# Exploring an Aspect-Oriented Approach to Operating System Code

Yvonne Coady, Gregor Kiczales, and Michael Feeley

*Department of Computer Science*
*University of British Columbia*

September 2, 2000

## Abstract

*This paper presents the initial results of our experiment using an aspect-oriented approach to simplify operating system code. The intuition behind this work is that complexity comes from crosscutting concerns.*

*In this experiment, we re-implemented prefetching for page fault handling and file system read requests using a hypothetical language, AspectC – a variant of AspectJ for C – and hand-compiled the code to C. We believe that the aspect-oriented code is easier to understand, and in particular that it is easier to understand the interaction between the virtual memory and file system prefetching mechanisms.*

## 1 Introduction

Operating systems have a problem with modularity. Complex interactions between components of the system blur module boundaries and make them highly susceptible to erosion. A study of OS/360 done in the early 70s [7] showed that the number of modules involved in a change rose from 14.6% in releases 2–6, to 31.9% in releases 12–16 due to "unintentional interaction" among components.

This situation does not appear to be improving. Modern commercial operating systems, like Windows NT for example, require third party file system designers to be intimately familiar with "patterns of interaction" that necessarily exist between the file system, cache manager and virtual memory manager [11]. Research in extensible systems [3, 9, 10] has failed to facilitate a key feature, incremental customization [4, 5], for the same reason. That is, it is not possible to ensure that a change will require an effort proportional to the amount of code involved when key concerns lack locality.

In our experiment we factored out and re-implemented the functionality of a crosscutting concern in an operating system. We used a hypothetical language, *AspectC* – a variant of AspectJ [1] for C. We then hand-compiled this implementation into C in order to get a running system. We believe that an aspect-oriented approach allowed us to bring important context to what had been complex interactions in the original implementation, making them easier to understand.

## 2 Prefetching: background and experiment

Operating systems use *prefetching* to amortize the costs of accessing disk. As a result, prefetching functionality accompanies activity from all parts of the system that access disk. For example, prefetching is associated with both the the virtual memory system and the file system. Independent of what subsystem prefetching is part of, however, its functionality always includes three elements: (1) a prediction of what information is likely to be needed in the near future, (2) an evaluation of the cost-effectiveness of retrieving this information, and (3) a request to the lower level disk subsystem to fetch this information.

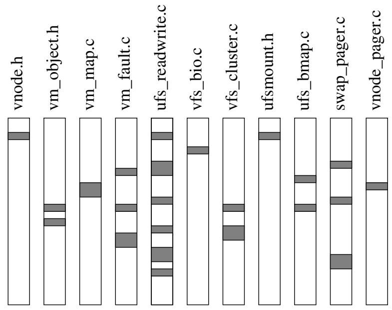


Figure 1: Prefetching code in FreeBSD.

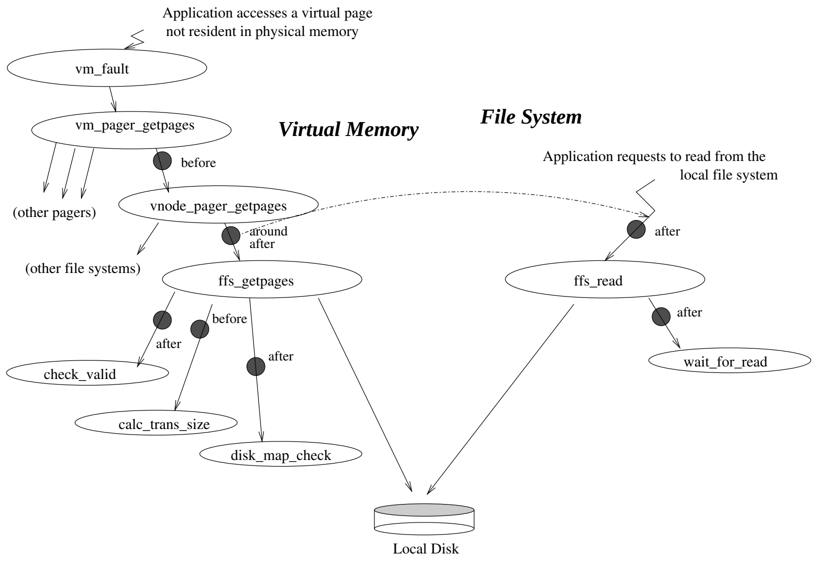


Figure 2: Page fault and file system read paths in FreeBSD.

In FreeBSD version 3, an operating system written in C, the implementation of prefetching is spread throughout the existing system modularity, as outlined in Figure 1. Some of these scattered pieces of prefetching code work together to perform a coordinated task, while others are different manifestations of the same prefetching policy enforced by subsystem-specific mechanisms. Much of the complexity comes from the interaction of virtual memory and file system prefetching strategies.

The goal of this experiment was to determine if an aspect-oriented approach can be used to improve the modularity of OS code, and further, to consider if a language like AspectC could be of use. In this study, we re-implemented prefetching in FreeBSD using an aspect-oriented approach [6, 8]. Since an aspect-oriented programming language for C does not exist, we postulated one very similar to the static part of AspectJ version 0.7. We wrote code in this language and then carefully hand-compiled it to ordinary C. The hand-compiled version of our code is available at on our web page [2].

# 3 Aspect-oriented prefetching

Figure 2 shows a high level overview of the main functions involved in the page fault handling and file system read paths in FreeBSD. The *before*, *after* and *around* labels associated with the circles on the edges of the call graph denote locations where advice applies.

In this section, we present our prefetching aspects incrementally. First, we focus on prediction and retrieval in the page fault handling path, and the file system path, respectively. Then, we show how the modification of the prediction aspect from the file system can be extended to include page fault handling. Finally, we see how this extension can be used to build a new strategy for page fault handling to include file system prefetching functionality.

### 3.1 Page fault handling path

A virtual memory *object* is a collection of related pages. In the example we consider here, the object we access is a file that has been mapped into memory using the *mmap()* command.

In FreeBSD, every virtual memory object has a declared behaviour, which is *normal* by default, and can be set to *random* or *sequential* using the *madvise()* system call. This declared behaviour is used to help determine how many and which pages should be prefetched during the handling of a page fault: random objects do not prefetch, normal objects prefetch a few pages before and a few pages after the faulted page, and sequential objects prefetch a larger range of pages that come after the faulted page.

In our prototype, the simple aspect *DeclaredBehaviour* is used to introduce, initialize, set and get an object's declared behaviour, as shown below.

```
aspect DeclaredBehaviour {

  static Behaviour NORMAL      = ..;
  static Behaviour RANDOM      = ..;
  static Behaviour SEQUENTIAL  = ..;


  introduce Object { Behaviour declaredBehaviour = NORMAL; }

  void set_declared_behavior(Object object, Behaviour behaviour) {   //madvise() calls this
    object->declaredBehaviour = behaviour;
  }

  Behavior getDeclaredBehaviour( Object object ) {
    return object->declaredBehaviour;
  }

}
```

A second aspect, *FFSDefaultPagingPolicy*, uses declared behaviour to plan prefetching. This plan is implemented by allocating additional pages of physical memory to the object associated with the faulted page. Then, lower levels will fetch those additional pages. Note that some or all of these pages can later be deallocated if the page becomes valid, if the page is not found in the map of blocks on disk, or if the contiguity of pages has changed.

Before examining the advice in this aspect, we introduce the following shared pointcut used to specifically identify control flow rooted at a call to vm_fault:

```
aspect PrefetchingSharedDefinitions {
  pointcut handlingFault(Map map):
    cflow( callto( vm_fault(map, ..) ) );
}
```

The first piece of advice in *FFSDefaultPagingPolicy*, shown below, executes before calls to vnode_pager_getpages that occur during the handling of a page fault. Here, the prediction aspect identifies the declared behaviour of the object, and additional pages are added to the map of pages for this object as the behaviour dictates. Note that the use of the *handlingFault* pointcut gives us access to the page map passed into vm_fault, which must be locked while pages are inserted for the object.

```
aspect FFSDefaultPagingPolicy {

  before( Map map, Object obj, VMPage[] pagelist, int* length, int faultedPage ):
           PrefetchingSharedDefinitions.handlingFault(map) &&
           callto( vnode_pager.getpages( obj, pagelist, length, faultedPage )) {

    Behaviour declaredBehaviour = DeclaredBehaviour.getDeclaredBehaviour( obj );

    if ( declaredBehaviour != RANDOM ) {
      VMmap.lock( map );
      alloc_prefetch_page_range( obj, declaredBehaviour, pagelist, *length, faultedPage );
      VMmap.unlock( map );
    }
  }
```

The second piece of advice handles the case where disk access for the faulted page is cancelled. This occurs if the page becomes resident in memory or cannot be found in the map associated with the disk. Both these conditions are checked during the execution of ffs_getpages, and if either are true, all pages for prefetching are freed.

```
after( Object obj, VMPage[] pagelist, int* length, int faultedPage, boolean check ):
        cflow( callto( ufs_readwrite.ffs_getpages(obj, pagelist, length, faultedPage ))) &&
        ( callto( check_valid(..)     returning check ) ||
          callto( disk_map_check(..) returning check ) ) {

  if ( check )
    dealloc_all_prefetch_pages( pagelist, *length, faultedPage );
}
```

Before we actually read the page, interrupts or other system activities could change the location of blocks on disk. The final piece of advice re-establishes the continuity of the pages included in the final request to transfer from disk. Any currently non-contiguous pages are freed at this point.

```
before(Object obj, VMPage[] pagelist, int* length, int faultedPage):
        cflow( callto( ufs_readwrite.ffs_getpages(obj, pagelist, length, faultedPage ))) &&
        callto( calculate_transfer_size(..) ) {

  check_for_and_deallocate_noncontig_prefetch_pages( pagelist, *length, faultedPage );
}
```

## 3.2 File system read

The prediction strategy for prefetching in the file system read path relies on the dynamic detection of sequential access. This detection requires maintenance of the position of the last access in the file, stored in the data structure associated with the file in memory, the *vnode*. File access is on the granularity of *blocks* rather than pages, where a block is typically the size of two pages.

In our prototype, we localized prediction for the file system into a simple aspect, *ObservedBehaviour*. This aspect introduces the *lastBlockRead* field to the *vnode*, and sets this field according to the advice shown below. Since this field must be updated according to the last access point in the file, this advice executes after every file system read request.

```
aspect ObservedBehaviour {

  introduce Vnode { Block lastBlockRead      = null; }

  after( Vnode_ptr vp, Buffer buff, Block block, int size):
        ufs_readwrite.ffs_read( vp, buff, block, size) {

    vp->v_lastBlockRead = block;
  }
```

Sequential behaviour is detected when a read request accesses the block immediately following the most recent position of access in the file. Since the most recent read request is tracked using *v_lastBlockRead*, sequential access is defended relative to this value, as shown in the *getBlockBehaviour* function below.

```
  Behaviour getBlockBehaviour( Vnode_ptr vp, Block block ) {
    if ( block == vp->lastBlockRead + 1)
      return SEQUENTIAL;
    else
      return null;
  }
}
```

### 3.2.1 File system sequential prefetching

In the file system read path, when access is sequential, an asynchronous request to prefetch is issued even if the actual requested block is found in memory.

```
aspect DefaultBlockPrefetchPolicy {
```

The ffs_read pointcut denotes the invocation of the initial entry point to the local file system. The ffs_read_transaction pointcut denotes the entire operation of reading the missing block.

```
pointcut ffs_read(Vnode_ptr vp, Buffer buff, Block block, int size):
  callto(ufs_readwrite.ffs_read(vp, buff, block, size);

pointcut ffs_read_transaction(Vnode_ptr vp, Buffer buff, Block block, int size):
  cflow( ffs_read(vp, buff, block, size) );
```

So what we need to do is issue the asynchronous read either right before we do a synchronous read or after we finish the whole transaction if there was no synchronous read during the transaction. The next two pieces of advice implement this. *Didit* is a dynamically bound variable that allows the two advice to communicate whether the synchronous read was done.

```
special didit;

around( Vnode_ptr vp, Buffer buff, Block block, int size ):
         ffs_read(vp, buff, block, size) {

  dynamic boolean didit = false;
  continue;
}


before(Vnode_ptr vp, Buffer buff, Block block, int size):
          ffs_read_transaction(vp, buff, block, size) &&
          callto( wait_for_read( .. ) ) {

  if (ObservedBehaviour.getBlockBehaviour(vp, block)) {
    vfs_cluster.do_asynch_prefetching( vp, buff, block );
    didit = true;
  }
}
```

The second piece of advice executes after ffs_read, and only issues a prefetching request if it has not been done.

```
after ( Vnode_ptr vp, Buffer buff, Block block, int size ):
         ffs_read(vp, buff, block, size) {

  if ( !didit && (ObservedBehaviour.getBlockBehaviour(..) == SEQUENTIAL ))
    vfs_cluster.do_asynch_prefetching( vp, buff, block );
}


}
```

### 3.3 Observed behaviour for the page fault path

So far, we have seen prefetching as it exists independently in the page fault handling and file system paths. We now introduce the dynamic detection of sequential access for the page fault handling path, and a new strategy that allows page fault handling to use the file system prefetching.

The functionality for observed behaviour is already in place for the file system path, so introducing this dynamic prediction to the page fault path requires a new field, *lastPageReadbyFault*, and new set and get functionality. These changes to our *ObservedBehaviour* aspect are shown below.

```
aspect ObservedBehaviour {

  introduce Vnode  { Block  lastBlockRead      = null; }      // was
                                                              // in
  after( Vnode_ptr vp, Buffer buff, Block block, int size):   // previous
          ufs_readwrite.ffs_read( vp, buff, block, size) {    // version
                                                              // of
    vp->v_lastBlockRead = block;                              // aspect
  }                                                           //
                                                              //
  Behaviour getBlockBehaviour( Vnode_ptr vp, Block block ) {  //
    if ( block == vp->lastBlockRead + 1)                      //
      return SEQUENTIAL;                                      //
    else                                                      //
      return null;                                            //
  }                                                           //


  introduce Object { VMPage lastPageReadbyFault = null; }

  after( Object, VMPage[] pagelist, int* length, int faultedPage ):
          handlingFault(Map) &&
```

```
        ufs_readwrite.ffs_getpages( obj, pagelist, length, faultedPage ) {

    obj->lastReadbyFault = pagelist[*length];
  }


  Behaviour getPageBehaviour( Object obj, VMPage[] pagelist, int length, int faultedPage ) {
    if ( pagelist[faultedPage] == obj->lastReadbyFault + 1)
      return SEQUENTIAL;
    else
      return null;
  }
}
```

Now that sequential access can be detected in the page fault handling path, we can build a more aggressive prefetching strategy for the page fault path by redirecting this disk request to the file system read path. This functionality is implemented by the around advice in *FFSaggressivePagePrefetchPolicy* shown below. Here, if aggressive prefetching is deemed to be warranted, the size of the read request is doubled and serviced by ffs_read, where further asynchronous prefetching may be applied.

```
aspect FFSaggressivePagePrefetchPolicy {

  around( Object obj, VMPage[] pagelist, int* length, int faultedPage ):
          callto(ufs_readwrite.ffs_getpages( obj, pagelist, length, faultedPage ) {

    if ( should_aggressively_page_ahead( obj, pagelist, length, faultedPage ) ) {

      int size = length * 2;
      Buffer buff = pagelistToBuff(pagelist);

      ufs_readwrite.ffs_read( vp, buff, block, size );
    }
    else
      continue;
  }
```

The predicate used to determine if aggressive prefetching is warranted is shown below. Objects with declared behaviour that is *sequential* automatically qualify. In the original implementation, *normal* objects with sequential access would also qualify, but we refined this slightly in our prototype by introducing a minimum threshold number of contiguous pages. *Normal* objects below this threshold do not use aggressive prefetching, and thus are not forced to wait for multiple disk reads.

```
  const int MIN_PAGES_TO_AGGRESSIVELY_PAGE_AHEAD = 5; //min threshold

  Boolean should_aggressively_page_ahead( Object obj, VMPage[] pagelist,
                                          int length, int faultedPage ) {
    return (( DeclaredBehaviour.getDeclaredBehaviour(obj) == SEQUENTIAL ) ||
            ( DeclaredBehaviour.getPageBehaviour( obj, pagelist,
                                                  length, faultedPage) &&
              ( length >= MIN_PAGES_TO_AGGRESSIVELY_PAGE_AHEAD )));
  }

  }
```

# 4 Evaluation

Although we are only in the initial stages of experimentation, we have been pleased with our experience of coding prefetching using AspectC. We feel that the aspect-oriented code is easier to understand, and in particular that it is easier to understand the interaction between the different virtual memory and file system prefetching mechanisms.

In one test of whether the AspectC implementation was more maintainable, we developed a mechanism to develop a "stride" access pattern. The changes required to do so were straightforward and relatively well localized. In the ObservedBehaviour aspect we added a new field to Object, the field holds the relative distance, in terms of page numbers, between successive accesses. Additional advice in the aspect maintains the field. This new form of access detection is shared between the virtual memory system and the file system.

When providing the lower level mechanism to support this pattern of disk access, we also discovered a lingering bug in the FreeBSD version 3 code. The *breadn* function only works with *n = 1*.

# 5 Conclusions

Our prototype implementation of prefetching using an aspect-oriented approach simplifies page fault handling code and creates a better context in which to understand the data and algorithms that implement virtual memory and file system prefetching.

Further work is needed to see whether other parts of the operating system code would also be improved by being written this way. Based on this additional experimentation we need to make a decision about whether to actually implement AspectC and do more extensive experimentation.

At the workshop we hope to be able to talk with other groups that have re-engineered complex software using aspect-oriented or other separation of concerns techniques. We want to share our experiences so that we can better understand the strengths, weaknesses and applicability of this technology.

# References


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

[2] http://www.cs.ubc.ca/spider/ycoady/aspectc.html.

[3] 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.

[4] 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.

[5] Gregor Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings of IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992.

[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] Cristina Videira Lopes and Gregor Kiczales. Recent developments in aspectj. In *European Conference on Object-Oriented Programming (ECOOP)*, 1998.

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

[10] 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.

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