Structuring Operating System Aspects

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

Key elements of operating systems crosscut – their implementation is inherently coupled with several layers 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 request source and low-level costs of additional retrieval. In a traditional OS implementation, 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.

This article explores the use of AOP [4] to improve OS structure [5] by highlighting an AOP-based implementation of a subset of prefetching in the FreeBSD v3.3 operating system.

Example: 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 a different representation – a block associated with a file – and processed by the file system (FFS). 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.

Applications associate an access behaviour, typically *normal* or *sequential*, with each VM object. Prefetching uses this declared behaviour to plan which pages to prefetch, and allocates physical memory pages according to this plan. Allocating pages involves VM-based synchronization, since 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 previously allocated pages. 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 allocated physical pages are filled.

Prefetching structure and the original code

In the original FreeBSD v3.3 code, the implementation of prefetching is both scattered and tangled. The code 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* [1] – 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 subset of AspectJ [2] (see article in this issue), without any support for OOP or explicit modules. Instead, we use the C convention of using a file to conceptually delimit a module. Aspect code, known as *advice*, interacts with primary functionality at function call boundaries and can run *before*, *after* or *around* existing function

calls. 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, mechanism. Cflow supports the coordination of high-level and low-level prefetching activity along an execution path by exposing specific high-level context, such as function calls and parameters, to lower-level advice.

Execution paths to disk

Figure 1 shows three colour-coded paths to disk, two of which have been previously introduced: normal and sequential page fault handling. The third is the file system read path. Functions in the (simplified) primary functionality call graph are represented by ellipses labeled with function names.

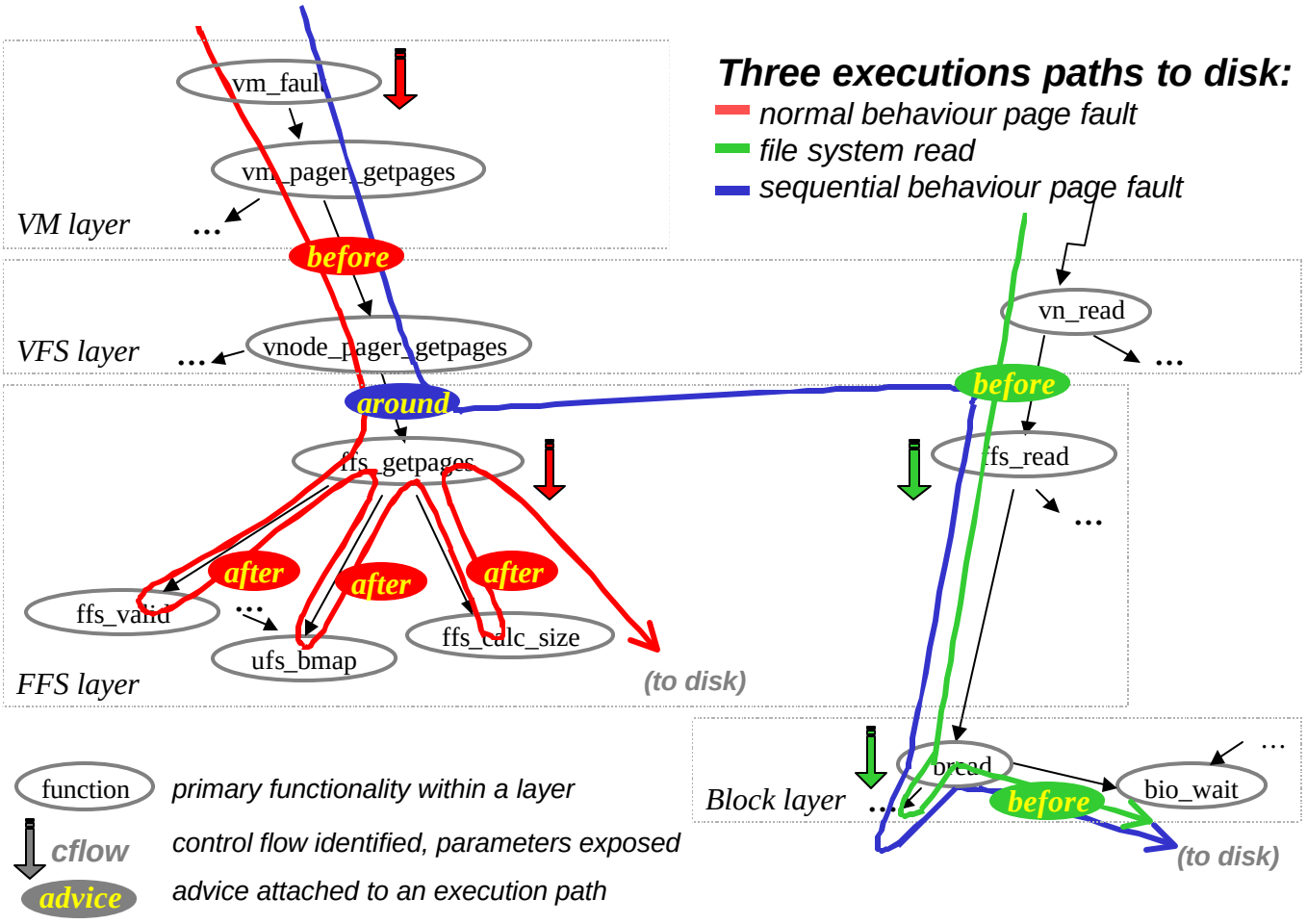

Figure 1. Execution paths to disk.

Normal behaviour prefetching in AspectC

Figure 2 shows an aspect-oriented implementation of prefetching along the normal behaviour page fault path. To develop this implementation, we first stripped prefetching out of the primary page fault handling. We then 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*.

This small aspect, *normal_prefetching*, contains two pointcut declarations, which identify and expose important control flow information, and four advice declarations, structured according to these pointcuts.

```
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 2. Aspect for normal behaviour prefetching during page fault handling.

Pointcut declarations

A pointcut identifies a collection of function calls and specific arguments to those calls. The first declaration in the aspect is a pointcut named *vm_fault_cflow*, with one parameter, *map*. The details are in the second line of the declaration: this pointcut refers to all function calls within the control flow of calls to *vm_fault*, and exposes *vm_fault*'s first argument, the page map. This pointcut is used by advice to access the page map for planning and allocating prefetched pages. The '..' in this parameter list means that although *vm_fault* has more parameters, they are not exposed by this pointcut.

Similarly, the second declaration is another pointcut, named *ffs_getpages_cflow*, which allows advice to access the entire parameter list of *ffs_getpages*. This pointcut is used by advice for de-allocating planned pages.

Advice declarations

Advice in this aspect are shown as four colour-coded ellipses associated with normal behaviour page fault handling in Figure 1. Each is labeled as executing *before* or *after* the function directly below it in the call graph. Places where control flow information is exposed are indicated by small arrows adjacent to specific functions.

The first advice in the aspect is responsible for the high-level planning and allocating of prefetched pages according to the object's behaviour. 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*.

In more detail, the first line of the header 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 exposes 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 low-level details associated with retrieval. There are 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 the necessary parameters and to ensure the advice runs only within the control flow of an execution path that includes *ffs_getpages*. This is important because *ufs_bmap* is part of many other execution paths in the system.

Implementation Comparison

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 pages along a given execution path.

Structuring the code this way – as path-specific customizations – has helped us refactor several other prefetching aspects, including one for sequential behaviour page fault handling and another for file system reads [3].

Conclusion

In its original implementation, prefetching in FreeBSD v3.3 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. This structuring hinges on the ability to identify and capture dynamic execution context. This result suggests that proper use of AOP may enable improving OS modularity beyond what is possible with procedural and OO programming.

References

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

[2] AspectJ. www.aspectj.org

[3] Yvonne Coady, Gregor Kiczales, Mike Feeley and Greg Smolyn. **Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code**. In Proceedings of Joint ESEC and FSE-9, 2001.

[4] 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.

[5] Paniti Netinant, Constantinos Constantinides, Tzilla Elrad, Mohamed Fayad. **Supporting Aspectual Decomposition in the Design of Operating Systems**. Position paper, ECOOP Workshop on Object-Orientation and Operating Systems, 2000