

Crosscutting the Great Divide: Exploring an Aspect-Oriented Approach to OS Code

Thesis Proposal
Yvonne Coady

January 22, 2001

“There probably isn’t a ‘best’ way to build the system, or even any major part of it; much more important is to avoid choosing a terrible way, and to have clear division of responsibilities among the parts.” – Butler Lampson [33]

Abstract

Operating system code is complex. But, while substantial complexity is inherent to this domain, some complexity is caused by modularity problems. The goal of the work proposed here is to explore aspect-oriented programming as a means of making this kind of complexity unnecessary. Towards this end, we need to identify aspects of the system, use the proposed mechanisms of AOP to modularize them, and establish the impact this approach has on the code.

1 Introduction

Operating systems have a problem with modularity. Complex interactions among components of the system blur module boundaries and make them highly susceptible to erosion. A study of OS/360 done in the early 70s [37] showed that the average number of modules involved in change rose from 14.6% in releases 2–6, to 31.9% in releases 12–16 due to “unintentional interaction” among components. Modern commercial operating systems do not appear to be faring any better with modularity. Windows NT, for example, requires 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 [58]. Recently, Engler et al. captured popular sentiment with the observation that “restrictions link together the entire system, making a fragile and intricate mess” [14].

The question of how to structure operating system services in an evolutionary-friendly manner remains largely unanswered. Research in extensible systems [55, 4, 52] highlighted this problem when a seemingly reasonable goal – incremental customization [10, 29] – proved to be elusive. That is, attempts to ensure that making a change to a service would require an effort proportional to the amount of code involved met with only limited success [11].

Recently, the aspect-oriented programming (AOP) [30] 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].

This proposal suggests a plan to explore AOP as a means of improving the modularity of OS code. The work required to perform this exploration includes facets of systems, programming languages and software engineering. Specifically, the plan proposes to identify several coarse grain aspects in operating system code, implement them using AOP language constructs, and analyze the impact from a software engineering perspective.

2 Related Work: Structure and Change

Ossher defined structure as the decomposition of a system into parts and the interaction of those parts [45]. The software engineering community has shown that structure plays a key role in determining the cost of change [48, 37, 44], and structural deficiency severely impairs maintenance and evolution [20].

Breaking a system into parts 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 [48]. Stevens et al. later suggested that functional binding, 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 [20]. Meantime, in the programming language community, Dahl and Nygaard started breaking systems into what we now know as objects, and provided linguistic support for object-oriented programming in Simula67 [8].

2.1 A systems perspective

The advantage of structure is well known and appreciated within the systems community. Hierarchical or a layered structure, which enforces independence of higher levels of the system from lower levels, has been recognized as key since the THE multiprogramming system [9] in the 60s. End-to-end arguments [50, 49] 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. Similar to the philosophy of reduced instruction set (RISC) architecture, the goal is to provide only primitive tools and not to anticipate client's needs.

In keeping with the end-to-end argument, Lampson points out that a service should have a predictable cost, and features needed by only a few clients should not be included in an interface as it may penalize others as a result. For an example of offering too much, he points to Pilot [18], which was the first operating system to allow virtual pages to be mapped to file pages, thus subsuming file I/O within the virtual memory system. This extra functionality resulted in a larger and slower implementation relative to smaller and simpler interfaces offered by Alto [34] and Interlisp-D [19]. Part of the problem is the circularity of having a file system that would like to use the virtual memory system, while virtual memory depends on files.

One of the key contributions of a layered approach is that it can reduce complexity of interaction. At a coarse granularity, the simple layering of Petal and Frangipani [36, 35] compared with the single layer approach in xFS [3] is a good example of this principle at work. Petal is a distributed storage system upon which the Frangipani file system was built in a number of months. xFS is a file system designed with similar goals as Frangipani, but its implementation was substantially more complex.

Object-oriented structure has also proven to be valuable in systems. The *x*-kernel [24] uses an efficient implementation of an object-based framework to provide the practical decomposition of network protocols into *protocol* and *session* objects. This separation makes protocols easier to write and understand because they are isolated from connection issues, and allows services common to all protocols to be re-used. Although performance limitations may have curbed the widespread adoption of explicit language support for OO within system implementations, manual application of some of its key principles, such as polymorphism through the use of function-table dispatch, is a key structural element in OS code. Fundamental components in modern Unix systems – memory management, message-based communication, process scheduling and the file system interface – are all designed using OO concepts [54].

The need to support flexibility in operating system services was first addressed by attempts to separate policy and mechanism in Hydra [61]. The motivation behind this work was the observation that implementing policy in user space would allow it to be more easily modified. The microkernel architecture of the Exokernel [15, 17, 16, 26], took a similar approach by attempting to move all abstractions that could bias

application performance into libraries that were linked into an application's address space. In practice, this approach facilitated flexibility at the granularity of libraries [56]. Similar in-kernel support for coarse grain flexibility has been addressed by the Unix *VFS/Vnode* architecture [31] which supports multiple file system types.

Projects demonstrating the advantages of making particular policies flexible include paging in Mach [42], scheduling through activations [21], communication with active networks [53, 49], and prefetching and caching strategies with parameterized or split-level policy modules such as LRU-SP [5, 6, 7, 13, 23]. Efforts to customize policy through reflection also ranged theoretically from all OS policy in Apertos [62], to file system services [40, 41, 39] and virtual memory management [32] using metaobject protocols.

The degree to which the structure of OS code could support a finer granularity of change came under closer scrutiny with research in extensible systems such as Lipto [12], SPIN [4, 22], Vino [52], and Kea [55]. A key motivating factor driving research in this area was an application's need for incremental customization, or fine grain control over some part of a service, such as the prefetching policy of a file system. The goal was to make the effort required to implement this type of change proportional to the desired change in functionality.

Although this line of research was fruitful in terms of safety and performance issues, each research system was able to accommodate only a fixed range of anticipated changes. This range was dictated by the underlying structure of the functionality targeted for replacement. For example, SPIN's extensions enabled customized handling of fine grain events, such as page faults or the reception of a network packet, by allowing applications to register event handlers with a centralized dispatcher. One of the real problems SPIN uncovered with this fine grain approach was that the interface required to support extensibility was an order of magnitude wider than the typical Unix interface, and extremely hard to manage due to extensive semantic properties involved [51]. A similar problem was encountered in Kea, where the inability to structure a clean interface for policy-related elements of services inhibited extensibility [57].

2.2 A programming language perspective

The effort required to make a change is strongly related to the locality of that change, and thus closely related to structure. As a result, changes that span multiple parts of a system tend to be the most costly. One strategy from the programming language community designed to address this problem in object-oriented systems was the use of reflection in the form of metaobject protocols [28, 27]. This approach offered a powerful way of making system-wide, crosscutting changes by facilitating the modification of the language of implementation.

Structuring implementations along dimensions that go beyond standard procedural or object-oriented technology has been addressed by several other research projects in the separation of concerns research commu-

nity. Subject-oriented programming [47] and subsequent work on Hyperspaces [46] deal with collections of classes which define a view of a domain, and provide a means of integrating these multiple views for the development of complex systems. Composition filters take aim at problems associated with flexibility and reuse by allowing an object's behaviour to be composed according to an appropriate combination of filters [2].

Aspect-oriented programming as defined by work in the AspectJ project [1, 43, 25, 38], provides simple linguistic support for aspects. This approach is based on the premise that whereas some concerns dictate a natural primary modular decomposition of a system, others naturally crosscut this structure. One of the goals of this approach is to use simple mechanisms to better separate crosscutting functionality from the dominant decomposition of the system.

3 The problem

The problem is modularity. Despite much concerted effort, the implementation of key elements of operating system services seems to inevitably get spread out in the code. This property not only makes systems code difficult to reason about and change, but also was a significant barrier to incremental customization in extensible systems research. When an implementation is spread throughout the system, the effort associated with changing that implementation is typically not proportional to the amount of code involved.

4 Proposed solution

The structural deficiencies in OS code appear to be symptomatic of a modularity that contains inherent crosscutting. The goal of the work proposed here is to use AOP to eliminate the complexity introduced by the presence of crosscutting implementation in OS code. That is, we aim to make OS code easier to reason about and change. In order to do this, we have to: (1) identify elements of the system that are crosscutting, (2) use the proposed mechanisms of AOP to modularize them, and (3) establish how this approach impacts modularity.

5 Prototype

This section presents the initial results of our experiment using an aspect-oriented approach to simplify operating system code. In this experiment, we re-implemented a subset of prefetching for mapped files in

FreeBSD v3.3 using a hypothetical language, AspectC – a variant of AspectJ for C – and hand-compiled the code to C. AspectC extends C by adding linguistic support for aspects. Unlike AspectJ however, where aspects crosscut objects, aspects in AspectC crosscut functions. Overall, only a small portion of the code written in AspectC relies on these linguistic extensions for crosscutting. 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.

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 layout of the code.

5.1 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; it moves through 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.

5.2 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 *advise* 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 the locking of the page map.

Beyond the VM layer, the access behaviour of the VM object determines how execution proceeds. Normal

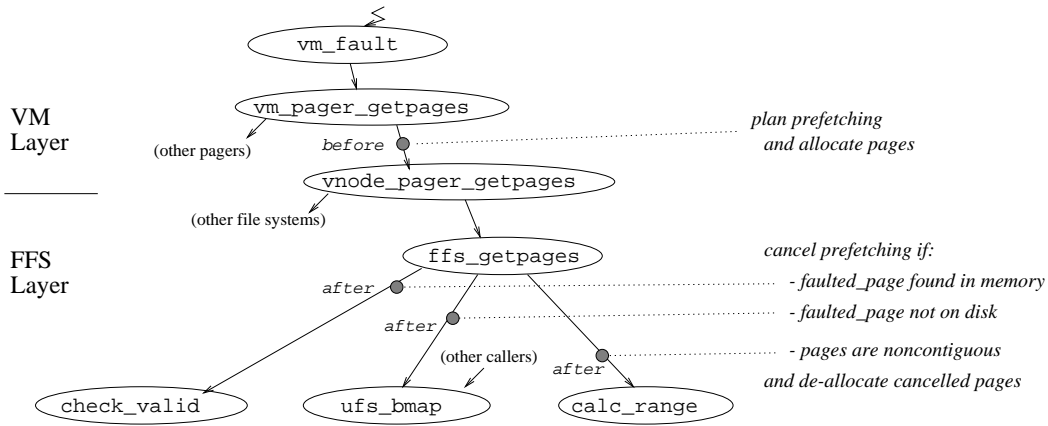


Figure 1: The structure of the page fault handling path for objects with behaviour declared to be normal. Only the top two layers, VM and FFS, are shown. The ovals represent functions comprising the primary page fault handling structure, the small circles and text in italics represent the structure of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

objects use the path shown in Figure 1, while the path for sequential objects is shown Figure 3.

5.2.1 Normal access prefetching

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

5.2.2 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 addi-

```

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.

tional asynchronous prefetching for the next blocks required in a sequential access pattern. It 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.

5.3 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 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.

5.4 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.

5.4.1 Normal prefetching in AspectC

Figure 2 shows our aspect-oriented implementation of prefetching for normal declared access. 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 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 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* 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 transferred.

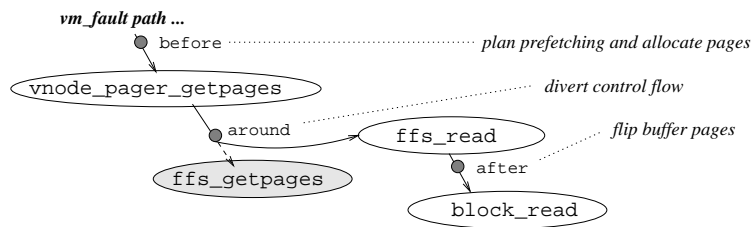


Figure 3: Execution path for objects with sequential declared access.

```

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.

5.4.2 Sequential prefetching in AspectC

Figure 3 illustrates the structure of prefetching for objects with sequential access patterns. The corresponding implementation is shown in Figure 4.¹

When *advise* 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 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 *vnnode*.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. The helper function *io_prep* sets up a read request using the faulted page and the maximum buffer size to be synchronously retrieved. *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, only when execution has been diverted along this special path.

5.5 Implementation comparison

To develop the AOP implementation, we first stripped the prefetching related code from the primary implementation of the page fault handling path. 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 of *ffs_getpages* to spawn two new small functions, *check_valid* and *calc_range*.

The key difference between the original and aspect-oriented implementations is that when implemented using an aspect, 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. By moving this functionality into an aspect and expressing it as advice, we were able to preserve the context of prefetching related execution and make the structure of the crosscutting explicit.

Although this is not a complete treatment of prefetching in the system, this experiment shows that some of the complexity in OS code is unnecessary because it comes from improper modularization techniques

¹The 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.

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

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.

6 The “TO DO” list

Exploring AOP as a means of improving the modularity of OS code will involve work in systems, programming languages, and software engineering. Specifically, we need to identify the structure of coarse grain aspects in operating systems, use AOP language constructs to modularize them, and establish the material impact of AOP on understandability and changeability. The following subsections focus on proposed work in each of these areas.

6.1 System aspects

All our refactorings will focus on versions 2-4 of FreeBSD. We need at least three big working examples to explore issues of scalability and interaction between aspects. The advantage of working within one system will be that we can build more sophisticated organizations of aspects more quickly. The disadvantage is the potential for the work to be less general. Code analysis by visual inspection of other systems, such as Linux and NetBSD, may be helpful to determine commonality among the implementations.

Prefetching is a good first step at identifying a concern that is inherently crosscutting. Other prefetching-related aspects we have experimented with include an aspect for tracking and observing sequential behaviour, and an aspect for asynchronous prefetching in the file system read path. Prefetching for other pagers and in other parts of the file system remains to be refactored, and will be the first thing we implement once AspectC is up and running.

VM and the file system are known to have a complex circular relationship, and thus are potentially a source of several interesting aspects. Another important element that crosscuts VM and the file system is page replacement. Developing aspects for this implementation in addition to prefetching should allow us to better reason about and potentially change these two closely coupled system policies. Once we have prefetching in place, we will have a better idea of the overall structure of these and other related policies. The final decision of exactly which VM/FFS aspects to focus on will thus be made after the implementation of prefetching.

Communication protocols is another area of the system we will target to identify aspects. Although the scope of this implementation is in part dependent on how much new ground it will cover relative to the previous results, the current plan is to explore an AOP implementation of functionality ranging from fine

grain issues such as duplicate message suppression and delivery acknowledgment to more comprehensive issues such as quality of service. The plan is to target both kernel and user-level implementations, and to consider the role aspects could play in new infrastructures for distributed systems [60].

Scheduling is another example that would be interesting to at least analyze from the perspective of cross-cutting if not implement. Replacing the scheduler in any of the BSDs is known to require a considerable amount of time just to find and fix dependencies in other modules. Speculation is that a system like Windows NT, with a stronger interface, could avoid these problems [59]. By targeting the scheduler in BSD, we may be able to better determine if these dependencies are merely an artifact of a weak interface, or if they are rooted in something more inherent, such as crosscutting. Attention to this issue is mostly dependent upon the sufficiency of results from the previous examples. That is, if we have not been able to achieve examples that build, scale and interact in a meaningful way, scheduling is a good candidate to have waiting in the wings.

6.2 Crosscutting modularization mechanisms

AspectC is a simple subset of AspectJ. As such, it is expected that developing a bare-bones, no-frills translator for this language will not be an overly difficult task. Although it is reasonable to assume that AspectC will evolve during the course of this project, AspectC is not the focus of the work, and our hope is that the general constructs we start with will be sufficient for our needs.

6.3 Evaluating Impact

We want to improve the modularity of the code – and that is tough to quantify. In the evaluation, we plan to present as many quantifiable metrics as we can, a logical argument for understandability, and several examples of changeability. All of these results will be reported on a case by case basis, with some additional comprehensive qualification of the overall organization in terms of scalability and interaction.

For metrics, we can establish an approximate degree of crosscutting for a given implementation using values such as the number of lines, clusters, functions, and subsystems involved. Another value that cannot be overlooked is performance, as we cannot afford to introduce a significant penalty with AOP.

Measuring understandability will be more challenging, and we are primarily counting on a logical argument to suffice. That is, we plan to show or describe the AOP implementation and highlight the nature of the coordination provided. However, softer indicators such as the number of minutes for an experienced programmer to find a bug [14] and an informal surveys of a groups of programmers will be useful to augment

this argument.

In terms of changeability, it may be that any given change will be local to crosscutting implementation, primary functionality, or some of both – but better modularity should help in all three of these cases. To test this, we plan to consider ‘evolutionary’ change in both the original and AOP implementations. Evolutions of this type will come from two sources. First, starting with a basic functionality of an early version of FreeBSD we can retroactively apply real evolutionary changes documented in the CVS repository. Second, speculating on future trends in commodity hardware and the escalating demands of applications, we can implement plausible future system evolution. We can then report on the relative support, in terms of locality, and related effort required to affect the change in both systems.

6.4 Planned submission deadlines

1. Feb 2001 - CACM AOP Issue - a short paper, prefetching example.
2. March 2001 - FSE - AspectC is up and running, larger-scale prefetching is implemented.
3. May 2001 - Middleware - aspects in communication, QoS issues.
4. Spring 2002 - OSDI -Implementation of several system aspects and evaluation of impact.
5. Hope to contribute to various AOP workshops along the way.

References

- [1] <http://www.aspectj.org>.
- [2] 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.
- [3] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.
- [4] 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.
- [5] Pei Cao. LRU-SP: An allocation algorithm for application-controlled caching. In *Proceedings of the 1997 Grace Hopper Celebration of Women in Computing Conference*, 1997.
- [6] Pei Cao, Edward W. Felten, Anna Karlin, and Kai Li. Implementation and performance of integrated application-controlled caching, prefetching and disk scheduling. In *ACM Transactions on Computer Systems (TOCS)*, November 1997.
- [7] Pei Cao, Edward W. Felten, and Kai Li. Application-controlled file caching policies. In *Proceedings of the USENIX Summer Technical Conference*, 1994.
- [8] Ole-Johan Dahl and Kristen Nygaard. How object-oriented programming started. In http://www.ifi.uio.no/kristen/FORSKNINGS/DOK_MAPPE/F_OO_start.html. Dept. of Informatics, University of Oslo.

- [9] E.W. Dijkstra. The structure of THE-multiprogramming system. *Communications of the ACM*, 11(5), 1968.
- [10] 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.
- [11] Peter Druschel, Vivek S. Pai, and Willy Zwaenepoel. Extensible kernels are leading OS research astray. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems (HotOS-VI)*, May 1997.
- [12] 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.
- [13] John K. Edwards and Pei Cao. User-oriented resource scheduling in UNIX. Technical report, Univ. of Wisconsin, 1996. CS-TR-96-1318.
- [14] 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.
- [15] Dawson R. Engler and M. Frans Kaashoek. Exterminate all operating system abstractions. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*. IEEE Computer Society, May 1995.
- [16] Dawson R. Engler, M. Frans Kaashoek, and James O’Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP ’95)*, December 1995.
- [17] Dawson R. Engler, M. Frans Kaashoek, and James W. O’Toole Jr. The Exokernel approach to extensibility (panel statement). In *Proceedings of the 1st USENIX Symposium on Operating System Design and Implementation (OSDI ’94)*, November 1994.
- [18] D.D. Rendel et al. Pilot: An operating system for a personal computer. 23(2), Feb. 1980.
- [19] R.R. Burton et al. Interlisp-D overview. In *Papers on Interlisp-D*. Technical report SSL-80-4, Xerox PARC, 1981.
- [20] Stevens et al. Structured design. *IBM Systems Journal*, 13, 1974.
- [21] Thomas E. Anderson et al. Scheduler activations: Effective kernel support for user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1), February 1992.
- [22] Marc Fiuczynski and Brian Bershad. An extensible protocol architecture for application-specific networking. In *Proceedings of the 1996 Winter USENIX Conference*, 1996.
- [23] Gideon Glass and Pei Cao. Adaptive page replacement based on memory reference behavior. In *Proceedings of SIGMETRICS’97*, 1997.
- [24] Norm Hutchinson and Larry Peterson. The x-kernel: An architecture for implementing network protocols. In *IEEE Transactions on Software Engineering*, volume 17(1), 1991.
- [25] John Irwin, Jean-Marc Loingtier, John R. Gilbert, Gregor Kiczales, John Lamping, Anurag Mendhekar, and Tatiana Shpeisman. Aspect-oriented programming of sparse matrix code. In *Proceedings of the International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE)*, 1997.
- [26] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP ’97)*, October 1997.
- [27] Gregor Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings of IMSA’92 Workshop on Reflection and Meta-level Architectures*, 1992.
- [28] Gregor Kiczales, Jim des Rivieres, and D Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

- [29] 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.
- [30] 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.
- [31] S.R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the 1986 USENIX Conference*, 1986.
- [32] Keith Krueger, David Loftesness, Amin Vahdat, and Thomas Anderson. Tools for the development of application-specific virtual memory management. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 1993.
- [33] Butler Lampson. Hints for computer system design. In *ACM Operating Systems Rev.*, October 1983.
- [34] B.W. Lampson and R.S. Sproull. An open operating system for a single user machine. 13(5), December 1979.
- [35] Edward K. Lee and Chandramohan A. Tekkath. Frangipani: A scalable distributed file system. In *SOSP*, 1997.
- [36] Edward K Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *ASPLOS*, 1996.
- [37] L.L. Lehman and L.A. Belady. Program evolution. *APIC Studies in Data Processing*, (27), 1985.
- [38] Cristina Videira Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical report, Xerox Palo Alto Research Center, 1997. SPL97-010 P9710047.
- [39] C. Maeda and B. Bershad. Service without servers. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, October 1993.
- [40] Chris Maeda. Flexible system software through service decomposition. In *OOPSLA*, August 1994.
- [41] Chris Maeda. *Service Decomposition: A Structuring Principle for Flexible, High Performance Operating Systems*. PhD thesis, CMU, 1997.
- [42] Dylan McNamee and Katherine Armstrong. Extending the Mach external pager interface to allow user level page replacement policies. In *Technical Report UWCSE 90-09-05, University of Washington*, September 1990.
- [43] Anurag Mendhekar, Gregor Kiczales, and John Lamping. RG: A case-study for aspect-oriented programming. Technical report, Xerox Palo Alto Research Center, 1997. SPL97-009 P9710044.
- [44] Gail C. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, University of Washington, 1996.
- [45] H. Ossher. A mechanism for specifying the structure of large, layered systems. *Research Directions in Object-Oriented Programming*, 1987.
- [46] 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.
- [47] 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.
- [48] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1972.
- [49] D.P. Reed, J.H. Saltzer, and D.D. Clark. Active networking and end-to-end arguments. In *IEEE Network*, June 1998.
- [50] 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.

- [51] Stefan Savage. Comment in session 4b: The thin red line. In *Digest of Proceedings of the Seventh IEEE Workshop on Hot Topics in Operating Systems*, March 1999.
- [52] Christopher Small and Margo Seltzer. A comparison of OS extension technologies. In *Proceedings of the USENIX Conference*, 1996.
- [53] D.L. Tennenhouse and D.H. Wetherall. Towards an active network architecture. *ACM Computer Communications Review*, 26(2):5–18, April 1996.
- [54] Uresh Valhalla. *UNIX internals, The new frontiers*. Prentice Hall Inc, 1996.
- [55] 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.
- [56] Alistar Veitch. *A Dynamically Reconfigurable and Extensible Operating System*. PhD thesis, University of British Columbia, 1998.
- [57] Alistar Veitch. A conversation after several glasses of wine at an asi reception. 1999.
- [58] Werner Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP)*, 1999.
- [59] Werner Vogels. Windows 2000 research edition, where the academic white knights meet the evil empire. February 1999.
- [60] Andrew Warfield and Norm Hutchinson. The importance of good plumbing. Technical report, UBC, 2001. 222-2222.
- [61] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. In *Communications of the ACM*, volume 17(6), 1974.
- [62] Yasuhiko Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 1992.