice is print a message and values for `x` and `y` before all executions of `low_level` that occur dynamically within an execution of `high_level`.

2. Examples of Crosscutting Concerns

Examples of crosscutting concerns in kernel code include page daemon activation, disk quotas, prefetching, scheduler activation, checksums, and profiling. This section starts by providing detailed information about the original implementation of page daemon activation in FreeBSD 4.4. This example is then used to show how we have refactored this code as an aspect and structured the implementation in one modular unit. Similarly, the original implementation of disk quotas and a corresponding aspect-oriented implementation of a portion of the original code are shown. The remaining examples are then surveyed, and put in context with related work.

2.1. Page daemon wakeup

In FreeBSD 4.4, the page daemon is responsible for freeing space in the virtual memory system. The page daemon imposes overhead for determining which pages will be replaced and for writing them back to disk if necessary. As a result, timing is an important factor when waking the page daemon – we want to do it only when the number of available pages has fallen below some threshold. The function that wakes the page daemon, `pagedaemon_wakeup`, is invoked from 6 places in the kernel code: 4 calls from VM page operations, 1 call from page fault handling code, and 1 call from buffer allocation. The specific files and functions involved are shown in Table 1.

File [x(frequency)]	*Functions*
vm/vm_page.c [x4]	vm_page_unqueue vm_page_alloc
vm/vm_fault.c [x1]	vm_fault_additional_pages
kern/vfs_bio.c [x1]	allocbuf

Table 1. Scattering of calls to `pagedaemon_wakeup` across virtual memory and buffer cache code.

2.1.1. Page daemon wakeup aspect

We have reimplemented this page daemon wakeup functionality in one aspect. Figure 1 shows the page daemon aspect in its entirety, preceded by a few small helper functions. The code uses named pointcuts to clearly identify specific points in kernel execution when paging may be needed. These four pointcuts, associated with unqueuing pages, allocating pages, faulting pages, and allocating buffers respectively, are shown in the top half of the aspect. Each of the four advice declarations in the bottom half of the aspect uses one of these named pointcuts to say what page daemon wakeup test should happen at each point.

The first pointcut, `unqueuing_available_pages` names points in the execution when `vm_page_unqueue` executes in the control flow of any one of the four functions listed, and makes the `vm_page_t` parameter to whichever of those four functions the execution is within available to advice that uses this pointcut. The first advice executes `around` these points, using the AspectC keyword `proceed` to allow the originally intended function, `vm_page_unqueue`, to execute.

Localizing this implementation in this way allows us to see which global counters are used and when. As highlighted in Figure 1, `cache_min` is not used when faulting, and `free_reserved` is not used when allocating buffers. In the original implementation, establish-

```
/* helper functions */
int pages_available()    { return cnt.v_free_count    + cnt.v_cache_count; }
int vm_page_threshold()  { return cnt.v_free_reserved + cnt.v_cache_min;   }
int vfs_page_threshold() { return cnt.v_free_min      + cnt.v_cache_min;   }


aspect pageout_daemon_wakeup {

  /* when we are unqueuing */
  pointcut unqueuing_available_pages(vm_page_t m):
    execution(void vm_page_unqueue(m))
    && cflow(execution(void vm_page_activate(vm_page_t))
             || execution(void vm_page_wire(vm_page_t))
             || execution(void vm_page_unmanage(vm_page_t))
             || execution(void _vm_page_deactivate(vm_page_t, int)));


  /* when we are allocating new pages */
  pointcut allocating_pages(vm_object_t object, vm_pindex_t pindex, int page_req):
    execution(vm_page_t vm_page_alloc(object, pindex, page_req));


  /* when we are faulting in pages */
  pointcut faulting_pages(int rbehind, int rahead):
    execution(boolean_t vm_pager_has_page(vm_object_t, vm_pindex_t, int*, int*))
    && cflow(execution(int vm_fault_additional_pages(vm_page_t, rbehind, rahead,
                                                     vm_page_t*, int*)));

  /* when we are allocating buffer space */
  pointcut allocating_buffers(vm_page_t m, int also_m_busy, const char* msg):
    execution(int vm_page_sleep_busy(m, also_m_busy, msg))
    && cflow(execution(int allocbuf(struct buf*, int)));


  /* below threshold for VM when unqueuing */
  around(vm_page_t m):
    unqueuing_available_pages(m)
  {
    int queue = m->queue;
    proceed(m);
    if (((queue - m->pc) == PQ_CACHE) && (pages_available() < vm_page_threshold()))
        pagedaemon_wakeup();
  }


  /* page alloc fails, or below threshold for VM when allocing */
  around(vm_object_t object, vm_pindex_t pindex, int page_req):
    allocating_pages(object, pindex, page_req)
  {
    vm_page_t allocd_page = proceed(object, pindex, page_req);
    if (allocd_page == NULL)
      pagedaemon_wakeup();
    else
      if (pages_available() < vm_page_threshold())
        pagedaemon_wakeup();
    return allocd_page;
  }


  /* prefetching past modified threshold for VM */
  after(int rbehind, int rahead):
    faulting_pages(rbehind, rahead)
  {
    if ((rahead + rbehind) > (pages_available() - cnt.v_free_reserved))
      pagedaemon_wakeup();
  }


  /* buffer allocing when below threshold for VFS */
  around(vm_page_t m, int also_m_busy, const char* msg):
    allocating_buffers(m, also_m_busy, msg)
  {
    int had_to_sleep = proceed(m, also_m_busy, msg);
    if (!had_to_sleep  && ((m->queue - m->pc) == PQ_CACHE)
                       && (pages_available() < vfs_page_threshold()))
      pagedaemon_wakeup();
    return had_to_sleep;
  }

}
```

Named pointcuts identify points in the execution of the kernel -- paging maybe be needed at any one of these four pointcuts.

Advice declarations make a relationship between advice code and when it runs.

This advice executes around points when we are unqueuing_available_pages(). It has access to vm_page_t m.

Page unqueuing and allocating both use vm_page_threshold() (shown at top of page).

Page fault handling uses a modified threshold, without cnt.v_cache_min

Allocating buffers uses yet another threshold, vfs_threshold() (shown at top of page)

Figure 1. The page daemon wakeup aspect captures the points in the system where the page daemon may be activated if the system is running low on free pages and the possible activation of the daemon at each point.

ing these threshold conditions requires visiting 3 files.

2.1.2. Impact of Page Daemon Aspect

This example shows how an aspect can structure the implementation of page daemon activation within one modular unit. The entire invocation behaviour of the page daemon is captured in this single page of source code. The impact on the paging code is that page daemon activation is no longer scattered in the functions listed in Table 1.

2.2. Quota

Disk quotas are an optional feature of FreeBSD 4.4, configured through a combination of settings in both a kernel configuration file and on a per-file system basis. Through a collection of 37 `#ifdef QUOTA` preprocessor directives in UFS, FFS and EXT2, and 9 `#if QUOTA` directives in EXT2, calculating and maintaining disk quotas is scattered and tangled within 22 functions from 10 files in these file systems, as shown in Table 2. As indicated in the Table, there is overlap between FFS and EXT2 with respect to quota.

Implementing quota with these 46 preprocessor directives supports efficient, coarse grained configurability – we can turn off quota functionality and know it is not part of the binary. Unless we are working directly with quota or code it affects, we can treat this code separately, as it is not part of the core functionality of the file system. Preprocessor directives, however, make it difficult to reason comprehensively about quota, and understand the structural relationships that hold. They also obscure reading of the code quota is scattered in.

2.2.1. Quota aspect

The aspect-oriented implementation of quota localizes the code in a single module. Because aspects can be unplugged from the system by excluding them in the Makefile, the aspect-oriented implementation maintains the same unpluggability as the original preprocessor based implementation.

Figure 2 shows the code for the VFS portion of the aspect-oriented implementation of quota that uses shared advice for FFS and EXT2 – the same quota advice is attached to corresponding functions from the two file systems. As with the daemon activation aspect, the relevant points in the execution of the program are first identified as named pointcuts. The last of these, `vget`, identifies all calls to `ufs_ihashins` within the `cflow` of the execution of either `ffs_vget` or `ext2_vget`. In this example, the 7 `#ifdefs` in this portion of Table 2 are replaced with 3 advice shown in Figure 2.

2.2.2. Impact of Quota Aspect

Looking at the pointcut declarations, we can see which core file system functions and values are involved, along with their similarities and differences with respect to quota. Looking at the bodies of advice, we see essentially what had been bracketed by preprocessor directives in the original base code. Relative to the preprocessor based implementation, unpluggability has not been compromised. The impact on the rest of the file system code is that the preprocessor directives and associated quota functionality are no longer tangled in the file system functions.

2.3. Advantages of these new perspectives

The intent of our aspect-oriented refactoring is to better separate page daemon activation and disk quotas from the code they crosscut. Aspect-oriented programming naturally involves tool support, similar to that of object-oriented class browsers, that supports easy navigation between aspects and the code they advise. AspectC does not yet support these tools, however, extensions to Emacs, JBuilder, NetBeans and Eclipse are available for AspectJ.

An aspect localizes both the operations of a crosscutting concern, and the declaration of points in the system when the operations happen. The page daemon and quota examples considered here demonstrate how AspectC localizes crosscutting implementation in a way that makes structural information associated with crosscutting explicit. These aspects provide new perspectives that help with page daemon wakeup and file system quota in slightly different ways, respectively.

Knowing when the page daemon may be made runnable makes it easier to reason about activation system-wide. This perspective can be used to more easily ensure a consistent and minimal set of activation points across subsystems.

In particular, seeing the thresholds used to determine daemon activation and the contexts in which they are applied together makes it easier to reason about subtle relationships that exist, such as: (1) page fault handling has the only threshold check that does not use `cache_min`, the minimum number of pages desired on the cache queue, and (2) while VM uses the number of `free_reserved` pages, the number of pages reserved for dealing with deadlock, VFS uses the more conservative value of `free_min`, the minimum number of pages

Component [*x(frequency)*]	*File System* UFS	FFS	EXT2
VNode [x21]	ufs_access[x2] ufs_chown[x3] ufs_mkdir[x4] ufs_makeinode[x4]		ext2_mkdir[x4] ext2_makeinode[x4]
VFS [x9]	ufs_quotactl[x1] ufs_init[x1]	ffs_flushfiles[x1] ffs_sync[x1] ffs_vget[x1]	ext2_flushfiles[x2] ext2_sync[x1] ext2_vget[x1]
INode [x8]	ufs_inactive[x1] ufs_reclaim[x2]	ffs_truncate[x2]	ext2_truncate[x3]
Alloc [x8]		ffs_alloc[x3] ffs_balloc[x1] ffs_realloccg[x1]	ext2_alloc[x3]

overlap

Table 2. Scattering of 46 `#ifdef/#if QUOTA` across UFS, FFS, and EXT2. The overlap shown in the table refers to identical quota code in both FFS and EXT2.

```
aspect disk_quota {

  pointcut flush(register struct mount *mp, int flags, struct proc *p):
    execution(int ffs_flushfiles(mp, flags, p))
    || execution(int ext2_flushfiles(mp, flags, p));

  pointcut sync(struct mount *mp):
    execution(int ffs_sync(mp, int, struct ucred*, struct proc*))
    || execution(int ext2_sync(mp, int, struct ucred*, struct proc*));

  pointcut vget(struct inode *ip):
    execution(void ufs_ihashins(ip))
    && cflow(execution(int ffs_vget(struct mount*, ino_t, struct vnode**)))
             || (execution(int ext2_vget(struct mount*, ino_t, struct vnode**)));

  ...

  around(register struct mount *mp, int flags, struct proc *p):
    flush(mp, flags, p)
  {
    register struct ufsmount *ump;
    ump = VFSTOUFS(mp);
    if (mp->mnt_flag & MNT_QUOTA) {
      int i;
      int error = vflush(mp, NULLVP, SKIPSYSTEM|flags);
      if (error)
        return (error);
      for (i = 0; i < MAXQUOTAS; i++) {
        if (ump->um_quotas[i] == NULLVP)
          continue;
        quotaoff(p, mp, i);
      }
    }
    return proceed(mp, flags, p);
  }
```

These pointcuts correspond to file system operations in both FFS and EXT2, on which disk quota advice applies.

```
  after(struct mount *mp):
    sync(mp)
  {
    qsync(mp);
  }


  before(struct inode *ip):
    vget(ip)
  {
    int i;
    for (i = 0; i < MAXQUOTAS; i++)
      ip->i_dquot[i] = NODQUOT;
  }
  ...
}
```

Figure 2. Shared advice in the VFS portion of a quota aspect: (a) `around` prevents `ffs/ext2_flushfiles` from executing if `vflush` returns an error; (b) `after` attaches `qsync` to `ffs/ext2_sync`; (c) `before` attaches quota operations to all executions of `ufs_ihashins` in the control flow of `ffs/ext2_vget`.

desired to be kept free.

Though the rationale behind the different thresholds is not immediately apparent to a non-expert and is not documented in the original implementation, it is clear that thresholds are context sensitive. Coalescing them into one module brings these differences to light. Because they appear side by side, it should be easier for the original implementor to document the rationale.

With respect to disk quotas, looking at the pointcut declarations in Figure 1 is like looking at a more detailed version of the structural relationships outlined in Table 2. We can see which core file system functions and values are involved, along with their similarities and differences with respect to quota. We can now reason about and configure quota across these file systems, sharing its implementation where appropriate.

This perspective can be used to eliminate redundancy and more easily ensure the consistent application of quota operations across file systems. In particular, half the code indicated by the overlap in Table 2 can be eliminated because it can now be shared between file systems.

2.4. Other crosscutting concerns

In addition to page daemon activation and disk quotas, we are exploring the aspect-oriented implementation of elements of prefetching, scheduling, networking and profiling.

Prefetching involves coordination between high level allocation of pages in VM, and subsequent possible low level deallocation in file systems. Tracing the page-fault path in the FreeBSD v3.3 implementation requires traversing 5 files, 2 levels of function tables, and 4 changes in variable names. Our preliminary aspect-oriented refactoring of normal and sequential prefetching as path-specific customizations is reported in [6].

Scheduling code spans interrupt handlers, device drivers, and process synchronization. One of the challenges in the development of Bossa, a domain specific language for schedulers, was to precisely identify all the scheduling points, or circumstances under which the scheduler is activated throughout the OS [4]. Extending the scheduler to respond to Bossa-defined scheduling events requires access to the context of the scheduler invocation. To get an idea of how extensive the challenge is to track this context, Table 3 shows the functions that call the scheduler with `mi_switch` in FreeBSD 4.4, along with the number of places where those callers are called.

Callers of mi_switch	*Number of Calls to Callers*
`tsleep`	483
`await`	19
`exit`	95
`issignal`	95
`uio_yield`	4
`usrret`	9

Table 3. Control Flow to mi_switch in FreeBSD 4.4

Networking involves some concerns that run the length of the protocol stacks of communicating processes. Optimizing or customizing a protocol often requires introducing integrated layer processing and/or passing additional parameters to new functionality introduced at each layer. Something as simple as suppressing checksums to improve performance of consenting processes not concerned with data integrity requires symmetrical changes to the sending and receiving stacks. One of the contributions of both the *x*-kernel [8], a framework for implementing network protocols, and later Plexus [7], an extensible protocol architecture for application-specific networking, was the use of protocol graphs for representing standard protocols, with augmentation for modified functionality. These systems used these graphs to represent new protocols in terms of a cohesive set of related modifications to the standard.

Profiling inherently involves action at a variety of points in the system. Whether it be for tracing execution, verifying system rules, or as a basis for building more sophisticated gray-box information and control layers [2], the ability to build a comprehensive profile is a prerequisite for dependable systems. Preprocessor directives are commonly used to introduce unpluggable profiling code in the kernel. The `/dev/usb` directory in FreeBSD 4.4 contains approximately 50 such `#ifdef DIAGNOSTIC` statements scattered throughout roughly 10,000 lines of code. System-wide, there are 314 `#ifdef DIAGNOSTIC` directives.

3. AspectC runtime

Like AspectJ, most AspectC constructs are resolved at compile time. They introduce no more overhead than a call to an inlineable function containing the advice body. (Though the pre-processor could inline these directly, it currently does not, in order to help make the pre-processor output more readable.)

But `cflow` is a dynamic construct and hence has runtime overhead associated with it. We follow the AspectJ implementation model for `cflow`, in which the overhead is distributed across executions of functions

Granularity	*Cflow Function*	*Overhead (nanoseconds)*
per-process	cflow_add_pid_entry	777
	cflow_del_pid_entry	141
per-call	cflow_push	79
	cflow_pop	80
	cflow_test	86
	cflow_get	73

Table 4. Microbenchmarks for core `cflow` overhead.

that are *cflow-tested*, and dispatch to advice involving a cflow test.

In the example from Section 1.2, a `cflow_push` and `cflow_pop` are effectively added to the code for `high_level`. A `cflow_test` is effectively added to `low_level`, as part of testing whether the advice should run. If the advice does run, `cflow_get` is called to access the parameters. These push/pop/test/get operations would all use a process-local stack specifically dedicated to `high_level`.

Our current implementation of the push/pop/test/get runtime routines is trivially naive. An open hash table tracks this information on a per-process basis. A pool of entries, sufficiently large to track the maximum number of processes in the system, is statically allocated at boot time. Each entry tracks the necessary `cflow` information for a single process, uniquely identified by the process identifier (PID).

Table 4 provides microbenchmarks for our prototype AspectC runtime. These benchmarks were taken on a 700MHz Pentium-III processor. The first two rows in the Table show the costs of adding and deleting hash table entries during process initialization and tear-down. The next four rows show the per-call costs of the other push/pop/test/get routines.

4. Future work and open issues

In order to more rigorously assess the impact aspects have on kernel code, issues of scalability, configurability, extensibility, evolability and performance require in depth cost/benefit analysis. Improving modularity of OS kernel code will not be meaningful if aspects substantially adversely impact performance. Specifically, we need to know the costs associated with sophisticated compositions of aspects in terms relative to a tangled implementation. In kernel code, we are often faced with a fine granularity of tangling of multiple concerns, making refactoring challenging. For example, IP security (`IPSEC`) and IPv6 functionality (`INET6`), are configured with compiler directives and implemented using a shared `if` statement and `goto` labels as shown below:

```
#ifdef IPSEC
#ifdef INET6
  if (isipv6) {
    if (inp != NULL &&
        ipsec6_in_reject_so(m,
          inp->inp_socket)) {
      ipsec6stat.in_polvio++;
      goto drop;
    }
  } else
#endif /* INET6 */
  if (inp != NULL && ipsec4_in_reject_so(m,
                       inp->inp_socket)) {
    ipsecstat.in_polvio++;
    goto drop;
  }
#endif /*IPSEC*/
```

Though AspectC is modeled after AspectJ, there are several important differences that must be addressed. This includes C-specific issues, such as code-bloat associated with the C preprocessor. Working within the kernel may also demand that we explore different kinds of runtime support than is required for user-level AOP.

5. Related work

Structuring kernel code to make it more amenable to change common theme in many research projects. Customization has been a leading motivating factor. Support for application-specific customization of services range from operating systems that target specific policy, such as paging in Mach [13], to those that have taken a more comprehensive approach, such as the use of reflection in Apertos [21]. Approaches that structure client participation in OS policy include scheduler activations [1], active networking [18], policy servers in user space [20, 9, 12], and application-specific extensions [15, 7, 17, 19]. Approaches aimed at improving structure in general include the use of frameworks for end-to-end optimization [16], domain specific languages [14, 4], and gray-box techniques [2].

Our work is particular in its focus on the modular implementation of existing crosscutting concerns that map to key decisions in kernel design.

6. Conclusions

One of the reasons kernel code is brittle is that some key system concerns naturally crosscut others. Crosscutting concerns are not modular when implemented using traditional techniques – their implementation is scattered and tangled throughout other modular units of the system.

AOP is poised to help. It offers mechanisms that allow us to explicitly structure crosscutting concerns

as new, first class modules called aspects.

In this paper, a subset of AOP mechanisms are applied in the context of two examples, page daemon activation and disk quotas. The benefits of implementing these particular concerns as aspects are improved comprehensibility and configurability. By further studying the ways in which aspects can be used to improve the modularity of key design decisions in an existing kernel, we hope to promote dependability in future systems.

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1), Februray 1992.

[2] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *18th ACM Symposium on Operating System Principles (SOSP)*, 2001.

[3] AspectC. www.cs.ubc.ca/labs/spl/aspects/aspectc.html.

[4] L. P. Barreto and G. Muller. Bossa: a language-based approach for the design of real time schedulers. In *Proceedings of the 23rd IEEE Real-Time Systems*, 2002.

[5] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *18th ACM Symposium on Operating System Principles (SOSP)*, 2001.

[6] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. In *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT In ternational Symposium on th Foundations of Software Engineering (FSE-9)*, 2001.

[7] M. E. Fiuczynski and B. N. Bershad. An extensible protocal architecture for application-specific networking. In *Winter Usenix Conference*, 1996.

[8] N. Hutchinson and L. Peterson. The *x*-kernel: An architecture for implementing network protocols. In *IEEE Transactions on Software Engineering*, volume 17(1), 1991.

[9] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, October 1997.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. AspectJ home page. http://www.aspectj.org.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[12] C. Maeda. Flexible system software through service decomposition. In *OOPSLA*, August 1994.

[13] D. McNamee and K. Armstrong. Extending the Mach external pager interface to allow user level page replacement policies. In *Technical Report UWCSE 90-09-05, University of Washington*, September 1990.

[14] G. Muller, C. Consel, R. Marlet, L. P. Barreto, F. Merillon, and L. Reveillere. Toward robust oses for appliances: A new approach based on domain-specific languages. In *Eurpoean Workshop on Operating Systems*, 2000.

[15] P. Pardyak and B. Bershad. Dynamic binding for an extensible system. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1996.

[16] A. Sane, A. Singhai, and R. Campbell. Framework design for end-to-end optimization. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1998.

[17] C. Small and M. Seltzer. A comparison of OS extension technologies. In *Proceedings of the USENIX Conference*, 1996.

[18] D. Tennenhouse and D. Wetherall. Towards an active network architecture. *ACM Computer Communications Review*, 26(2):5–18, April 1996.

[19] A. C. Veitch and N. C. Hutchinson. Kea - a dynamically extensible and configurable operating system kernel. In *Proceedings of the 1996 Third International Conference on Configurable Distributed Systems (ICCDS)*, 1996.

[20] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. In *Communications of the ACM*, volume 17(6), 1974.

[21] Y. Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 1992.