Structuring System Aspects

Yvonne Coady, Gregor Kiczales, Mike Feeley, Norm Hutchinson, and Joon Suan Ong
University of British Columbia

Key elements of operating systems are crosscutting – their implementation is necessarily spread across several core components of the system. *Prefetching*, for example, is a critical architectural performance optimization that amortizes the cost of going to disk by predicting and retrieving additional data with each explicit disk request. The implementation of prefetching, however, is tightly coupled with both high-level context of the original request and low-level costs of additional retrieval. As a result, small clusters of customized prefetching code appear at both high and low levels along most execution paths that involve going to disk. This makes prefetching difficult to reason about and change, and interferes with the clarity of the primary functionality within which prefetching is embedded.

Aspect-oriented programming (AOP) provides a means of tackling some of the well-known modularity problems operating systems face when implemented with procedural and OO programming alone. This paper presents the use of AOP in structuring the implementation of a subset of prefetching in the FreeBSD v3.3 operating system.

Page fault handling and prefetching

A process generates a 'page fault' by accessing an address in virtual memory (VM) that is not resident in physical memory. Page fault handling begins in the VM layer as a request for a page associated with a VM object. This request is then translated into different representation – a block associated with a file – and processed by the file system. Finally, the request is passed to the disk system, where it is specified in terms of cylinders, heads and sectors associated with the physical disk. The division of responsibilities among these layers is centered around the management of their respective representations of data.

FreeBSD v3.3 associates a 'behaviour' of access, typically set to *normal* or *sequential*, with each VM object. Prefetching in the VM layer uses this declared behaviour to plan which pages to prefetch, and allocates physical memory according to this plan. Allocating physical memory involves VM-based synchronization, as the VM object's page map must be locked during this operation.

The execution path taken subsequent to the VM layer depends upon the declared behaviour of the VM object and requires that the file system pay special attention to the pages allocated during the planning phase. Normal behaviour involves checking the plan and de-allocating pages if it is no longer cost effective to prefetch. Sequential behaviour involves requesting a larger amount of data through the regular file system read path, while still ensuring the physical pages allocated during the planning phase are filled.

Prefetching structure and the original code

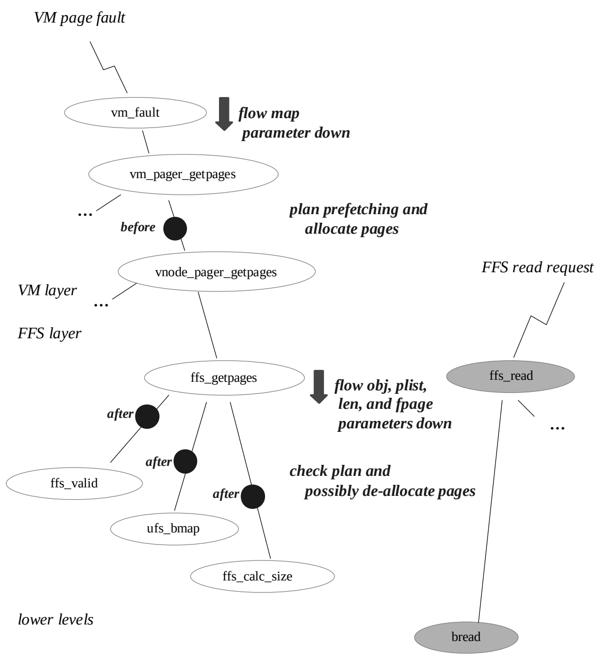
In the original code, the implementation of prefetching is scattered and tangled. In this example, it is spread out over approximately 260 lines in 10 clusters in 5 core functions from two subsystems. There are clusters of code operating on VM abstractions sitting in FFS functions. This implementation makes it very difficult to see the coordination of prefetching activity, and obfuscates the primary functionality of the page fault handling and file system read paths.

AspectC

The structured implementation of prefetching presented here uses *AspectC* – a simple AOP extension to C. Overall, only a small portion of the code relies on these linguistic extensions. These extensions modularize crosscutting concerns by allowing fragments of code that would otherwise be spread across several functions to be co-located and to share context.

AspectC is a simple subset of AspectJ. 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*, extension. *Cflow* supports the coordination of high-level and low-level prefetching activity along specified paths of execution.



```
aspect normal_prefetching {

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

  pointcut ffs_getpages_cflow( vm_object_t obj, vm_page_t* plist, int len, int fpage ):
      cflow( calls( int ffs_getpages( obj, plist, len, fpage )));

  before( vm_map_t map, vm_object_t obj, vm_page_t* plist, int len, int fpage ):
      calls( int vnode_pager_getpages( obj, plist, len, fpage ))
      && vm_fault_cflow( map )
  {
    if ( obj->declared_behaviour == NORMAL ) {
      vm_map_lock( map );
      plan_and_alloc_normal( obj, plist, len, fpage );
      vm_map_unlock( map );
    }
  }

  after( vm_object_t obj, vm_page_t* plist, int len, int fpage, int valid ):
      calls( valid ffs_valid(..) )
      && ffs_getpages_cflow( obj, plist, len, fpage )
  {
    if ( valid )
      dealloc_all_prefetch_pages( obj, plist, len, fpage );
  }

  after( vm_object_t obj, vm_page_t* plist, int len, int fpage, int error, int reqblkno ):
      calls( error ufs_bmap( struct vnode*, reqblkno, ..) )
      && ffs_getpages_cflow( obj, plist, len, fpage )
  {
    if ( error || (reqblkno == -1) )
      dealloc_all_prefetch_pages( obj, plist, len, fpage );
  }

  after( vm_object_t obj, vm_page_t* plist, int len, int fpage, struct t_args* trans_args ):
      calls( int ffs_calc_size( trans_args ))
      && ffs_getpages_cflow( obj, plist, len, fpage )
  {
    dealloc_noncontig_prefetch_pages( obj, plist, len, fpage, trans_args );
  }
}
```

Figure 1: Normal mode prefetching

Normal mode prefetching in AspectC

Figure 1 shows an aspect-oriented implementation of prefetching for VM objects with normal declared access. The call graph on the left illustrates each element of the aspect code on the right. Elements of the implementation associated with control flow correspond to the arrows adjacent to functions, and each advice declaration is represented by a small circle on an edge of the call graph.

The first declaration in the aspect in Figure 1 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.

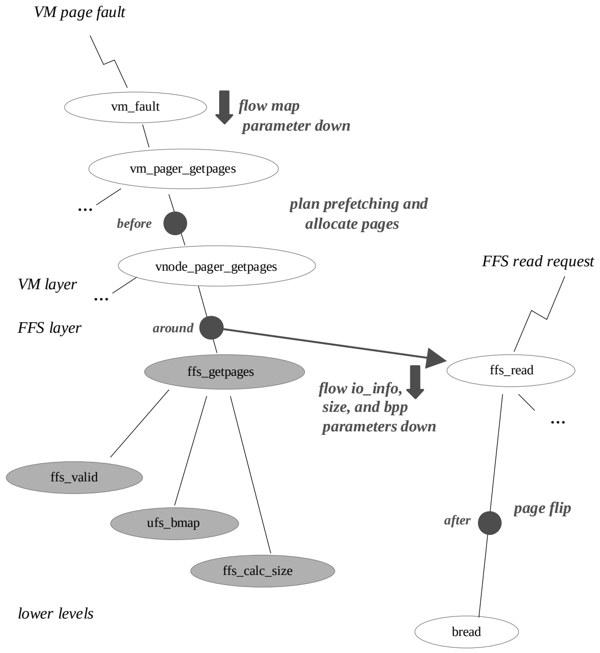
The second declaration is another pointcut, this time named *ffs_getpages_cflow*, which allows advice in the aspect to access the entire parameter list of *ffs_getpages*. This pointcut will be used by the lower advice that for de-allocation of planned pages.

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 ':', 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* associated with the particular fault currently being serviced (i.e., from a few frames back on the stack). The body of the advice is ordinary C code.

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 of pages previously allocated for prefetching.

Each of these after advice uses *ffs_getpages_cflow* to provide access to higher-level parameters and to ensure that the advice to de-allocate runs only within the control flow of an execution path rooted in *ffs_getpages*.



```
aspect sequential_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_inf, int size, struct buff** bpp ):
      cflow( calls( int ffs_read( vp, io_inf, size, bpp )));

  before( vm_map_t map, vm_object_t obj, vm_page_t* plist, int len, int fpage ):
      calls( int vnode_pager_getpages( obj, plist, len, fpage ))
      && vm_fault_cflow( map )
  {
    if ( obj->declared_behaviour == SEQUENTIAL ) {
      vm_map_lock( map );
      plan_and_alloc_sequential( obj, plist, len, fpage );
      vm_map_unlock( map );
    }
  }

  around( vm_object_t obj, vm_page_t* plist, int len, int fpage ):
      calls( int ffs_getpages( obj, plist, len, fpage ))
  {
    if ( obj->behaviour == SEQUENTIAL ) {
      struct vnode* vp = obj->handle;
      struct uio* io_inf = io_prep( plist[fpage]->pindex, MAXBSIZE, curproc );
      int error = ffs_read( vp, io_inf, MAXBSIZE, curproc->p_ucred );
      return cleanup_after_read( error, obj, plist, len, fpage );
    } else
      proceed;
  }

  after( struct uio* io_info, int size, struct buf** bpp ):
      calls( int bread(..) )
      && 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 2: Sequential mode prefetching

Sequential mode prefetching in AspectC

Figure 2 shows the structure of prefetching for objects with declared sequential access. In this case, the request is bumped to a maximum buffer size and routed through *ffs_read* instead of *ffs_getpages*. Once the transfer is complete, appropriate buffer pages are 'flipped' with the pages allocated for prefetching in order to avoid an expensive copy operation.

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 only when control flow has been diverted along this special path, as specified by the pointcuts *vm_fault_cflow* and *ffs_read_cflow.*

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. In this example, refactoring spawned two new functions, *ffs_valid* and *ffs_calc_size*, from *ffs_getpages.*

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 along these execution paths. At the same time, the AOP implementation also clearly specifies the exact dynamic execution context for each advice relative to the primary functionality involved.

These properties are essential for structuring aspects in systems code, where execution paths are often differentiated by their respective sets of restrictions and requirements. In this example, the execution path for normal mode prefetching is subject to tighter cost constraints relative to the sequential path, while the sequential path must instead reconcile the destination of the disk transfer.

Conclusion

In its original implementation, prefetching is tangled – spread throughout the code in an unclear way. Implemented with AOP, the crosscutting structure of prefetching is clear and tractable to work with. A semantically accurate structuring of prefetching hinges on the ability to identify and capture dynamic execution context. This result holds out promise that after years of trying to improve OS modularity with procedural and OO programming, AOP now may help us make significant progress.