Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code

Yvonne Coady, Gregor Kiczales, Mike Feeley and Greg Smolyn
University of British Columbia

Abstract

Layered architecture in operating system code is often compromised by execution path-specific customizations such as prefetching, page replacement and scheduling strategies. Path-specific customizations are difficult to modularize in a layered architecture because they involve dynamic context passing and layer violations. Effectively they are vertically integrated slices through the layers.

An initial experiment using an aspect-oriented programming language to refactor prefetching in the FreeBSD operating system kernel shows significant benefits, including easy (un)plugability of prefetching modes, independent development of prefetching modes, and overall improved comprehensibility.

Keywords

aspect-oriented programming, software modularity, operating system design

1. INTRODUCTION

Almost 35 years ago, Dijkstra proposed a layered architecture for operating systems, primarily for the purpose of hiding complexity and supporting the development process [2]. But in practice, OS code does not have a clean layered structure. A study of OS/360 done in the early 70s showed that the average number of modules involved in a given change rose from 14.6% in releases 2–6, to 31.9% in releases 12–16 due to "unintentional interaction" among components [12]. Windows NT requires third party file system designers to be intimately familiar with "patterns of interaction" that exist between the file system, cache manager and virtual memory manager [25]. Recently, Engler et al. captured popular sentiment with an observation that disparate parts of operating system kernel code are linked together in a "fragile and intricate mess" [3].

One significant source of modularity problems is that OS kernels involve a number of *path-specific customizations* critical to delivering required performance and functionality. Path-specific customizations involve tailoring a service based on the context in which it is invoked. For example, prefetching on page faults to a randomly accessed file must be different than for a sequentially accessed file.

Two critical properties of path-specific customizations make them difficult to modularize in a layered architecture: (1) They depend on *dynamic context information* (e.g. what is being done with the data that caused the page fault). (2) They involve *layering violations* (e.g. both high-level information about predicted access patterns and low-level information about contiguity on disk) [23].

The lack of support for implementing path-specific customizations in a comprehensible fashion is a known disadvantage of layered architectures [5]. Problems arise because passing dynamic context is inherently messy, and leads to coupling as higher-level context passes through lower-levels. Layering violations also lead to coupling in fragments of code that operate on abstractions from multiple levels. In essence, path-specific customizations cut a vertical slice through the layers.

Recently, the aspect-oriented programming (AOP) [11] community has focused attention on the concept of *crosscutting concerns*, which are elements of a system that cut through the primary system modularity. They have proposed linguistic mechanisms intended to allow implementation of crosscutting concerns as first class modules called *aspects* [8, 16, 14, 1].

The goal of our work is to determine if the mechanisms of AOP can be used to improve the modularity of OS code. Specifically, we want to determine whether path-specific customizations can be considered to be crosscutting concerns, and whether they can be modularized using the mechanisms of AOP.

Most AOP language research is in Java. To enable a range of experiments for operating systems written in C, we developed a paper design for *AspectC*. Conceptually and in syntax, AspectC is a simple subset of AspectJ [8, 10]. As an initial experiment, we used AspectC to modularize the implementation of prefetching within page fault handling in the FreeBSD OS kernel.

Our methodology was to start by refactoring existing code using AspectC, and hand-compiling the code to C. We are comfortable with the code we have developed for two reasons. First, our design for AspectC is based on AspectJ, so we are confident that it can be implemented. Second, hand compilation is straightforward (albeit boring) which makes us confident that our refactored code is correct.

We begin with a description of the type of crosscutting concerns we are focusing on, path-specific customizations, and specifically consider prefetching in page fault handling as a concrete example. Section 3 shows how we have used AspectC to modularize two path-specific prefetching customizations. Section 4 presents an analysis of the implementation. Section 5 presents future work and discusses open issues. Section 6 reviews related approaches, and Section 7 summarizes our results and future work.

2. A REPRESENTATIVE EXAMPLE – PREFETCHING IN FREEBSD

Prefetching is a critical element of all operating systems. It is a performance optimization that aims to amortize the cost of fetching data from the disk by retrieving additional data with each disk request. Prefetching is based on combining predictions about what additional data is likely to be needed in the future with a analysis of what additional data would be most cost-effective to fetch at any given time.

Prefetching is a classic path-specific optimization. Dynamic context information is required because the lower levels of the page fault mechanism need to know where the fault came from in order to predict future demands. Layering violations are inherent in the combining of predictions and cost analysis.

This paper focuses on two particular aspects of prefetching in the FreeBSD 3.3 operating system. Both have to do with prefetching during a page fault to mapped files. The first handles the case where the declared access pattern is normal, the second is for declared sequential access.

The next section describes the relevant mapped file functionality as it would be with no prefetching. This is followed by a general description of prefetching, and a description of the two specific prefetching modes.

Sections 2.1 and 2.2 discuss the required prefetching functionality independent of any particular code that implements it. Discussion of the original implementation is deferred to Section 2.3.

2.1 The virtual memory abstraction

FreeBSD and other Unix operating systems allow the programmer to map any file into virtual memory. Such a mapped file is called a VM object, and can be accessed as ordinary virtual memory. When a file is mapped, it is not entirely transfered from disk at that time. Instead it is brought into memory as needed, one page at a time.

A *page fault* occurs when a process references a virtual address that is not in physical memory. A page fault is basically an exception raised each time a non-resident virtual page is accessed. Over time, pages accessed in the mapped file are demand-paged into memory.

In the absence of prefetching, handling a page fault is fairly straightforward. It is well supported by a layered architecture in which the virtual memory system is a client of the file system, which is in turn a client of the disk system.

A page fault starts in the virtual memory (VM) system as a request for a page associated with a VM object; it moves through to the local file system, FFS in our case, and is translated into a block-based request associated with a file; it finally passes to the disk system where it is expressed in terms of a cylinder, head, and sectors. The division of responsibilities between these three layers is centered around the management of their respective representations of data. That is, the functionality within each layer primarily deals with controlling resources in terms of its own set of abstractions.

2.2 Prefetching

This paper is focused on two path-specific prefetching customizations, both having to do with virtual memory mapped files. But it is important to note that prefetching in the OS kernel is more extensive. Essentially all execution paths that lead to the disk or the network have some form of path-specific prefetching associated with them. (We have already refactored two more prefetching aspects and have identified three more which we intend to implement once we have a running AspectC compiler.)

Prefetching involves four key activities: prediction, cost analysis, planning and actually fetching the pages. (Note that this discussion of prefetching is intended to enable the non-OS expert reader to understand the results of this refactoring experiment, not to serve as a detailed survey of prefetching literature.)

At a high-level, context at the origin of the request is used to predict future requests. The virtual memory system, the local file system, and the remote file system all use different criteria to make this prediction. For example, the file system might determine that a file is being accessed sequentially, and predict future requests on that basis.

Cost analysis involves lower level factors such as the cost of disk access, contiguity of data on disk, and the destination of the data. These are used to determine which disk blocks can be prefetched in the most cost-effective way.

Planning involves combining prediction and cost analysis information to determine which data should actually be fetched from disk, and whether that should happen as a synchronous part of the current read, or asynchronously.

Fetching the data normally just involves executing the plans, but there are cases where disk or other system state may change between planning and fetching, which can cause plans to be changed or cancelled.

Figure 1 provides a simplified overview of the primary func-

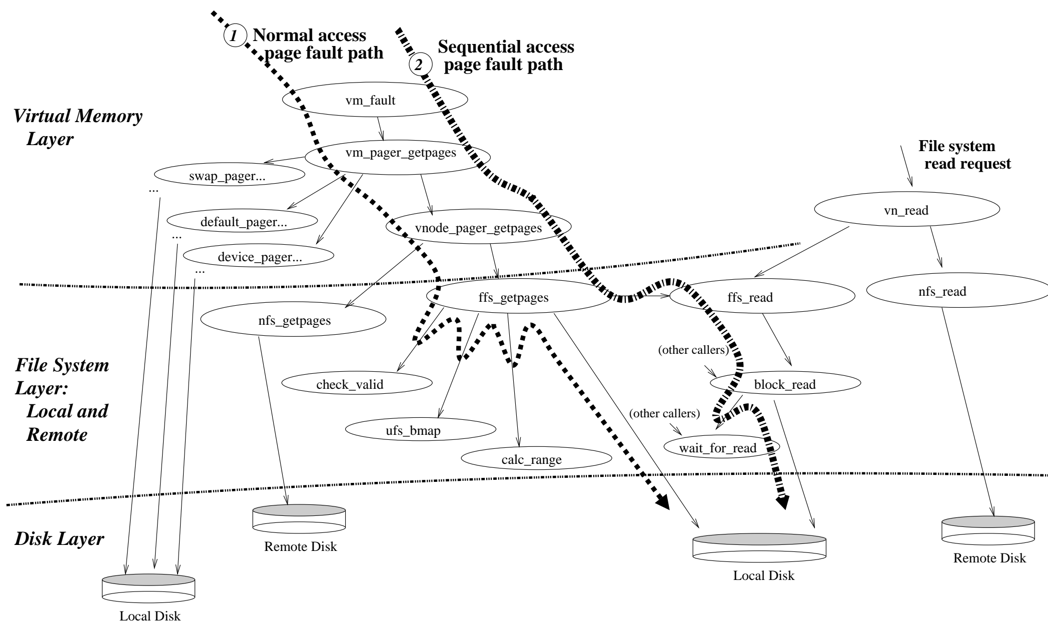

Figure 1: The layered kernel architecture and the structure of two path-specific optimizations. The ovals represent functions in the primary code for the virtual memory system, file system and disk system layers. The path-specific optimizations are numbered: (1) normal access mode page fault and (2) sequential access mode page fault.

tionality of the kernel layers involved in execution paths going to disk: the virtual memory system, the local and remote file systems, and the disk system. The execution paths discussed in the paper are shown as numbered lines: (1) is for normally accessed files, and (2) is for sequentially accessed files.

2.2.1 Prefetching for normal access (path 1)

VM objects have a declared access behaviour, which can be set to *random*, *normal* or *sequential* using the *madvise* system call.

For VM objects with normal behavior, prediction follows a simple locality heuristic that addresses close to the faulted address (+/- a specific window) are more likely to be used next. Cost analysis looks at both available memory and contiguity on disk.

In planning normal mode prefetching the cost analysis factors are given more weight than the prediction factors. Prefetching is synchronous and pages that are not contiguous are not prefetched even if they appear within the predicted window.[1]

As part of planning, physical pages are allocated to hold the pages to be prefetched. Because this allocation requires locking the page map associated with the VM object, and because that page map is already locked in the VM layer as part of preparing to fetch the faulted page, it is advantageous to do planning while execution is still in the VM layer.

Cost analysis inherently involves dynamic context passing and layer violations, since it looks at both VM layer information (available memory) and disk layer information (contiguity). Combining cost analysis with prediction is another source of layer violations.

By the time the normal mode execution path reaches the file system layer, important system state may have changed in a way that invalidates the prefetching plan. There are three conditions under which the file system layer will choose not to prefetch planned pages for normal objects: the faulted page has become valid, the faulted page is no longer on disk, or the planned pages are no longer contiguous.

The file system layer must de-allocate memory for virtual pages it decides not to fetch. This is an additional source of context passing and layer violations, since the file system layer must access the VM page map.

2.2.2 Prefetching for sequential access (path 2)

For VM objects with sequential behavior, prediction simply says that future accesses will directly follow the current access.

In sequential mode prefetching, planning allows prediction information to dominate cost analysis – predicted pages are prefetched even if they are not contiguous on disk. Some of the prefetching is done asynchronously.

This aggressive sequential prefetching is handled by redirecting control flow through *ffs_read* instead of *ffs_getpages*. This path triggers yet another prefetching mechanism specific to the file system read service[2], which asynchronously prefetches ac-

[1] Intuitively this is because the prediction in normal mode access is not strong enough to warrant the risk of additional disk waits.

[2] We have implemented this as a separate aspect, not included here.

cording to a sequential access pattern.

The path-specific customization required in this case involves the final destination of the read from disk to be associated with the pages allocated to the VM object. A page-aligned transfer from the file buffer cache to the VM allocated pages is not part of a typical file system read operation. This ‘page flipping’ avoids an expensive copy operation and is associated only with this particular execution path.

2.3 The original implementation

In the original implementation, code for prefetching for mapped files is *scattered* over approximately 265 lines, grouped into 10 contiguous blocks, in 5 functions from three layers. In other words, it is poorly modularized.

Dynamic context passing makes the code *tangled* as parameters from high level functions are passed down through lower ones. Layering violations further tangles the code in places where one segment of code uses both VM and FS layer abstractions.

The net effect is that it is extremely difficult to understand the structure and behavior of prefetching in the original implementation. Even just identifying all the prefetching code takes a significant amount of work. Understanding how the code works is difficult because it is poorly localized, and its relation to the execution flow of the main code is hard to follow. In fact, in our study of the original implementation a significant amount of work was required before we were able to conceptually separate normal mode prefetching from sequential mode prefetching.[3]

Based on our analysis, it appears that the natural modularity of prefetching modes is that of a single execution path, rather than of the layers in the system. But these execution paths crosscut the layers, as shown in Figure 1. This crosscutting property of the prefetching modes appears to be the reason they are difficult to modularize using traditional techniques, and is the basis of our decision to explore whether aspect-oriented programming can improve the modularity of this code.

3. ASPECTC IMPLEMENTATION

The aspect-oriented implementation of prefetching presented here uses AspectC – a simple AOP extension to C. These extensions support modular implementation of crosscutting concerns by allowing fragments of code that would otherwise be spread across several functions to be co-located and to share context.

Our implementation should be considered as a refactoring using AspectC [4]. Overall, only a small portion of our implementation of prefetching relies on the AspectC extensions, the rest is ordinary C code from the original implementation.

[3]Our inspiration to explore an AOP approach to prefetching is largely due to observing an experienced systems programmer in our lab devote several days to tracking down all the sources of page allocation and de-allocation in the code.

AspectC is a simple subset of AspectJ [8]. Aspect code, known as advice, interacts with primary functionality at function call boundaries and can run before, after or around the call. The central elements of the language are a means for designating particular function calls, for accessing parameters of those calls, and for attaching advice to those calls. Key to structuring the crosscutting implementation of prefetching is the ability to capture dynamic execution context with the control flow, or *cflow*, language extension.

In this experiment, our primary goal was to evaluate whether AOP had the potential to improve the modularity of OS kernel code. To do this as quickly as possible, we initially deferred building an AspectC compiler. Instead, we wrote code in AspectC and hand-compiled it to native C. Since these results have proven to be quite promising we are now working on an AspectC compiler that will enable us to take the work farther.

The remainder of this section presents our AspectC implementation of normal and sequential mode prefetching. AspectC itself is presented on an as-needed basis.[4]

3.1 Normal prefetching in AspectC

Figure 2 illustrates the structure of prefetching for objects with normal access patterns. The corresponding aspect-oriented implementation is shown in Figure 3. The first two declarations have to do with making values from higher-levels of the page fault handling path available to prefetching code in lower-levels. The next four declarations correspond directly to the small circles in Figure 2.

The first declaration in Figure 3 allows advice in the aspect to access the page map in which prefetching pages must be allocated. This map is the first argument to *vm_fault*. Reading the declaration, it declares a *pointcut* named *vm_fault_cflow*, with one parameter, *map*. A *pointcut* identifies a collection of function calls and arguments to those calls. The second line of this declaration provides the details. This pointcut refers to all function calls within the control flow of calls to *vm_fault*, and picks out *vm_fault*'s first argument. The ‘..’ in this parameter list means that although there are more parameters in this list, they are not picked out by this pointcut.

The second declaration is another pointcut, this time named *ffs_getpages_cflow*, which allows advice in the aspect to access the parameter list of *ffs_getpages* for de-allocation of planned pages.

The third declaration defines before advice that examines the object's declared behaviour, plans what virtual pages to prefetch, and allocates physical pages accordingly. In plain English, the header says to execute the body of this advice before calls to *vnode_pager_getpages*, and to give the body access to the *map* parameter of the surrounding call to *vm_fault*.

Reading the header in more detail, the first line says that this advice will run *before* function calls designated following the

[4]For a more detailed understanding of AspectC consult [8, 9]. AspectC can be understood as a subset of AspectJ in which C functions are analogous to static methods in Java, all aspects are singletons, there are only method call join points, and there is no introduction.

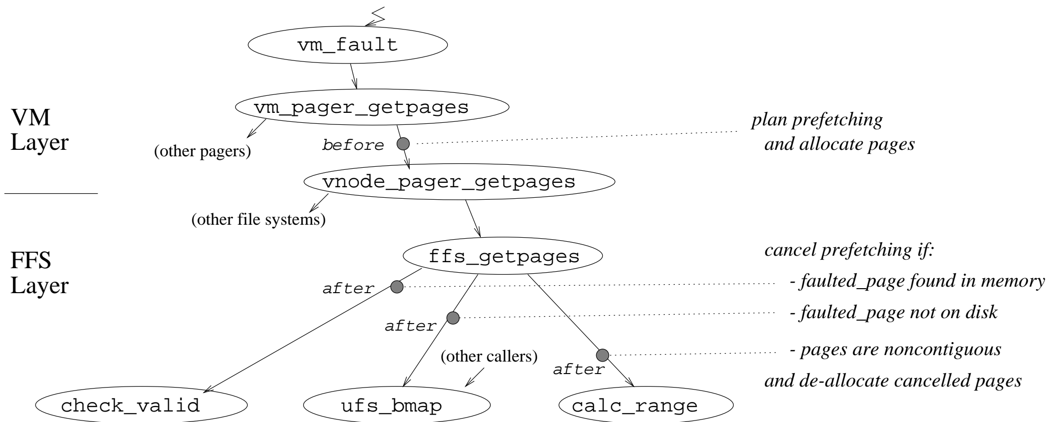

Figure 2: The structure of the AspectC implementation of normal mode prefetching. Only the top two layers, VM and FFS, are shown. The ovals represent functions comprising the primary page fault handling structure, the small circles and text in italics represent the structure of the prefetching aspect.

```
aspect normal_mapped_file_prefetching {

  pointcut vm_fault_cflow( vm_map_t map ):
      cflow( calls( int vm_fault( map, .. )));

  pointcut ffs_getpages_cflow( vm_object_t object, vm_page_t* pagelist, int* length, int faulted_page ):
      cflow( calls( int ffs_getpages( object, pagelist, length, faulted_page )));

  /* plan the prefetching and allocate the pages */
  before( vm_map_t map, vm_object_t object, vm_page_t* pagelist, int* length, int faulted_page ):
      calls( int vnode_pager_getpages( object, pagelist, length, faulted_page )) &&
      vm_fault_cflow( map )

  {
    if ( object->declared_behaviour == NORMAL ) {
      vm_map_lock( map );
      plan_and_alloc_normal_prefetch_pages( object, pagelist, length, faulted_page );
      vm_map_unlock( map );
    }
  }

  /* three cases in which prefetching might be cancelled for normal objects */

  after( vm_object_t object, vm_page_t* pagelist, int* length, int faulted_page, int valid ):
      calls( valid check_valid(..) )  &&
      ffs_getpages_cflow( object, pagelist, length, faulted_page )
  {
    if ( valid )
      dealloc_all_prefetch_pages( object, pagelist, length, faulted_page );
  }

  after( vm_object_t object, vm_page_t* pagelist, int* length, int faulted_page, int error, int* reqblkno ):
      calls( error ufs_bmap( struct vnode*, reqblkno, ..) )  &&
      ffs_getpages_cflow( object, pagelist, length, faulted_page )
  {
    if ( error || (*reqblkno == -1) )
      dealloc_all_prefetch_pages( object, pagelist, length, faulted_page );
  }

  after( vm_object_t object, vm_page_t* pagelist, int* length, int faulted_page, struct trans_args* t_args ):
      calls( int calc_range( t_args ))  &&
      ffs_getpages_cflow( object, pagelist, length, faulted_page )
  {
    dealloc_noncontig_prefetch_pages( object, pagelist, length, faulted_page, t_args );
  }
}
```

Figure 3: AspectC code for prefetching pages for objects of normal behaviour.

':', and lists five parameters available in the body of the advice. The second line specifies calls to the function *vnode_pager_getpages*, and picks up the four arguments to that function. The third line uses the previously declared pointcut *vm_fault_cflow*, to provide the value for *map* that is associated with the particular fault currently being serviced (i.e., from a few frames back on the stack).

The body is ordinary C code. The helper function *plan_and_alloc_normal_prefetch_pages* further determines how many and which pages to allocate, depending on the availability of memory and layout of the pages on disk.

The next three declarations implement the three conditions under which the FFS layer can choose not to prefetch. In each case, the implementation of the decision not to prefetch results in de-allocation.

The first after advice de-allocates all pages to be prefetched if the faulted page is now valid. This executes after calls to *check_valid*, which occur when the normal page fault path is checking to see whether the page has become valid. When *check_valid* returns non-zero, it is telling the normal paging code that the page is now present in memory. In this case, prefetching advice cancels all the prefetching.

The second after advice de-allocates all prefetching pages if the faulted page is not found on disk. This may happen for one of two reasons – either an error has occurred in which case *error* is non-zero, or the fault will instead be satisfied by a zero-filled page, in which case the parameter *reqblkno* from *ufs_bmap* is -1. It is important to note that the use of *ffs_getpages_cflow* not only makes parameters available to advice that executes after calls to *ufs_bmap*, but also ensures that this advice only executes within this control flow. That is, calls to *ufs_bmap* in other paths do not execute this advice.

The third after advice de-allocates some or all prefetching pages if the contiguity of the pages on disk has changed since being checked by *plan_and_alloc_normal_prefetch_pages* in the VM-layer. The helper function takes all the parameters from *ffs_getpages_cflow* and *calc_range*, and de-allocates any pages that were originally requested but not within the actual range that will be fetched.

3.2 Sequential prefetching in AspectC

Figure 4 illustrates the structure of prefetching for objects with sequential access patterns. The corresponding implementation is shown in Figure 5. The careful reader will notice a small amount of code duplication with the previous aspect. In particular, the *vm_fault_cflow* pointcut is in both aspects. This is deliberate for clarity. AspectC includes features that would allow us to eliminate this duplication, and thereby clearly reflect the common functionality of the two aspects. Specifically, a shared aspect can be used to define common elements.

Similar to normal mode prefetching, the two pointcuts use cflow to make values from higher-levels of the page fault handling path available to prefetching code in lower-levels. Note that this before advice operates independently of the before advice in the aspect for normal mode prefetching, even though they both advise the same function.

This aspect uses *around* advice to divert the execution path to *ffs_read* when access is sequential, or to *proceed* with *ffs_getpages* otherwise. Around advice differs from before and after advice in that it has control over whether or not the advised function call proceeds as planned.

Looking closely at the parameters to the call to *ffs_read* in this advice, the constant *MAXBSIZE* is used to dictate the size of the synchronous read request. This indicates that the amount of data to be synchronously fetched will be the maximum buffer size, regardless of the layout on disk. Consequently, unlike normal mode prefetching, prediction in this case dominates cost analysis.

The after advice executes under the control flow of the pointcuts *ffs_read_cflow* and *vm_fault_cflow*. That is, it executes only when control flow has been diverted along this special path.

This after advice is responsible for ensuring that additional costs are not incurred when transferring the data from the buffer cache to the allocated VM pages. Since this is a special case page-aligned transfer, copying can be avoided by simply reassigning or 'flipping' allocated VM pages with the appropriate file buffer cache pages.

4. ANALYSIS

This section analyzes the AspectC implementation in terms of the benefits traditionally associated with modular programming [19, 24]. Where appropriate we also compare the new and original implementations.

4.1 Pluggable functionality

In the AspectC implementation, the code for each prefetching mode is textually localized in a single aspect. This enables plug and play prefetching modes. We can place each aspect in a single file, and use standard makefile techniques to include or exclude specific combinations of prefetching modes.

We have verified this by compiling and running the kernel in four configurations: no prefetching, normal mode prefetching only, sequential mode prefetching only, and both prefetching modes. (Not surprisingly, things ran much more slowly without prefetching.)

With the simple implementation of AspectC we are currently using, our ability to do prefetching aspect (un)plugging is limited to compile-time selection. Even so, this is significantly more plug and play control over such a deeply crosscutting concern than has previously been possible in operating systems. As suggested in [6], it also appears this control could be sufficient to support certain product-line architectures, but more work is required to confirm this.

In the non-aspect implementation the work required to put each prefetching mode on a switch would be extensive – 10 clusters of prefetching code from 5 files would have to be edited to use use compiler directives (#ifdef).

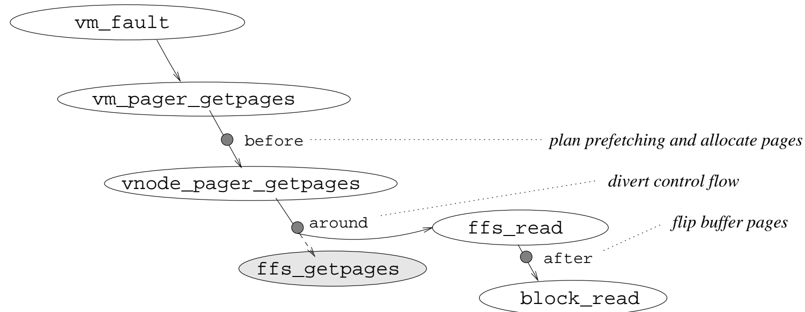

Figure 4: The structure of the AspectC implementation of sequential mode prefetching.

```
aspect sequential_mapped_file_prefetching {

  pointcut vm_fault_cflow( vm_map_t map ):
      cflow( calls( int vm_fault( map, .. )));

  pointcut ffs_read_cflow( struct vnode* vp, struct uio* io_info, int size, struct buff** bpp ):
      cflow( calls( int ffs_read( vp, io_info, size, bpp )));

  /* plan the prefetching and allocate the pages */
  before( vm_map_t map, vm_object_t object, vm_page_t* pagelist, int* length, int faulted_page ):
      calls( int vnode_pager_getpages( object, pagelist, length, faulted_page ))  &&
      vm_fault_cflow( map )

  {
    if ( object->declared_behaviour == SEQUENTIAL ) {
      vm_map_lock( map );
      plan_and_alloc_sequential_prefetch_pages( object, pagelist, length, faulted_page );
      vm_map_unlock( map );
    }
  }

  /* divert to ffs_read */
  around( vm_object_t object, vm_page_t* pagelist, int* length, int faulted_page ):
      calls( int ffs_getpages( object, pagelist, length, faulted_page ))

  {
    if ( object->behaviour == SEQUENTIAL ) {
      struct vnode* vp = object->handle;
      struct uio* io_info = io_prep( pagelist[faulted_page]->pindex, MAXBSIZE, curproc );
      int error = ffs_read( vp, io_info, MAXBSIZE, curproc->p_ucred );
      return cleanup_after_read( error, object, pagelist, length, faulted_page );
    } else
      proceed;
  }

  /* page flip buffer pages */
  after( struct uio* io_info, int size, struct buf** bpp ):
      calls( int block_read(..) )  &&
      vm_fault_cflow(..)           &&
      ffs_read_cflow( struct vnode*, io_info, size, bpp )

  {
    flip_buffer_pages_to_allocated_vm_pages( (char *)bpp->b_data, size, io_info );
  }
}
```

Figure 5: AspectC code for prefetching on behalf of sequentially accessed memory mapped files.

4.2 Independent development

The interface between the prefetching aspects is clear. We can easily tell what functions in the main page fault handling code the prefetching aspect knows about, and what arguments to those functions it sees.

When working with the aspect, the interface is apparent from a quick reading of the code. When working with the main code, simple editor extensions, such as discussed in [10], can flag functions that are the targets of advice with links back to the aspect.

The interface is also relatively abstract, representing an abstraction of the internal structure of page fault handling rather than true details of the page fault handling code. The interface is similar in nature to those presented by object-oriented frameworks [7].

Because the interface is clear and abstract, it is possible to develop the main code and each of the aspects quite independently. Of course, as with any abstract interface, there are some kinds of changes that will require changing the interface, and all of the code that depends on it.

In the original implementation, the code for these two prefetching modes is so scattered and tangled through the main page fault handling code that the question of doing independent development hardly even makes sense to ask. In the best case scenario the developer would know from prior experience exactly which functions contained this code and be able to start there. But even so there are roughly 265 lines of prefetching code distributed over 5 functions that contain a total 1950 lines. (125 out of 825 lines in the VM layer, 120/250 in the FFS layer, and the 20/875 in the disk layer.)

4.3 Comprehensibility

Decomposing page fault handling into the main page fault handling functionality and prefetching aspects allows us to reason about the different parts and their respective interaction separately.

The behavior of page fault handling with several different prefetching modes is still complex. But the ability to reason about it as a combination of different modules materially increases comprehensibility relative to the original implementation.

4.3.1 Aspect interaction with rest of code

The interaction between prefetching and rest of code is declarative. Advice declarations and pointcuts tell us when advice runs, what values it sees, and what effect it can have on the execution of the rest of the code.

For example, we know that the first after advice in Figure 3, runs after check_valid, ignores the parameters, depends on the return value but cannot change it, and has access to arguments to the surrounding call to *ffs_getpages*. We also know that the before advice from the two aspects operate independently, even though they advise the same function.

Making the interaction declarative means that we can reason about it at an abstract level. Understanding that *vm_fault_cflow* makes the first argument to *vm_fault* available to other advice within the aspect is easier than understanding traditional code that passes dynamic context down through layers of function calls.

Declarative aspect interaction also gives us guarantees about value flow and execution that are not available in the original code, i.e. it is easy to find every piece of code that has access to the map, whereas in the traditional context passing approach that is harder to be sure of.

4.3.2 Aspect internal structure

Because each prefetching mode is localized, it is easier to understand its internal structure. Within the aspect and it helper functions, we can see interactions such as the planning and allocation of prefetched pages and the subsequent checking and de-allocation of those pages.

This localization makes it easier to reason about and change the aspect's internal behavior. In the original code, this is more complex. For example, in order to see the coordination between allocation and de-allocation for normal mode prefetching, we would have to trace the execution path through 5 files, 2 levels of function tables handled by macros, and 4 changes in variable names.

5. FUTURE WORK AND DISCUSSION

Our work targets the understanding and modularization of path-specific customizations. Other examples of this kind of crosscutting we plan to explore in kernel code involve page replacement and scheduling strategies.

Page replacement is the process of evicting pages that are currently resident in memory in order to make room for new pages. A page replacement strategy is invoked any time the system is low on available memory. In the general case, the criteria for page eviction is based on a least-recently used policy. Of course, this is exactly the wrong policy to apply in the case of sequential access, where the most recently used page is actually the best candidate for eviction.

As with prefetching, high-level context important for page replacement can be explicitly set using *madvise*. This allows eviction to use the pattern of access as part of the selection criteria for removal. Low-level context associated with writing pages out to disk hinge on the ability to cluster writes in a way that will support contiguity for subsequent reads. Similar to prefetching, layer violations occur because the decision of which pages to evict are typically made at a low level but require interacting with higher level abstractions.

Scheduling involves sharing the processor between all active processes. A scheduling policy tries to ensure that all currently executing processes make progress. Time slicing, priority levels, and points in the execution where a process is naturally blocked waiting for disk I/O, are all used to determine which process will get the processor next.

The path-specific customizations we are exploring with regard to scheduling are related to high-level process state and low-

level blocking I/O requests. Layer violations in this case stem from the need to reconcile cross-layer information such as access patterns, process priority, and disk requests in order to make good scheduling decisions.

5.1 Open issues

Essential open questions regarding path-specific customization, AspectC and aspect-oriented programming in general include issues of efficiency, scalability, and development tools. In terms of efficiency, improving modularity of OS kernel code is not helpful if it adversely impacts performance. Specifically, we need to know what overheads AspectC adds to code in terms relative to a tangled implementation. We are now implementing an AspectC compiler as a simple pre-processor. Experience with AspectJ, as well as our own hand-compiling of the code indicates that this kind of implementation can evolve to produce the performance characteristics we need.

Another issue, currently under investigation in the AOP community in general, is scalability. With respect to our work, a possible criticism is that as we introduce more aspects to the kernel, we introduce more interfaces, more interaction and more complexity. Without principled application, the possibility of degrading comprehensibility exists. Although we need more experience to comment concretely on heuristics for creating and managing sophisticated, multi-aspect structures, we are optimistic that our future work will provide insight into this issue. Since our technique makes code for two prefetching modes more comprehensible, it would be surprising if it made code for a large number of aspects in the kernel more complex. But this is something we will have to explore, in particular with respect to understanding interaction between aspects.

Tools are another area under investigation by several groups in the AOP community. The impact aspect-oriented programming will have on the development process, the support required to facilitate its use, and the metrics used to determine degrees of crosscutting, scattering and tangling are all issues that require attention. Once we have more experience with the use of AspectC in kernel code, we expect to develop additional tool support, including debugging support, following the same basic path as AspectJ.

6. RELATED WORK

Work on modularizing path-specific customization touches on work from systems, separation of concerns, and programming language communities.

6.1 Operating system structure

The advantages of a layered architecture has been recognized as key since the THE multiprogramming system [2] in the late 60s. End-to-end arguments [22, 21] provide a set of principles for determining the placement of functions within layered designs. These principles advocate an organization where a function or service belongs in a layer only if it can be completely implemented in that layer and is needed by all clients.

An aspect-oriented approach to structure in an operating system is compatible with, and incrementally applicable to, a layered architecture. Separating the implementation of these customizations from the primary functionality may better support end-to-end arguments by allowing principled vertical aspects to capture customizations in a layered system.

The Synthetix project [20] uses specialization to optimize commonly used paths in the system. Specialization uses incremental partial evaluation, largely consisting of constant folding and macro expansion, to generate multiple path-optimized implementations for the same interface. Other related Synthetix projects use specialization for survivability, end-to-end quality, and adaptability.

Customization of a specific execution path is central to both our application of aspect-oriented programming and specialization. Our approach advocates the separation of path-specific customization by the programmer in the original source code to better achieve comprehensibility. Specialization aims to automatically specialize the original source code.

An issue of great importance within operating systems is the untangling or streamlining of data flow in order to improve performance. Scout [15] is an operating system designed to optimize communication by specifying a fast data path to move priority data (such as video streams) through the system with as little overhead as possible.

Although the nature of this optimization in Scout may lend itself to some form of path-specific customization, our intuition is that mechanisms to support data flow will be different from those we have used for control flow. The role AspectC can play in data flow is an area of crosscutting we plan to explore.

6.2 Separation of concerns

Our work stems directly from the approach to separation of concerns (SOC) supported by the language extensions developed by the AspectJ project [8]. Specifically, we are currently applying this linguistic support to one kind of crosscutting concern: path-specific customization.

Separation of concerns requires some criteria for decomposition. Parnas suggested decomposition should begin with a list of either difficult design decisions or design decisions that are likely to change, and those decisions should be hidden into modules first [19]. Stevens et al. later suggested that functional binding, or cohesion based upon the execution of a single task, produces less complex interaction between modules relative to weaker bindings such as temporal execution or the referencing of common data [24].

In our work we are advocating a modularity where primary functionality can be implemented by a traditional means, and the crosscutting path-specific customizations are implemented as aspects. We believe this separation achieves the qualities associated with good modularity better than the scattered and tangled implementation operating systems are currently faced with.

A number of general approaches to separation of concerns in complex systems have emerged in the last few years. Work on subject-oriented programming [17] and hyperspaces [16]

is aimed at composing hierarchies of concerns and focuses on multiple dimensions of concerns. Composition filters [1] separate objects into internal parts and interfaces to which filters can be applied.

Although all of these approaches involve explicit separation at the source code level in order to increase comprehensibility, our work hinges directly on the ability to specify dynamic execution context in order to modularize crosscutting concerns.

6.3 Programming language support

Several programming languages provide access to dynamic context. Perl [27] and Tcl [18] allow access to the call stack at run-time. Explicit support for access in the form of dynamic scoping is provided by languages such as Lisp which allow variable names to be bound according to the state of the call stack.

Access to dynamic context for path-specific customization requires specific linguistic support for principled call stack access. Perl and Tcl do not have a general mechanism for accessing specific parameters in a principled way. Dynamic scoping may be important to support, but at this stage in development it is not part of AspectC. Further experimentation is required to know the pros and cons of supporting this feature in systems code.

6.4 Other work

Implicit context [26] targets the removal of extraneous embedded knowledge (EEK) to improve separation of concerns and support software evolution and reuse. This approach provides reflective access to the call history of the system. Another research project, Implicit parameters [13], allows a set of intervening functions to be excluded from the parameter passing between two endpoints. A new parameter can thus be passed directly from a sender to a receiver without changing the source code for functions that execute between them.

Our current experience with path-specific customization has more modest needs with regard to dynamic context than the complete call history provided by implicit context. The performance considerations of the kernel may preclude attempts to maintain quite this much history, but it may be useful to write aspects that gather some subset of call history information beyond what we are currently using. Implicit parameters have some of the power of using cflow to pass dynamic context. The difference is that the source must be written to explicitly use the implicit parameter mechanism. This does not support separation concerns to the degree we can achieve with AspectC.

7. CONCLUSION

Operating systems have a problem with modularity. Part of the problem is that general low-level services are commonly tailored to different high-level contexts within which it they are invoked. We refer to this tailoring as path-specific customization, and identify dynamic context information and layering violations as the properties that make it hard to modularize.

In this paper, we show preliminary results of how an aspect-oriented refactoring of two path-specific customizations associated with prefetching for mapped files improves modularity. Our results show that the AspectC implementation presented here supports unpluggability, independent development and comprehensibility better than the tangled implementation.

Our work to date has focussed on evaluating the potential for AspectC to improve the modularity of OS kernel code. We are currently working to implement AspectC and plan to use it to explore other kinds of crosscutting concerns common to operating system implementations.

Acknowledgments: Many thanks to Stephan Gudmundson, Jan Hannemann, Gail Murphy and Andrew Warfield for their insightful comments on drafts of this paper. Many thanks also to Ida Chan, whose tenacious approach to kernel code ensured that our perfetching aspects were sound, and to Christopher Dutchyn for his invaluable contributions to the AspectC prototype.

8. REFERENCES

[1] M. Aksit and B. Tekinerdogan. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. In *OOPSLA AOP'98 workshop position paper*, 1998.

[2] E.W. Dijkstra. The structure of THE-multiprogramming system. *Communications of the ACM*, 11(5), 1968.

[3] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[4] Martin Fowler. Addison-Wesley Object Technology Series, 1999.

[5] David Garlan and Mary Shaw. An introduction to software architecture. Technical report, CMU, 1994. CS-94-166.

[6] Martin Griss. Implementing product-line features by composing component aspects. In *Proceedings of First International Software Product Line Conference*, August 2000.

[7] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1, June 1998.

[8] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. AspectJ home page. www.aspectj.org.

[9] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. AspectJ primer. www.aspectj.org/doc/primer/index.html.

[10] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An

overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, June 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] L.L. Lehman and L.A. Belady. Program evolution. *APIC Studies in Data Processing*, (27), 1985.

[13] J. Lewis, M. Shields, E. Meijer, and J. Launchbury. Implicit parameters: Dynamic scoping with static types. In *Symposium on Principles of Programming Languages*, January 2000.

[14] K.J. Lieberherr. *Adaptive Object-Oriented Software: the Demeter Method with Propagation Patterns*. Boston: PWS Publishing Company, 1996.

[15] Allen B Montz, David Mosberger, Sean W. O'Malley, Larry L. Peterson, Todd A. Proebstign, and John H. Hartman. Scout: A communications-oriented operating system. Technical report, University of Arizona, 1994. TR 94-20.

[16] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the Hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2000.

[17] Harold Ossher, William Harrison, Frank Budinsky, and Ian Simmonds. Subject-oriented programming: Supporting decentralized development of objects. In *Proceedings of the 7th IBM Conference on Object-Oriented Technology*, 1994.

[18] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[19] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1972.

[20] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, 1995.

[21] D.P. Reed, J.H. Saltzer, and D.D. Clark. Active networking and end-to-end arguments. In *IEEE Network*, June 1998.

[22] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. In *ACM Transactions on Computer Systems (TOCS)*, November 1984.

[23] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[24] W.P. Stevens, G.J. Meyers, and L.L. Constantine. Structured design. *IBM Systems Journal*, 13, 1974.

[25] Werner Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP)*, 1999.

[26] Rob Walker and Gail Murphy. Implicit context: Easing software evolution and reuse. In *Proceedings of the Proceedings of the ACM SIGSOFT Eighth International Symposium on the Foundations of Software Engineering (FSE-8)*, November 2000.

[27] L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl*. O'Reilly and Associates, 2nd edition, 1996.