Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code

Yvonne Coady
University of British Columbia
201 2366 Main Mall
Vancouver, BC V6T 1Z4
ycoady@cs.ubc.ca

Gregor Kiczales
University of British Columbia and
Intentional Software Corporation
500 108th Ave NE #1050
Bellevue, WA 98004
gregor@intentsoft.com

ABSTRACT

The FreeBSD operating system more than doubled in size between version 2 and version 4. Many changes to primary modularity are easy to spot at a high-level. For example, new device drivers account for 38% of the growth. Not surprisingly, changes to crosscutting concerns are more difficult to track. In order to better understand how an aspect-oriented implementation would have fared during this evolution, we introduced several aspects to version 2 code, and then rolled them forward into their subsequent incarnations in versions 3 and 4 respectively. This paper describes the impact evolution had on these concerns, and provides a comparative analysis of the changes required to evolve the original versus aspect-oriented implementations.

Our results show that for the concerns we chose, the aspect-oriented implementation facilitated evolution in four key ways: (1) changes were better localized, (2) configurability was more explicit, (3) redundancy was reduced, and (4) extensibility aligned with an aspect was more modular. Additionally, we found the aspect-oriented implementation had negligible impact on performance.

1. INTRODUCTION

The FreeBSD operating system has grown from roughly 212,000 lines of code (LOC) in version 2 (v2), to 357,000 LOC in v3, and 474,000 LOC in v4. Many evolutionary changes to primary modularity are easy to appreciate at a high-level. For example, compared with v2, new device drivers in v4 introduced 581 new subdirectories and files, accounting for over 100,000 new LOC, or 38% of the growth. Changes to crosscutting concerns however, are not as straightforward to track. For example, the number of places where the page daemon may be activated was reduced by 50% between v2 and v4, whereas the number of places a process can block in device driver code grew by two orders of magnitude.

In order to better understand how an aspect-oriented [11] implementation would have fared from an evolutionary standpoint, we re-factored four crosscutting concerns in v2 code into aspects: waking the page daemon, prefetching for mapped files, quotas for disk usage, and tracing blocked processes in device drivers. These implementations were then rolled forward into their subsequent incarnations in v3 and v4 respectively. This paper describes the impact evolution had on these concerns, and provides a comparative analysis of the changes required to evolve the original versus aspect-oriented implementations.

Our experiment was designed to explore a range of evolutionary scenarios – each concern evolved differently. In terms of the number of places the implementation of each concern had effect: one became consistently smaller, one grew then became smaller, one grew only incrementally, and one grew by an order of magnitude in each version. In each case we found that, with tool support, the aspect-oriented implementation better facilitated independent development and localized change. In three cases, configuration changes mapped directly to modifications to pointcuts and makefile options. In one case, redundancy was significantly reduced. Finally, in one case, the implementation of a system-extension aligned with an aspect was itself better modularized. These results suggest the ways in which an aspect-oriented implementation of these concerns would have been more evolution-friendly than the original implementation.

The paper starts with an introduction to the intent and inherent structure of each crosscutting concern (Section 2), and overviews their original implementations in versions 2, 3, and 4 (Section 3). After a brief introduction to AspectC [3], each concern's aspect-oriented implementation and changes required to evolve it are presented (Section 4). A comparison with the impact of evolution on the original implementation then suggests specific ways in which each aspect facilitated change, a collective analysis summarizes the results, and the costs associated with the AspectC runtime prototype are provided (Section 5). Finally, future work and open issues are described (Section 6).

2. Intent and Inherent Structure

Before considering any implementation, it is useful to first establish what each concern is intended to do, and identify its inherent structure.

2.1 Page Daemon Activation

The page daemon frees physical memory when the number available pages falls below some threshold. The daemon

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Conference '00, Month 1-2, 2000, City, State.
Copyright 2000 ACM 1-58113-000-0/00/0000…$5.00.

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 necessary. The daemon's activation is designed to assess context-specific thresholds whenever the system could be running low on available pages.

The inherent structure of page daemon activation is a set of context-specific triggers within the virtual memory system and the file buffer cache. Essentially, activation crosscuts operations that consume available pages.

2.2 Prefetching

Fetching pages from disk is a relatively expensive operation. Prefetching is a heuristic designed to amortize costs by bringing in additional pages that may be required in the near future. As a heuristic, we want to prefetch if it is cost effective to do so. The virtual memory system thus suggests pages for prefetching, but the file system decides whether or not to actually get them.

The inherent structure of prefetching is shaped by specific execution-paths that retrieve pages from disk. Prefetching crosscuts virtual memory and file systems, coordinating high-level allocation and low-level de-allocation of prefetched pages.

2.3 Disk Quotas

It is often necessary to apportion disk space between multiple users. Disk quotas are designed to track disk utilization and enforce limits for all users and groups.

The inherent structure of quota is a set of low-level disk space related operations that are consistently monitoring/limiting all disk usage. Quota crosscuts disk-consuming operations from multiple file systems.

2.4 Blocking in Device Drivers

Effective scheduling for the CPU requires processes to sometimes explicitly surrender the CPU and block, allowing another process to run. Blocking in device driver code is designed to ensure the CPU is kept busy while processes wait for device I/O.

The inherent structure of diagnostic statements related to blocking behaviour in device drivers shadows all points in the system where a process could indefinitely block on a device. Tracking process blocking in device drivers crosscuts all device-specific operations involved with I/O.

3. ORIGINAL IMPLEMENTATION

This section describes each concern's original implementation, and its evolution across three versions of FreeBSD[1].

3.1 Page Daemon Activation

The function `pagedaemon_wakeup()`, and its lower level counterpart, `wakeup(&vm_pages_needed)`, are invoked from 16 places in v2, 9 places in v3, and 8 places in v4, as overviewed by the partial source tree in Figure 1.

[1] Specifically, this study uses versions 2.2, 3.3 and 4.4.

As the virtual memory (VM) system evolved, the internals of many functions in the swap pager (`swap_pager.c`) were significantly revamped. Among many other changes, 7 triggers for page daemon wakeup were eliminated between v2 and v3. Between v3 and v4, the reworking of the function to allocate pages resulted in the loss of two triggers, and the introduction of a new synchronization operation in VM added a new low-level activation of the daemon.

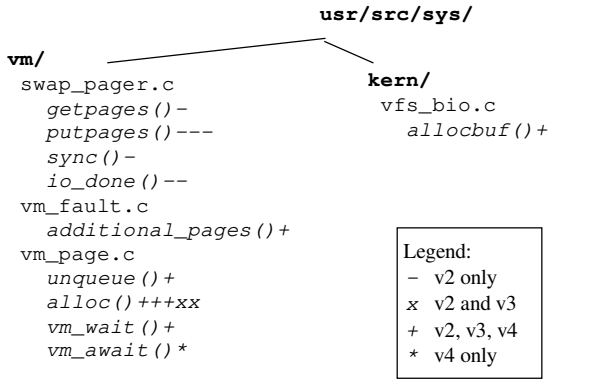


Figure 1. Partial source tree showing functions where activation of the page daemon may occur.

3.2 Prefetching for Mapped Files

Tracing the page-fault path in FreeBSD v3 requires traversing 5 files, 2 levels of function tables, and 4 changes in variable names, as previously reported in [6]. Figure 2 overviews key functions involved with the original implementation of this coordinated activity.

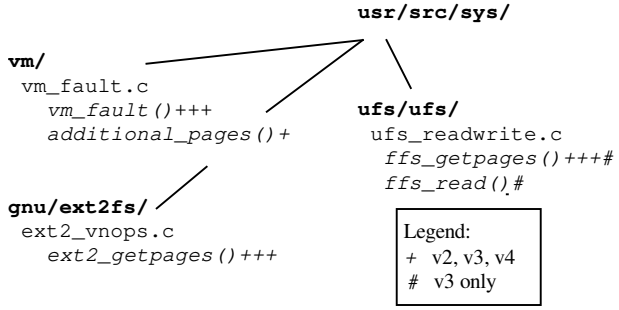


Figure 2. Partial source tree showing functions where prefetching for mapped files occurs.

The most significant evolutionary change to prefetching was between v2 and v3, when sequential mode prefetching in one file system was modified to be substantially more aggressive than normal mode. Control flow was diverted to the file system read operation where additional asynchronous prefetching would be applied, bringing pages into the buffer cache. This implementation of sequential mode prefetching was short lived however, as it was subsequently removed between v3 and v4.

3.3 Disk Quotas

Disk quotas are an optional feature of FreeBSD, configured through a combination of settings in both a kernel configuration file and on a per-file system basis with 46 compiler directives in

three file system subdirectories in v4: 37 `#ifdef QUOTA` preprocessor directives in UFS, FFS, and EXT2, and 9 `#if QUOTA` directives in EXT2. Quota spans 22 functions from 11 files in these file systems, as highlighted by the files shown in Figure 3. The file names in common from the different subdirectories in the Figure reflect an overlap in the implementation of quota.

Quota grew incrementally, as a result of new features being added to the file systems: from v2 to v3, it spread to 16 new places, and from v3 to v4 to one new place in FFS.

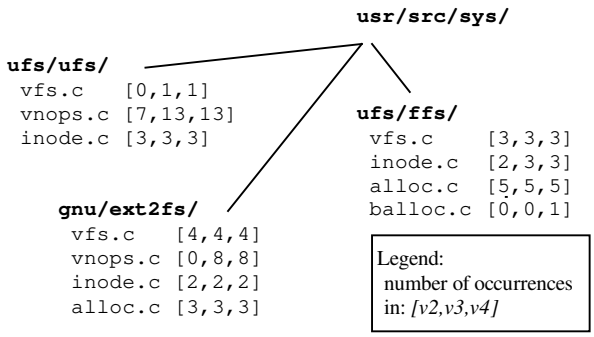


Figure 3. Partial source tree showing files where quota operations for disk usage occurs.

3.4 Blocking in Device Drivers

When waiting for device I/O, a process blocks by calling `tsleep`. Among other things, `tsleep` is passed a value to block on and a timeout after which the process will wake-up if it has not been unblocked. Essentially, a timeout of zero means a process will sleep until explicitly woken.

Driver code is primarily in the `/dev` subdirectory. Its growth, number of calls to `tsleep` in total, and number of calls to `tsleep` without timeout, are overviewed in Table 1.

Table 1. Device code from the `/dev` subdirectory.

version	LOC	devices (subdirs)	calls to `tsleep()`	`tsleep()` w/ no timeout
2	8400	7	5	2
3	46900	27	55	11
4	114800	69	110	22

4. EVOLVING SYSTEM ASPECTS

This section highlights the changes required to evolve the aspect-oriented implementation of each concern. For each aspect, its internal structure is overviewed, a sample of its core AspectC implementation from one or more of the FreeBSD versions is provided, and the impact of major evolutionary changes are described in terms relative to its structure.

We have developed a prototype for AspectC [3] based on AspectJ [10]. Though the prototype is still a work in progress, compared to AspectJ, AspectC has function *call* and *execution* join points, *call*, *execution*, *cflow* and *within* pointcuts, and *before*, *after* and *around* advice.

4.1 Page Daemon Activation Aspect

Re-factoring page daemon activation as an aspect in v2 involved removing code that controlled activation from the operations it crosscut, and reintroducing the same functionality using AspectC.

4.1.1 Internal Structure and Implementation

Since thresholds depend upon several system parameters, it is important that the aspect is structured to make three things clear: (1) the contexts of the threshold checks, (2) the specifics of the thresholds used, and (3) the relationships between the contexts and the thresholds, system-wide.

Figure 4 shows some of the core implementation of the page daemon wakeup aspect common to v2, v3 and v4 (helper functions not shown). The two named pointcuts identify specific points in kernel execution when paging may be needed. That is, they explicitly name the contexts of the threshold checks: when unqueuing available pages, and when allocating buffers respectively. Each of the two advice declarations uses one of these named pointcuts to define what page threshold should be used at each point, and wakes the daemon accordingly.

```
aspect page_daemon_wakeup {

  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)));

  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)));

  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();
  }

  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;
  }

  …
}
```

Figure 4. Core page daemon wakeup for v2, v3 and v4.

The first pointcut, `unqueuing_available_pages`, names points in the execution when the `vm_page_unqueue` function executes in the control flow of any one of the four functions listed. Furthermore, this pointcut makes the `vm_page_t` parameter available to advice that uses this pointcut. The four functions listed were the only functions that called `vm_page_unqueue` in the original implementation. There were, however, four other functions that called a related function, `vm_page_unqueue_nowakeup` – identical to `vm_page_unqueue` except for the wakeup of the daemon. During re-factoring, we

removed the `nowakeup` version of this function, and replaced calls to it with calls to `vm_page_unqueue`.

The first advice executes `around` these points in the system, using the AspectC keyword `proceed` to allow the originally intended function, `vm_page_unqueue`, to execute. Since it is only within the control flow of the functions explicitly identified by `cflow` that the advice is triggered, no other functions that call the unqueue operation trigger the advice, thus better localizing daemon activation, while preserving functionality from the original implementation.

For each of the advice, we can see the context of the threshold by looking at the pointcut, the details of the threshold by looking at the advice body, and the relationships between the two, system-wide, by looking at the aspect.

4.1.2 Impact of Evolution

Evolutionary changes that impact this aspect are a composite of three separate sets of changes to: (1) the swap pager, (2) the VM page operations, and (3) page daemon activation code, itself. Changes arising from (1) and (2) required adding/deleting specific pointcuts and advice – essentially configuration changes within the aspect. Changes to daemon activation directly mainly involved re-factoring of threshold checks – introducing some helper functions to reduce redundancy and increase readability. These changes were also local to the aspect.

4.2 Prefetching for Mapped Files Aspect

In re-factoring prefetching as an aspect in v2, we had to re-factor some of the existing VM and file system operations to allow for more fine-grain composition with the aspect than the original implementation would support. This initial re-factoring did not require further modification as the versions evolved.

It is important to note that the coordination is between VM and a given file system, specifically FFS or EXT2 in this example, but not between the file systems themselves. In fact, the optimized implementation of sequential mode prefetching associated with the evolution between v2 an v3 was only introduced to FFS, the native FreeBSD file system, and not to EXT2, a port of the Linux file system to FreeBSD.

4.2.1 Internal Structure and Implementation

A portion of the single aspect for both normal and sequential mode mapped file prefetching common to v2 and v4 is shown in Figure 5. The aspect structures coordination between the high-level allocation for prefetched pages and their possible subsequent low-level de-allocation, further down the execution path. Most of the details involved in this aspect are similar to those presented in [6], but are briefly overviewed here to provide background for its evolutionary scenario.

The pointcuts name the high-level and the low-level parts of the execution paths involved, `vm_fault_path` and `getpages_path` respectively, exposing the necessary parameters at each point. The `getpages_path` covers corresponding operations from two different file systems, FFS and EXT2, respectively. The advice then coordinates allocation/de-allocation along these control flows – the first advice allocates pages for prefetching, while the next two may de-allocate pages if it is no longer cost effective to retrieve them.

```
aspect mapped_file_prefetching {

  pointcut vm_fault_path(vm_map_t map):
    cflow(execution(int vm_fault(map, ..)));

  pointcut getpages_path(vm_object_t obj,
                         vm_page_t*  plist, int n, int fpage):
    cflow(execution(int ffs_getpages(obj,plist,n,fpage)))
          || execution(int vnode_leaf_pager_getpages(obj,
                                                plist,n,fpage)));

  before(vm_map_t map, vm_object_t obj,
         vm_page_t* plist, int n, int fpage):
    execution(int vnode_pager_getpages(obj, plist, n, fpage))
    && vm_fault_path(map)
  {
      if (obj->declared_behaviour != RANDOM) {
        vm_map_lock(map);
        plan_and_alloc_prefetch_pages(obj, plist, n, fpage);
        vm_map_unlock(map);
      }
  }

  after(vm_object_t obj, vm_page_t* plist, int n,
        int fpage, int valid):
    execution(valid check_valid(..))
    && getpages_path(obj, plist, n, fpage)
  {
      if (valid)
        dealloc_all_prefetch_pages(obj, plist, n, fpage);
  }

  after(vm_object_t obj, vm_page_t* plist, int n,
        int fpage, struct transfer_args* trans_args):
    execution(int calc_range(trans_args))
    && getpages_path(obj, plist, len, fpage)
  {
      dealloc_noncontig_prefetch_pages(obj, plist, n,
                                       fpage, trans_args );
  }
  …
}
```

Figure 5. Core prefetching common to v2 and v4.

4.2.2 Impact of Evolution

Unlike the page daemon aspect, none of the evolutionary changes to prefetching are the result of revamping the functions prefetching crosscuts. Instead, the changes are concentrated on the prefetching concern. The most significant evolutionary change was the use of the file system read-path for sequential mode between v2 and v3. In the aspect-oriented implementation, we split the existing v2 aspect into two separate, but related, aspects for normal and sequential modes respectively.

In v4, this sequential mode optimization was removed. In the aspect-oriented implementation, this meant removing the sequential mode aspect from the makefile, and modifying the normal mode aspect to reunite the two.

4.3 Disk Quota Aspect

Re-factoring quota in v2 involved separating the sections of quota code associated with compiler directives from the file system operations it crosscut. Some of this separation involved re-factoring file system operations to allow for composition.

4.3.1 Internal Structure and Implementation

Close inspection of the code in each file system reveals that the 17 compiler directives in EXT2 introduce identical functionality to the corresponding operations in the union of UFS and FFS. That is, all the quota code in EXT2 is redundant.

Figure 6 shows a portion of the aspect-oriented implementation of disk quota common to v2, v3 and v4. Specifically, 5 `#ifdefs` in FFS and EXT2 are replaced by the 2 advice shown in the

aspect. The pointcuts name the corresponding operations from the different file systems that are associated with shared quota operations. For example, the first pointcut, `flushfiles`, is associated with the flushfile operation in either FFS or EXT2. The around advice that uses this pointcut provides a single shared implementation of the associated quota operation.

```
aspect disk_quota {

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

  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):
    flushfiles(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);
  }

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

Figure 6. Core quota common to v2, v3, and v4.

4.3.2 Impact of Evolution

Changes to disk quota between v2 and v3 resulted from the introduction of a new feature for file servers, also implemented using compiler directives, which automatically assigned the ownership of a new file to be that of the enclosing directory.

Since ownership relates to disk consumption, this new feature was interleaved with quota code, introducing 14 new compiler directives for quota over 4 existing UFS and EXT2 operations. Between v3 and v4, a new FFS operation was introduced, also requiring quota tracking.

In order to evolve the aspect, we re-factored the original implementation of the new feature introduced within the UFS and EXT2 operations to allow composition, but retained its use of compiler directives as it did not impact the aspect-oriented implementation. Evolution thus primarily consisted of adding pointcuts and advice as needed to incrementally extend its configuration to include new functionality.

4.4 Tracing Blocking in Device Drivers Aspect

Device driver code has the highest rate of growth and, not surprisingly, the highest rate of bugs in the kernel [5]. Within driver code in v2, we introduced an aspect to track processes that block on device operations without a timeout. Though this is not an error, these situations are of particular interest when diagnosing problematic behaviour associated with device drivers, as processes may block indefinitely.

4.4.1 Internal Structure and Implementation

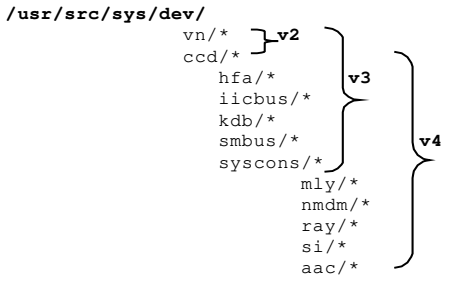
Figure 7 shows the aspect that tracks information about all processes blocked in driver code with a timeout of zero. The pointcut uses `within` to restrict the set of calls to `tsleep` to those invoked from within driver code, i.e. within the `/usr/src/sys/dev` subdirectory.

```
aspect device_blocked_notimeout {

  pointcut block_in_device(void* ident, int priority,
                           const char* wmesg, int timo):
    call(int tsleep(ident, priority, wmesg, timo))
    && within("/usr/src/sys/dev");

  around(void* ident, int priority,
         const char* wmesg, int timo):
    block_in_device(ident, priority, wmesg, timo)
  {
    if (timo == 0)
      printf("Blocking in device, no timeout:
              %s pid: %d\n", wmesg, curproc->p_pid);
    return proceed(ident, priority, wmesg, timo);
  }
}
```

Figure 7. Complete aspect for tracking in v2, v3 and v4.

4.4.2 Impact of Evolution

Though the number of calls to `tsleep` in driver code grew from 5 in v2, to 55 in v3, and 110 in v4, this aspect did not require modification, as it automatically applies to all code in the `/dev` subdirectory. Figure 8 shows the number of devices (or subdirectories) in each version of FreeBSD that trigger this aspect. Specifically, it shows that in v2, of the 7 devices total, 3 of them sleep, and 2 of those 3 (shown in the list of subdirectories) sleep without timeout; in v4, of the 69 devices total, 27 of them sleep, and 11 of those sleep without timeout.



	v2	v3	v4
devices(subdirs)	7	27	69
devs that tsleep	3	13	27
tsleep w/o timeout	2	7	11

Figure 8. Drivers that use `tsleep` in v2, v3 and v4.

5. ANALYSIS

This section presents an analysis of the results of our experiment. First, the ways in which the original evolution of each concern was problematic is overviewed, then, the general ways in which the aspects addressed these problems are summarized. Finally,

micro-benchmarks provide a cost analysis associated with runtime support for the aspect-oriented implementation.

5.1 Evolving Scattered and Tangled Code

In their original implementations, these concerns are non-modular – scattered and tangled in an unclear way throughout the primary modularity they crosscut. The specific problems that developed during the evolution of the original implementation of each concern are outlined here.

5.1.1 Page Daemon Wakeup

In the original implementation, identifying precisely when and why the page daemon may be activated, system-wide, is difficult because the code is spread out. This property is most likely the reason why some activation points were re-factored, while others were not.

Though the rationale behind the different thresholds is not immediately apparent to a non-expert, nor is documented in the original implementation, it is clear that thresholds require context, for example: (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, the buffer cache uses the more conservative value of `free_min`, the minimum number of pages desired to be kept free. These subtle differences are critical for understanding daemon activation system-wide, but difficult to appreciate in the original implementation.

Minor re-factoring of threshold calculations were included in the evolution of some code in VM, but this re-factoring was not consistently applied to all the thresholds involved with activation. Presumably this is because the parts of the system that daemon activation crosscuts – VM and the buffer cache – did not evolve simultaneously.

5.1.2 Prefetching

Though the optimized version of sequential mode prefetching introduced in v3 involved only a relatively small number of changes to the system, it introduced an important new interaction between VM, the file system, and the buffer cache that did not previously exist. At the level of the interfaces of the primary modularity involved, this new interaction was impossible to detect. Its subsequent removal in v4 required selective editing of code within FFS operations.

5.1.3 Disk Quotas

Implementing quota with 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.

Using preprocessor directives to configure a crosscutting concern like this, however, is cumbersome for three reasons: (1) it makes it difficult to reason comprehensively about quota, and identify the structural relationships that hold, (2) directives can obscure reading of the file system code quota is scattered in, and (3) drift can occur between portions of quota code that should be identical.

The nature of drift we found did not appear to introduce inconsistencies in quota. For example, the quota code embedded in `flushfile` operations from FFS and EXT2, from two different files respectively, is shown in Figure 9. The differences in the declaration of the variable and the assignment of the error condition are a merely matter of style, but are representative of the type of drift that currently exists.

```
int ffs_flushfiles(…) { // 13 LOC
…
#ifdef QUOTA
  if (mp->mnt_flag & MNT_QUOTA) {
    int i;
    error = vflush(mp, 0, SKIPSYSTEM|flags);
    if (error)
      return (error);
    for (i = 0; i < MAXQUOTAS; i++) {
      if (ump->um_quotas[i] == NULLVP)
        continue;
      quotaoff(p, mp, i);
    }
  }
#endif
…
}


int ext2_flushfiles(…) { // 9 LOC
…
#if QUOTA
  int i;
#endif
…
#if QUOTA
  if (mp->mnt_flag & MNT_QUOTA) {
    if ((error =  vflush(mp, 0, SKIPSYSTEM|flags))!=0)
      return (error);
    for (i = 0; i < MAXQUOTAS; i++) {
      if (ump->um_quotas[i] == NULLVP)
        continue;
      quotaoff(p, mp, i);
    }
  }
#endif
…
}
```

Figure 9. Quota in FFS and EXT2 `flushfile` operations.

5.1.4 Device Blocking

Driver code is problematic, in part, because it is the result of multiple independent developers interacting with subtle OS specific protocols. Though many of these protocols may be simple to state, they are hard to manually ensure in their scattered and tangled form. Further, extensions to the system-wide scheduling policy of an OS, such as the event-based scheme in Bossa [4], necessarily involve invasive, non-modular, modifications to a rapidly growing number of points in the system to detect events such as blocking.

5.2 General Improvements with Aspects

The results of our experiment are summarized in Table 2, highlighting evolution through the versions, structural challenges, the differences between the original versus aspect-oriented implementations, and the benefits of the aspects involved. These results indicate that the major differences between the original and aspect-oriented implementations of these concerns involve four key related properties: changeability, configurability, redundancy and extensibility. Each of these properties is discussed further in the subsections that follow.

Table 2. Summary of results.

concern	*original*			*major evolution*	*structural challenge*	*original* / *aspect*	*benefits*
	version 2	*version 3*	*version 4*				
page daemon wakeup	16 places 9 functions 4 files 2 subdirs	9 places 5 functions 3 files 2 subdirs	8 places 6 functions 3 files 2 subdirs	revamping of code it crosscuts: VM and buffer cache	multiple context specific thresholds	scattered activation / textually localized	independent development & localized change
prefetching for mapped files	9 places 4 functions 3 files 3 subdirs	11 places 5 functions 3 files 3 subdirs	9 places 4 functions 3 files 3 subdirs	change in design of sequential mode	new subsystem interaction along execution paths	internal to functions / explicit control flow	explicit subsystem interaction & pluggability in makefile
disk quota	29 places 19 functions 9 files 3 subdirs	45 places 21 functions 10 files 3 subdirs	46 places 22 functions 11 files 3 subdirs	new functionality in code it crosscuts: UFS, FFS, EXT2	configurability and sharing across file systems	#ifdefs w/ redundant code / explicit sharing	pointcut configurability & reduced redundancy
device blocking	5 places 3 functions 3 files 3 subdirs	55 places 49 functions 34 files 13 subdirs	110 places 94 functions 53 files 27 subdirs	new device drivers added to the system	consistency across rapidly growing diversity	individualized devices / centralized assessment	comprehensive coverage & further extensibility modularized

5.2.1 Changeability

Major changes were of two forms: (1) directly to the concern itself, and (2) indirectly, as a result of revamping the code the concern crosscut. In the aspect-oriented implementation of each concern considered here, changes to the concern itself were facilitated by textual locality. Changes that resulted from revamping the code the concern crosscut should be equally accessible, given tool support[2].

Though we can only speculate on what evolving these concerns simultaneously with the many other evolutionary changes required to produce a new release of the system would have been like, we believe textual locality could further address two problems in the original implementation. First, the inconsistencies that arose from non-uniform evolution of the underlying primary modularity the concerns crosscut would most likely be reduced. That is, when localized, the concern would be more likely to evolve as a unit. Second, coalescing diverse context-specific elements, such as the thresholds in daemon activation, into one module brings their differences to light, making it a more natural setting for the original implementer to document the rationale.

5.2.2 Configurability

For three of the aspects, configuration changes mapped directly to modifications to pointcuts and/or makefile options. This had particular impact on the evolution of both prefetching and disk quotas. Pluggability is key to both.

With respect to prefetching, the optimization for sequential mode prefetching introduced a new interaction between multiple subsystems. Moreover, this interaction was unique to a single file system. Explicit configuration of this interaction as an aspect better supported independent development of prefetching modes, and ultimately, one's eventual removal from the system. Unplugging the optimization in the original code required changing the internals of key file system operations.

With respect to the disk quota aspect, the pointcut declarations reveal the underlying structural relationships between corresponding file system operations. We can see which core file system functions and values are involved, along with their similarities and differences with respect to quota. Structured this way, it should be easier to expand quota by explicitly configuring it across more file systems[3]. Additionally, there is no loss of pluggability relative to the preprocessor based implementation.

5.2.3 Redundancy

Related to the increased configurability of the quota aspect is the elimination of redundancy across file systems. Though the differences are benign, there exists drift in the implementation of quota in FFS versus EXT2. The ability to specify that quota in EXT2 is the same as the other file systems eliminates redundant code, prevents drift, and ensures that quota operations are consistently applied system-wide.

5.2.4 Extensibility

Scheduling code spans interrupt handlers, device drivers, and all places in the system where process synchronization occurs. One of the challenges in the development of Bossa, a domain specific language for schedulers, is to precisely identify all the scheduling points, or circumstances under which the scheduler is activated, throughout the OS. Extending the scheduler to respond to Bossa-defined scheduling events requires access to

[2] As yet, there is no tool-support for AspectC.

[3] Currently, quota only applies to FFS and EXT2, but FreeBSD v4 supports over a dozen different file systems.

the context of the scheduler invocation. To get an idea of how extensive the challenge is to track this context, the number of calls to `tsleep` system-wide is close to 500 in v4. Changing an OS to raise scheduling events thus requires invasive modifications to hundreds of places in the system, compromising the modularity of the extension. Aligning the extension with a scheduling concern structured as an aspect, such as the device blocking aspect, thus improves the modularity of the extension.

5.3 Runtime Costs

Like AspectJ, most AspectC constructs are static – 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 that are `cflow`-tested, and dispatch to advice involving a cflow test.

In the first pointcut in the page daemon wakeup aspect from Figure 4, a `cflow_push` and `cflow_pop` are effectively added to the code for each of the four VM operations listed. A `cflow_test` is effectively added to `vm_page_unqueue`, 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 all use a process-local stack specifically dedicated to `vm_page_unqueue`.

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 3 provides micro-benchmarks for our prototype AspectC runtime. These 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.

Table 3. Runtime costs.

granularity	cflow function	overhead (nanoseconds)
per-process	`add_pid`	777
	`del_pid`	141
per-call	`push`	79
	`pop`	80
	`test`	86
	`get`	73

Though AspectC is modeled after AspectJ, there are several important differences that still 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.

6. FUTURE WORK AND OPEN ISSUES

Aspect-oriented programming proposes new mechanisms to enable the modular implementation of crosscutting concerns. This paper evaluates aspect-oriented programming in the context of four crosscutting concerns evolving across three versions of FreeBSD kernels. The ways in which aspects allowed us to make these implementations modular, the support they provided for evolution, and the costs associated with our AspectC prototype all indicate that aspect-oriented programming could improve the evolvability of OS code, but many open issues remain. Here we briefly consider the limitations of the study, the generalizability of the aspects involved, and directions for further experimentation.

6.1 Evolvability and Metrics

Limitations of the results of this study are: (1) instead of producing full successive versions of FreeBSD, we focused only on the evolution of specific concerns in isolation, (2) a single developer evolved the concerns across all versions. As a result, we are only able to speculate how the aspects would have evolved under conditions with multiple developers working on multiple concerns in concert. The realistic construction of evolvability studies and the metrics to evaluate improvements associated with aspect-oriented implementations are general areas that require further research.

6.2 Other System Aspects

We believe the four aspects considered here share structural significance with other latent system aspects. For example, system profiling and networking concerns.

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 over 300 `#ifdef DIAGNOSTIC` directives.

Networking involves concerns that run the length of the protocol stacks of communicating processes. 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. Aspects may allow us to more accurately reflect this design technique in the corresponding implementation of these protocols.

6.3 Sophisticated Compositions

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 operating systems will not be meaningful if aspects substantially adversely affect performance. Specifically, we need to determine the precise

costs associated with more sophisticated compositions of aspects in terms relative to their current implementation.

7. RELATED WORK

Structuring kernel code to make it more amenable to change is a 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 [22].

Approaches that structure client participation in OS policy include scheduler activations [1], active networking [19], policy servers in user space [21, 9, 12], and application-specific extensions [16, 7, 18, 20]. Approaches aimed at improving structure in general include the use of frameworks for end-to-end optimization [17], domain specific languages [14, 4], gray-box techniques [2], and aspect-oriented frameworks [15].

Our work focuses primarily on using AspectC to modularize existing crosscutting concerns that map to key design decisions in OS kernels.

8. SUMMARY

This study compares the evolution of four scattered and tangled concerns in kernel code with an aspect-oriented implementation of the same concerns. Localized changeability, explicit configurability, reduced redundancy and subsequent modular extensibility, are shown to be the key benefits of the aspect-oriented implementation.

9. REFERENCES

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for userlevel management of parallelism. *ACM Transactions on Computer Systems*, 10(1), Februray 1992.

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

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

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

[5] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating system errors. In *18th ACM Symposium on Operating System Principles (SOSP)*, 2001.

[6] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg 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 International Symposium on the Foundations of Software Engineering (FSE-9)*, 2001.

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

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

[9] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, October 1997.

[10] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[11] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, 1997.

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

[13] Dylan McNamee and Katherine 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] Gilles Muller, Charles Consel, Renaud Marlet, Luciano Porto Barreto, Fabrice Merillon, and Laurent Reveillere. Toward robust OSes for appliances: A new approach based on domain-specific languages. In *Eurpoean Workshop on Operating Systems*, 2000.

[15] Paniti Netinant, Constantinos Constantinides, Tzilla Elrad, Mohamed Fayad. Supporting aspectual decomposition in the design of adaptable operating systems using aspect-oriented frameworks. In *Proceedings of 3rd Workshop on Object-Orientation and Operating Systems ECOOP-OOOWS* 2000.

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

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

[18] Christopher Small and Margo Seltzer. A comparison of OS extension technologies. In *Proceedings of the USENIX Conference*, 1996.

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

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

[21] 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.

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