Exploring an Aspect-Oriented Approach to OS Code

Yvonne Coady, Gregor Kiczales, Michael Feeley, Norman Hutchinson,
Joon Suan Ong and Stephan Gudmundson
University of British Columbia

Operating system code is complex. But, while substantial complexity is inherent to this domain, we believe that other complexity is caused by modularity problems. In this paper, we explore aspect-oriented programming as a means of making this kind of complexity unnecessary. We show that simple linguistic constructs can be used to modularize prefetching – an aspect of the system that is otherwise unclear because its implementation is spread out in the code.

1 Introduction

Operating systems have a problem with modularity. Despite our best efforts, the implementation of certain key elements of the system seems to inevitably get spread out in the code. From OS/360 to Windows NT, systems suffer from "unintentional interactions" between modules [7] and require developers to be intimately familiar with implicit "patterns of interaction" between subsystems [10]. We believe this not only makes systems code more difficult to reason about and change, but also was a significant barrier to incremental customization [3, 5] in extensible systems research [4, 2, 8, 9].

Recently, the aspect-oriented programming (AOP) [6] community has put forth the idea that some concerns are inherently *crosscutting* – by their very nature they are present in more than one module. They call such concerns *aspects* of the system. The goal of work in AOP is to make it possible to modularize the implementation of aspects by developing mechanisms that explicitly support crosscutting structure. Several applications have successfully used AOP to structure issues such as synchronization and performance optimization [1].

The purpose of our work is to determine if an aspect-oriented approach can improve the modularity of operating system code. We want to find out if our modularity problems are caused by crosscutting concerns, whether the proposed mechanisms of AOP can serve to modularize them, and whether AOP materially improves the code.

This paper describes an initial experiment using AOP to localize the implementation of a crosscutting concern – prefetching for mapped files in FreeBSD v3.3. Our initial results are promising. We are optimistic that AOP may be able to significantly improve OS modularity, and hopeful that this could support new work on OS structure, incremental customization, and extensibility.

2 Implementation overview

Before inspecting the aspect-oriented implementation of prefetching, the following subsections provide necessary context regarding the structure of the page fault handling path, prefetching for mapped files within this path, and the original structure of the code.

2.1 The page fault handling path

Referencing a page of a mapped FFS file that is not marked as resident in memory generates an exception. Handling this exception starts in the virtual memory system as a page fault associated with a VM object; execution moves to FFS and is translated into a block-based request associated with a file; and finally passes to the disk system where it is expressed in terms of a cylinder, head, and sectors. The division of responsibilities between these subsystems is centered around the management of their respective representations of data. That is, core functionality within each component primarily deals with controlling resources in terms of its own set of abstractions. Paths taken when fault handling in VM and FFS are illustrated by the large ovals labeled with function names in Figures 1 and 3.

2.2 Prefetching for mapped files

Figures 1 and 3 are also annotated with prefetching functionality. Access behaviour of VM objects can be declared as *random*, *normal* or *sequential* using the *madvise* system call. This declaration is used to plan which pages should be prefetched, subject to available memory and contiguity of pages on disk. Once the prefetching is planned, physical pages are allocated accordingly. Allocation requires appropriate VM-based synchronization, such as locking the page map.

Beyond the VM layer, the access behaviour of the VM object determines how execution proceeds. Normal objects use the path shown in Figure 1, while the path for sequential objects is shown Figure 3.

2.2.1 Normal access prefetching

By the time page fault handling reaches the file system, important system state may have changed. Normal ac-

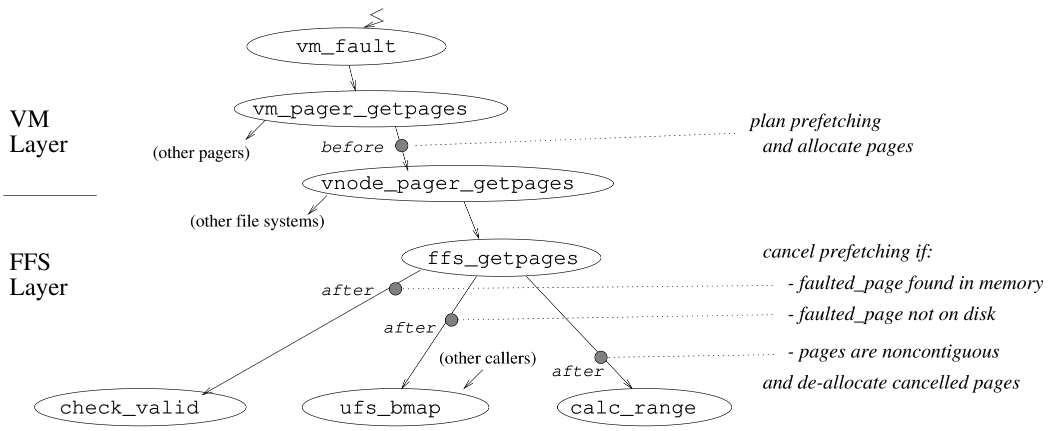

Figure 1: The structure of the page fault handling path for objects with behaviour declared to be normal. Only the top two layers, VM and FFS, are shown. The ovals represent functions comprising the primary page fault handling structure, the small circles and text in italics represent the structure of prefetching.

```
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 under 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 transfer_args* trans_args ):
      calls( int calc_range( trans_args ))
      && ffs_getpages_cflow( object, pagelist, length, faulted_page )
  {
    dealloc_noncontig_prefetch_pages( object, pagelist, length, faulted_page, trans_args );
  }
}
```

Figure 2: AspectC code for prefetching pages for objects of normal behaviour.

cess prefetching in FFS first determines cost effectiveness of retrieval according to current system state. Prefetched pages are synchronously retrieved with the faulted page, and consequently must not introduce multiple disk accesses.

There are three conditions under which FFS may choose not to prefetch planned pages for normal objects. First, the faulted page may be valid by the time execution reaches FFS, in this case none of the planned pages will be prefetched. Second, if the faulted page is not found on disk, no pages are prefetched and the page fault may be satisfied by a zero-filled page. Third, if any of the pages are no longer contiguous on disk, they are not prefetched. As part of checking whether it will request planned pages, FFS de-allocates pages it decides not to retrieve.

Figure 1 overviews the structure of prefetching for objects with normal declared access. The small circles and italicized text represent the elements of prefetching described above.

2.2.2 Sequential access prefetching

In order to aggressively prefetch on behalf of sequentially accessed mapped files, control flow is redirected through the file system read path using *ffs_read* instead of *ffs_getpages*. This path potentially offers additional asynchronous prefetching when access is sequential. The path also requires page flipping buffer pages to allocated VM pages in order to avoid an expensive copy operation. Figure 3 illustrates the structure of prefetching for objects with sequential declared access, which is described in more detail in Section 4.2.

2.3 Prefetching structure and the original code

When summarized as above and visualized as in Figure 1 and Figure 3, this prefetching structure is relatively clear. It is possible to reason about the coordination of activity between prefetching code in VM and FFS.

Unfortunately, in the original code this implementation is spread out over approximately 260 lines in 10 clusters of contiguous lines in 5 core functions from two subsystems – making it very difficult to see the coordination of prefetching activity. Moreover, there are clusters of code performing management tasks on VM abstractions sitting in FFS functions, which makes the code more confusing.

3 AspectC

In this experiment, we used a hypothetical language, AspectC, and hand-compiled the code to C. AspectC extends C by adding linguistic support for aspects. AspectC is a simple subset of AspectJ [1], so we are confident that it is possible build an efficient implementation, which is part of our present work. Overall, only a small portion of the code relies on these linguistic extensions. The majority of the code is in regular C functions.

AspectC provides mechanisms for defining additional code, called *advice*, that runs *before*, *after* or *around* existing function calls. The central elements of the language are means for designating particular function calls, for accessing parameters of those calls, and for attaching advice to those calls. These mechanisms are sufficient to modularize crosscutting concerns because they allow small fragments of code that would otherwise be spread across several functions to be placed right next to each other.

4 Implementation using AspectC

The following subsections present our re-implementation of prefetching for mapped files using AspectC. AspectC itself is presented incrementally on an ‘as-needed’ basis.

4.1 Normal prefetching in AspectC

Figure 2 shows our aspect-oriented implementation of prefetching for normal declared access. The first two declarations make 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 1.

The first declaration in Figure 2 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 deallocation 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 ‘:’, 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

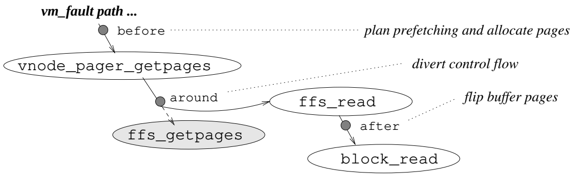

Figure 3: Execution path for objects with sequential declared access.

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 are no longer within the actual range that will be transferred.

4.2 Sequential prefetching in AspectC

Figure 3 illustrates the structure of synchronous prefetching for objects with declared sequential access. The corresponding implementation is shown in Figure 4.[1]

When *madvise* is used to declare access behaviour as sequential, prefetching in FreeBSD v3.3 is less conservative. The page fault request automatically is bumped to a a maximum buffer size, and the request is routed through *ffs_read* instead of *ffs_getpages*. This path not only synchronously fills the buffer, but possibly asynchronously prefetches additional blocks. Once the transfer is complete, the buffer pages are flipped in order to avoid copying to the pages VM allocated for prefetching. Asynchronous prefetching requires detection of sequential access in the file system using a marker stored in the *vnode*.[2]

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.

The after advice, which flips the pages, 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.

5 Implementation comparison

To develop the AOP implementation, we first stripped the prefetching related code from the primary implementation of page fault handling. We then we made several minor refactorings of the primary code structure to expose principled points for the definition of prefetching advice. With respect to Figure 1, we refactored *ffs_getpages* to spawn two new small functions, *check_valid* and *calc_range*.

The key difference between the original code and the AOP code is that when implemented using aspects, the coordination of VM and FFS prefetching activity becomes clear. We can see, in a single screenful, the interaction of planning and cancelling prefetching, and allocating and de-allocating or flipping pages. In short, we were able to preserve the context of prefetching related execution and make the structure of the crosscutting explicit.

6 Conclusion

AOP allowed us to modularize prefetching and make its structure explicit and clear. This experiment shows that some of the complexity in OS code is unnecessary because it comes from improper modularization techniques rather than being inherent to the domain. When crosscutting concerns are implemented without AOP they become tangled – spread throughout the code in an unclear way. When implemented with AOP, crosscutting structure is clear and tractable to work with.

[1] The careful reader will notice a small amount of code duplication with the previous aspect. This is deliberate for clarity. AspectC includes features that would allow us to eliminate this duplication.

[2] We implemented this in other aspects, not presented here.

```
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;
  }

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

We are currently working to implement AspectC and plan to use it to explore other crosscutting concerns in OS code. We believe that using aspects to localize the implementation of key OS elements such as prefetching, page replacement, quality-of-service requirements, and others, could enable significant improvements in OS structure, including to the ability to better support extensibility.

References

[1] http://www.aspectj.org.

[2] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, 1996.

[3] Peter Druschel. Efficient support for incremental customization of OS services. In *Proceedings of the Third International Workshop on Object Orientation in Operating Systems*, December 1993.

[4] Peter Druschel, Larry L. Peterson, and Norman C. Hutchinson. Beyond microkernel design: Decoupling modularity and protection in Lipto. In *Proceedings of the Twelfth International Conference on Distributed Computing Systems*, June 1992.

[5] Gregor Kiczales, John Lamping, Chris Maeda, David Keppel, and Dylan McNamee. The need for customizable operating systems. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, 1993.

[6] 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.

[7] L.L. Lehman and L.A. Belady. Program evolution. *APIC Studies in Data Processing*, (27), 1985.

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

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

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