IMPROVING EVOLVABILITY OF OPERATING SYSTEMS WITH ASPECTC

by

MONICA YVONNE COADY

B.Sc., Gonzaga University, 1985
M.Sc., Simon Fraser University, 1988

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

____

____

____

____

____

THE UNIVERSITY OF BRITISH COLUMBIA

JULY 2003

© Yvonne Coady, 2003

ABSTRACT

Operating system code is complex. But, while substantial complexity is inherent to this domain, other complexity is caused by modularity problems. The implementation of certain key system concerns seems to defy traditional modular boundaries and impede evolution. From OS/360 to Windows NT, systems suffer from unintentional interactions between modules [Lehman and Belady 1985] and require developers to be intimately familiar with implicit patterns of interaction between subsystems [Vogels 1999].

The thesis of this work is that aspect-oriented programming (AOP) can be used to improve evolution in operating system code by improving modularity. Specifically, AOP can alleviate modularity problems associated with concerns that are inherently *crosscutting* – no single modular decomposition can localize both the crosscutting concern and the concerns it crosscuts. Better modularization of crosscutting concerns requires that their implementation be localized, their interaction with the parts of the system they crosscut be explicit, and their internal structure be clear. By accomplishing these three things, AOP can provide better structural support for evolution.

The first part of the thesis provides a case study comparing modularity involving three crosscutting concerns in the original versus aspect-oriented implementation within the FreeBSD operating system [Lehey 1999]. This comparison highlights specific improvements in modularity of both the crosscutting concerns and the concerns that are crosscut, or *interacting* concerns, in the AOP implementation.

The second part of the thesis surveys evolutionary changes the three crosscutting concerns underwent between releases 2.2 (1997), 3.3 (1999) and 4.4 (2001) of FreeBSD, and identifies some of the specific impediments the original implementation poses with respect to evolution. A case study of the impact of these same evolutionary changes on the aspect-oriented implementation highlights improvements in locality of change afforded by the AOP implementation, and the ways in which improvements in modularity persist across the versions.

The final part of the thesis presents inferences and generalizations based on the results of the case studies. We infer from the case studies that AOP can be used to improve evolvability of the three crosscutting concerns and their interacting concerns in three versions of FreeBSD, without harming non-interacting concerns. We then generalize these modularity benefits to more concerns, more systems, and more versions, and infer support for our main claim – that aspect-oriented programming can be used to improve evolvability of operating system code by providing better modularity of crosscutting concerns and their interacting concerns, without harming non-interacting concerns.

Table of Contents

ABSTRACT I

TABLE OF CONTENTS III

LIST OF TABLES VII

LIST OF FIGURES VIII

1 INTRODUCTION 1

1.1 THE THESIS 2
1.2 RELATED WORK: PERSPECTIVES ON STRUCTURE AND CHANGE 3
1.2.1 A Programming Language Perspective 3
1.2.2 A Systems Perspective 5
1.2.3 A Software Engineering Perspective 8
1.2.4 Modularity and OS Code 10
1.3 ARGUMENT STRUCTURE AND EXPERIMENTAL SET UP 11
1.4 BACKGROUND: AOP USING ASPECTC 12
1.4.1 Introduction to AOP 12
1.4.2 Language Design of AspectC 13
1.4.3 Design of AspectC Prototype Implementation 17
1.4.3.1 Cpp Passes 18
1.4.3.2 Parsing and Weaving 18
1.4.4 Runtime Architecture Interface 20
1.4.5 Integrating AspectC Runtime with the FreeBSD Kernel 21
1.5 THESIS OVERVIEW 22

2 CASE STUDY ONE: SINGLE VERSION MODULARITY ANALYSIS 25

2.1 CASE STUDY BACKGROUND 26
2.2 ORIGINAL IMPLEMENTATION: O2 28
2.2.1 Page Daemon Activation 29
2.2.1.1 Analysis of O2 Implementation 29
2.2.1.2 Consequences of Scattered and Tangled Implementation 32
2.2.2 Mapped-File Prefetching 33
2.2.2.1 Analysis of O2 Implementation 33
2.2.2.2 Consequences of Scattered and Tangled Implementation 35
2.2.3 Disk Quota 36
2.2.3.1 Analysis of O2 Implementation 36
2.2.3.2 Consequences of Scattered and Tangled Implementation 38
2.3 ASPECT-ORIENTED IMPLEMENTATION: A2 39
2.3.1 Page Daemon Activation as an Aspect 39
2.3.1.1 Improvements in Modularity and Implementation Alternatives 44
2.3.2 Mapped-file Prefetching as an Aspect 45
2.3.2.1 Improvements in Modularity and Implementation Alternatives 50
2.3.3 Disk Quota as an Aspect 50
2.3.3.1 Improvements in Modularity and Implementation Alternatives 54
2.4 VALIDATION 54

3 CASE STUDY TWO: MULTI-VERSION MODULARITY ANALYSIS 57

3.1.1 Page Daemon Activation Evolution 59
3.1.2 Mapped-File Prefetching Evolution 61
3.1.3 Disk Quota Evolution 63
3.2 IMPACT OF CHANGE TASKS ON THE ASPECT-ORIENTED KERNELS 65
3.2.1 Evolution of Page Daemon Activation Aspect 66
3.2.1.1 Page Daemon Activation Change Task Summary 70
3.2.2 Evolution of Prefetching Aspect 71
3.2.2.1 Prefetching Change Task Summary 75
3.2.3 Evolution of Quota Aspect 75
3.2.3.1 Quota Change Task Summary 77
3.2.4 Improved Modularity Persists Across Versions 77
3.2.4.1 Consistency 79

3.2.4.2 Coordination 80
3.2.4.3 Configurability 81
3.2.4.4 Discussion 82
3.3 VALIDATION 83
4 INFERENCES OF EVOLUTION AND SCALE 87
4.1 EVOLUTION 87
4.2 ADOPTION OF AOP: TOOLS AND REFACTORING 89
4.2.1 Tool Support 89
4.2.2 Separating Concerns 92
4.2.2.1 Reducing Scattering 93
4.2.2.2 Reducing Tangling 93
4.2.2.3 Separation of page daemon activation 94
4.2.2.4 Separation of prefetching 95
4.2.2.5 Separation of quota 95
4.3 COSTS: MICROBENCHMARKS FOR ASPECTC RUNTIME 96
4.4 SCALE 99
4.4.1 GMS 100
4.4.1.1 Key GMS Activities 100
4.4.1.2 System Integration 101
4.4.1.3 Implementation 103
4.4.2 Resource Containers 103
4.4.2.1 Key RC Activities 104
4.4.2.2 System Integration 104
4.4.2.3 Implementation 104
4.4.3 Bossa 106
4.4.3.1 Background: Bossa, Scheduling Decisions and Events 107
4.4.3.2 Key Bossa Activities 107
4.4.3.3 System Integration 108
4.4.3.4 Implementation 109
4.4.4 Challenges with Existing Concerns 110
4.5 INFERENCE OF MAIN CLAIM 112
5 SUMMARY AND CONCLUSIONS 113
5.1 CONCLUSIONS 119

5.1.1 Future Work *120*
5.1.2 Contributions *121*
BIBLIOGRAPHY 123
APPENDIX 133

List of Tables

Table 1: Matching of Pointcut Designators. *14*
Table 2: Kinds of context. *14*
Table 3: Definitions associated with AspectC runtime support. *21*
Table 4: Crosscutting, interacting and non-interacting concerns. *28*
Table 5: Places in the system where the page daemon is activated in O2. .... *30*
Table 6: Files involved with prefetching functionality in O2. .... *34*
Table 7: Files involved in quota functionality in O2. .... *37*
Table 8: Summary of improved modularity in A2. .... *55*
Table 9: Change tasks overviewed. *58*
Table 10: Page daemon activation in FreeBSD v2, v3 and v4. .... *60*
Table 11: Prefetching functionality in FreeBSD v2, v3 and v4. .... *62*
Table 12: Quota in FreeBSD v2, v3, and v4. .... *63*
Table 13: Changes (√) versus no change (captured), in AOP implementation. *66*
Table 14: Summary of page daemon activation change tasks. *70*
Table 15: Summary change tasks and impact on the prefetching aspect. *75*
Table 16: Change tasks associated with the quota aspect. *77*
Table 17: Summary of modularity problems. *78*
Table 18: Configurability of crosscutting concerns over the 11 changes tasks. *82*
Table 19: Examples of exposure of otherwise function-local values for each concern. *94*
Table 20: A subset of GMS operations and where they occur in FreeBSD v2. .... *102*
Table 21: A subset of Resource Container operations and where they occur in FreeBSD v4.0 ... *104*
Table 22: Modifications required in Linux to implement the Bossa event-based notification. *108*
Table 23: A subset of Bossa events and where they are raised in Linux v2.4 .... *109*

List of Figures

Figure 1: Claim structure for the dissertation. *11*

Figure 2: Partially simplified code for an example sequential prefetching aspect. *17*

Figure 3: Integration of the AspectC runtime library with the FreeBSD kernel. *22*

Figure 4: Pager and file system hierarchy in FreeBSD. *27*

Figure 5: Modules and functions involved page daemon activation in O2. .... *30*

Figure 6: Analyzing scattered code for page daemon activation in O2. .... *32*

Figure 7: Scattered implementation of prefetching in O2. .... *33*

Figure 8: Scattering of quota implementation across UFS, FFS and EXT2FS. *37*

Figure 9: Implementation of Quota in flushfile operations from FFS and EXT2FS. *38*

Figure 10: Contexts for calls to page daemon wakeup in O2. .... *40*

Figure 11: Original code for activation from O2. .... *43*

Figure 12: Prefetching for mapped files in FreeBSD v2. .... *47*

Figure 13: Aspect for normal mode prefetching. *48*

Figure 14: A subset of contexts for quota aspect in UFS, FFS and EXT2FS. *51*

Figure 15: Sequential mode prefetching for FFS in FreeBSD v3. .... *72*

Figure 16. AspectC code for sequential prefetching in A3. .... *73*

Figure 17: A simple extension to Emacs for AspectJ. *90*

Figure 18: Mocked-up example of Emacs tool for AspectC. *91*

Figure 19: Support for compositional views of crosscutting for AspectJ within Eclipse. *91*

Figure 20: Claim structure for the dissertation. *114*

1 Introduction

Operating systems must perform well under an increasingly diverse set of workload demands, atop a widening spectrum of hardware capabilities. But evolving OS code is hard. It involves extending, integrating, optimizing, re-optimizing, and maintaining system functionality. It not only requires understanding the individual concerns within the system, but often their inherently complex interactions.

Modularity aids evolution by providing structure within which developers can better reason about and change the system. It is easier to evolve a system if its parts make fewer assumptions about each other, and if responsibilities are better separated between system elements, as Lampson commented with respect to OS design [Lampson 1983]:

> The designer usually finds himself floundering in a sea of possibilities, unclear about how one choice will limit his freedom to make other choices, or affect the size and performance of the entire system. 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.

But providing a clear division of responsibilities in OS code is hard. A study of OS/360 done in the early 70s [Lehman and Belady 1985] showed that the average 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 modules. Modern commercial operating systems, for example Windows NT with over 45 million lines of code [Salkever 2001], 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 [Vogels 1999]. Engler captured popular sentiment with the observation that restrictions link together OS code, making it a fragile and intricate mess [Engler, Chelf, *et al.* 2000].

Aspect-oriented programming (AOP) [Kiczales, Lamping, *et al.* 1997] is aimed at alleviating modularity problems associated with concerns that are inherently *crosscutting* – no single modular decomposition can localize both the crosscutting concern and the concerns it crosscuts. In a non-AOP implementation, crosscutting concerns require their implementation be present in multiple modules in the system, where the modules implement concerns that interact with the crosscutting concern, or *interacting* concerns. Better modularization of crosscutting concerns requires that their implementation be localized, their internal structure be clear, and their interaction with the parts of the system they crosscut be explicit. By accomplishing these three things, AOP can provide structural support for evolution of both crosscutting and interacting concerns.

1.1 The Thesis

The thesis of this work is that aspect-oriented programming can be used to improve evolvability of OS code by providing better modularity of crosscutting concerns and interacting concerns without harming non-interacting concerns. By better modularity of concerns, we mean that we have better separation of responsibilities among system elements.

This dissertation presents the results of two case studies. The first study shows improved modularity in an aspect-oriented implementation versus the original implementation for three crosscutting concerns and their corresponding interacting concerns in FreeBSD [Lehey 1999]. This case study also shows that non-interacting concerns are unharmed in the AOP implementation. The second case study shows that the improved modularity persists across three versions of FreeBSD. We infer from these results that AOP can be used to improve evolvability of the three crosscutting concerns and their interacting

concerns in three versions of FreeBSD, without harming non-interacting concerns. We then generalize these modularity benefits to more than three crosscutting concerns and to more than the three versions of FreeBSD, and to more systems, and infer that this work supports our main claim – that aspect-oriented programming can be used to improve evolvability of operating system code by providing better modularity of crosscutting concerns and their interacting concerns, without harming non-interacting concerns.

We begin by surveying previous work to establish perspectives on structure and change from the software engineering, programming language and systems communities (Section 1.2). We then more precisely identify the problem addressed by this thesis and the structure of our argument (Section 1.3), provide background information on AOP using AspectC (Section 1.4), and overview the chapters in the dissertation (Section 1.5).

1.2 Related Work: Perspectives on Structure and Change

This dissertation claims that a programming language technology – AOP – can improve evolvability of systems code by improving a software engineering property – modularity. The survey provided here first overviews previous work related to structure and change from each of the programming language, operating system, and software engineering communities, then closes with a discussion of modularity in systems code, identifying criteria we will use later in the qualitative assessment of the aspects we propose.

1.2.1 A Programming Language Perspective

ALGOL 60 (ALGOrithmic language) [Backus, Bauer, *et al.* 1960] set a standard for block structure as we know it today. It supported branching, looping, delimited scope of variables, pass by value, pass by name, and recursion. Soon after, Simula67 (SIMULAtion language) [Dahl and Nygaard 1966] provided linguistic support for object-oriented programming, and CLU (function CLUsters) [Liskov and Zilles 1974, Liskov 1992] provided linguistic support for data abstraction. Whereas Simula supported encapsulation if programmers obeyed rules, CLU offered further language enforcement,

contributing to a key idea in programming methodology from this same era that focused on *separation of concerns* [Dijkstra 1976], organizing systems into separate parts that could be dealt with in relative isolation. Though the idea of what precisely constitutes a concern remains somewhat nebulous even today [Murphy, Lai, *et al.* 2001], linguistic support for modules as a collection of operations with hidden information separating of the *what* from the *how* was standard early along in languages such as C [Kernighan and Ritchie 1978, Kernighan and Ritchie 1988], Modula-2 [Wirth 1985], and ML [MacQueen 1986], which all supported library modules with separate compilation.

Smalltalk [Ingalls 1978, Kay 1993], developed at roughly the same time as CLU, had early support for what was later called Metaobject protocols [Kiczales, Rivieres, *et al.* 1991, Kiczales 1992]. Based on previous work on reflection [Smith 1982, Maes 1987], metaobjects enabled dynamic manipulation of methods or types in a program. This approach offered a powerful way of making system-wide, crosscutting changes by facilitating the modification of the language of implementation. Open Implementation [Kiczales 1996] allowed clients of a module to influence its implementation by use of a metaobject, accessed through a separate module interface. The degree to which a module could be influenced was preordained, as it was encoded in the module itself.

Structuring implementations along dimensions that continue to go beyond standard procedural or object-oriented technology has been addressed by several research projects in the Aspect-Oriented Software Development research community [AOSD.net 2003], such as subject-oriented programming, adaptive programming, AspectJ, composition filters and implicit context are briefly overviewed here.

Subject-oriented programming [Harrison and Ossher 1993, Ossher, Harrison, *et al.* 1994] and subsequent work on Hyperspaces [Tarr 1999] deal with collections of classes that define a view of a domain, and provide a means of integrating these multiple views for the development of complex systems.

Adaptive Programming [Palsberg, Xiao, *et al.* 1995] provides a special graph language for structural traversal specifications. Components encode program structure in a way that allows their position in the final program to be inferred based on traversal information.

Aspect-oriented programming [Kiczales, Lamping, *et al.* 1997], as defined by work in the AspectJ project [Kiczales, Hilsdale, *et al.* 2001], provides linguistic support for concerns that are inherently *crosscutting* – by their very nature they are present in more than one module. The premise of this approach is that some concerns dictate a natural primary modular decomposition of a system, whereas others, called *aspects*, crosscut this structure. The goal is to better separate and modularize crosscutting functionality from the primary decomposition of the system using simple linguistic mechanisms.

Composition filters [Aksit and Tekinerdogan 1998] 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. In this approach, objects are wrapped inside of filters identified at the level of class definitions and have access to messages objects receive.

Implicit context [Walker and Murphy 2000, Walker 2003] separates extraneous embedded knowledge from otherwise independent world views within modules. Special constructs called boundary maps support interception and alteration of communication between modules.

1.2.2 A Systems Perspective

The advantage of structure is well known and has been appreciated within the systems community for over 35 years. Hierarchical or a layered structure, which enforces independence of higher levels of the system from lower levels, has been recognized as key since at least the THE multiprogramming system [Dijkstra 1968] in the 60s. End-to-end arguments [Saltzer, Reed, *et al.* 1984, Reed, Saltzer, *et al.* 1998] 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 [Kane 1988], 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 [Lampson 1983]. For an example of offering too much, he points to Pilot [Redell, Dalal, *et al.* 1979], 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 [Lampson and Sproull 1979] and Interlisp-D [Burton, Masinter, *et al.* 1981]. 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 [Lee and Thekkath 1996, Lee and Tekkath 1997] compared with the single layer approach in xFS [Anderson, Dahlin, *et al.* 1996] 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. The xFS file system was 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 [Hutchinson and Peterson 1991] 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 prevented 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 [Valhalia 1996].

The need to support flexibility in operating system services was first addressed by attempts to separate policy and mechanism in Hydra [Wulf, Cohen, *et al.* 1974]. 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 [Engler and Kaashoek 1994, Engler and Kaashoek 1995, Engler, Kaashoek, *et al.* 1995, Kaashoek, Engler, *et al.* 1997], 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 [Veitch 1998]. Similar in-kernel support for coarse grain flexibility has been addressed by the Unix *VFS/Vnode* architecture [Kleiman 1986] which supports multiple file system types.

Projects demonstrating the advantages of making particular policies flexible include paging in Mach [McNamee and Armstrong 1990], scheduling through activations [Anderson, Bershad, *et al.* 1992], communication with active networks [Tennenhouse and Wetherall 1996, Reed, Saltzer, *et al.* 1998], and prefetching and caching strategies with parameterized or split-level policy modules such as LRU-SP [Cao, Felten, *et al.* 1994, Edwards and Cao 1996, Cao 1997, Glass and Cao 1997]. Efforts to customize policy through reflection also ranged theoretically from all OS policy in Apertos [Yokote 1992], to file system services [Maeda and Bershad 1993, Maeda 1994, Maeda 1997] and virtual memory management [Krueger, Loftesness, *et al.* 1993] 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 [Druschel,

Peterson, *et al.* 1992], SPIN [Bershad, Savage, *et al.* 1996, Fiuczynski and Bershad 1996], Vino [Small and Seltzer 1996], and Kea [Veitch and Hutchinson 1996]. A key motivating factor driving research in this area was an application's need for incremental customization [Druschel 1993, Kiczales, Lamping, *et al.* 1993], 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 hard to manage due to the extensive semantic properties involved [Savage 1999]. A similar problem was encountered in Kea, where the inability to structure a clean interface for policy-related elements of services inhibited extensibility [Veitch 1999]. These extensible systems thus highlighted the fact that incremental customization was an elusive goal. 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 [Druschel, Pai, *et al.* 1997].

1.2.3 A Software Engineering Perspective

Heated debates over structured programming raged for more than 20 years, from Dykstra's, "GOTO Considered Harmful" in the late 60s [Dijkstra 1968], to Rubin's "'GOTO Considered Harmful' Considered Harmful" in the late 80s [Rubin 1987]. Though style and taste still trigger much debate, the use of structured control

mechanisms and separation of concerns according to functional decomposition and information hiding are accepted ways of supporting modularity – they make code easier to understand and maintain.

Breaking a system into modules requires some criteria for decomposition. Parnas originally 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 [Parnas 1972]. Additional criteria for good modularity included support for comprehensibility and independent development. More recent work along these lines includes fine-grained modularity for isolation of design decisions [VanHilst and Notkin 1996].

Stevens et al. [Stevens, Meyers, *et al.* 1974] were the first to describe inter-modular and intra-modular characteristics in terms of coupling, the degree of dependence between modules, and cohesion, the strength of the relationship between elements in a module. Though intuitive measures of good and bad modularity can be established on these grounds, objective measures are still hard to define [Emerson 1984, Lakhotia 1993, Bieman and Ott 1994, Bieman and Kang 1998]. No studies have convincingly shown that one programming methodology, for example a modular structure based on objects, is vastly superior to all others. Widespread adoption, however, would indicate that it works well in practice and delivers the expected benefits of good modularity – increased efficiency of the programming process and improvements in the quality of the product [Liskov 1992].

The software engineering community has repeatedly demonstrated that structure plays a key role in determining the cost of change [Parnas 1972, Dijkstra 1976, Murphy 1996] and that it tends to decay over time due to increasing dependencies between modules, further impairing evolution [Stevens, Meyers, *et al.* 1974, Lehman and Belady 1985]. Structural deficiency results in the need for non-local changes that require considerable effort associated with non-local reasoning [Wulf and Shaw 1973, Parnas and Clements 1990].

1.2.4 Modularity and OS Code

With respect to modularity, in many ways OS code is a testament to good, time-honoured, structure. But this survey identifies that complex dependencies between modules – whether they be manifested as unintentional interactions, patterns of interaction, or restrictions that link together the system – indicate that modularity problems still exist, and pose problems for evolution.

Lampson's legendary hints for system designers [Lampson 1983] provide fundamental guiding principles for good modularity within systems code. He notes that designing system interfaces is particularly difficult because of the tension between wanting to hide things that can change, as Parnas suggested, and wanting to expose access to power, a criteria particularly important in performance sensitive systems. Since system interfaces that tried to take on too much typically proved to be too complicated and slow, the ability to capture the minimum essentials of an abstraction is key in OS code.

Using AOP, we propose to improve system modularity by better separating both functionality and specification of interaction of the crosscutting concerns from the modules they crosscut. As a result of this separation, the modules become closer to what Lampson refers to as their minimum essential implementation, and the internal representation within the crosscutting concerns can be made more consistent. By then composing the crosscutting concerns with these modules using language mechanisms that leverage both dynamic execution context and structural information, access to power or shared state can be explicitly coordinated. Subsequent changes that involve either the functionality or the interaction of the crosscutting concern can be made within one module, the aspect, allowing localized change for system-wide configurability.

We believe that extending the notion of good modularity to crosscutting concerns within OS code thus includes improved internal consistency, explicit coordination, and system-wide configurability. We return to these criteria when we evaluate modularity for crosscutting concerns in Section 3.2.4.

1.3 Argument Structure and Experimental Set up

The argument structure used in this dissertation is overviewed in Figure 1. Our case studies use three versions of FreeBSD: v2.2.8 (v2), released December 1998, v3.3 (v3), released September 1999, and v4.4 (v4), released September 2001, and three originally non-modular crosscutting concerns. For each of these FreeBSD versions, v2, v3 and v4, we have a corresponding versions with an aspect-oriented implementation of the crosscutting concerns in kernels we call *A2*, *A3* and *A4*, respectively, while the original implementations are called *O2*, *O3* and *O4*, as shown at the bottom of Figure 1.

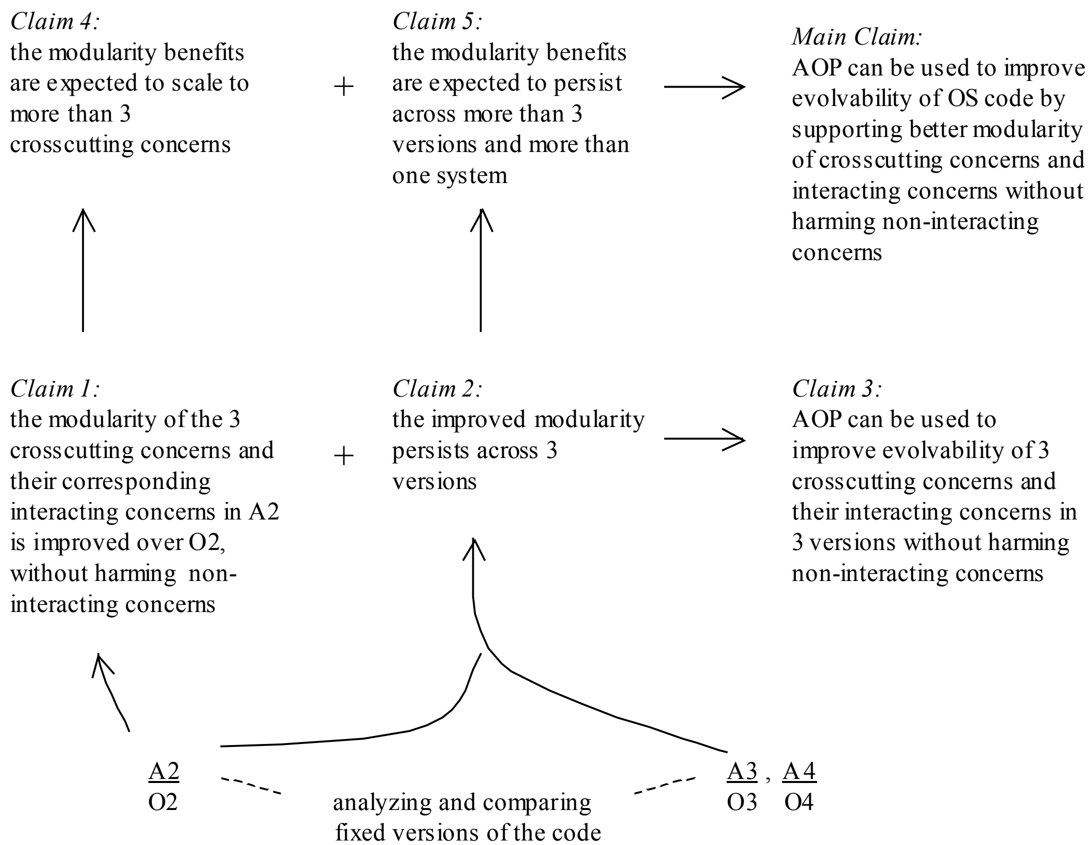

Figure 1: Claim structure for the dissertation.

FreeBSD is representative of a high quality implementation of an operating system, based on its design lineage in the research community and successful adoption in industry. FreeBSD is a mature code base, rooted in a strong design, and a direct descendent of

4.4BSD-lite [Berkeley 1994]. Now considered the most widely-distributed UNIX-based operating systems [Apple.com 2003], Mac OS X uses many core features of FreeBSD 4.4.

The three crosscutting concerns we use in the case studies are: page daemon activation, mapped-file prefetching, and disk quota. We determined these concerns to be crosscutting by aligning their design intent with the functional decomposition of FreeBSD. Section 2.1 elaborates on this by showing the properties we determined to be associated with these crosscutting concerns and their interacting concerns.

1.4 Background: AOP using AspectC

AOP supports modular implementation of crosscutting concerns by allowing fragments of code that would otherwise be spread across several modular units to be co-located and to share context within an aspect. The design of AspectC is directly derived from the non-object-oriented mechanisms in AspectJ [Kiczales, Hilsdale, *et al.* 2001].

1.4.1 Introduction to AOP

AOP is the most recent step in the evolution of programming language technologies designed to better separate concerns. When concerns are better separated, they are easier to reason about and work with, both independently and compositionally. Whereas object-oriented programming provides linguistic mechanisms designed to support the structure and modularity of object hierarchies, aspect-oriented programming provides linguistic mechanisms to support the structure and modularity of crosscutting concerns called *aspects* – concerns that naturally cut across primary modular units of a system.

The primary unit of modularity in object-oriented languages, such as Java, is the class. AspectJ is a simple extension that adds aspect-oriented programming capabilities to Java. In AspectJ, aspects structure and modularize concerns that crosscut classes. The primary unit of modularity in a programming language such as C, however, is not as well defined. In the case of operating system code, it can be argued that units of modularity

range in granularity from individual functions, to files, and further to subdirectories, representing operations, components, and subsystems respectively. AspectC [UBC 2002] is a simple extension that adds aspect-oriented programming capabilities to C. In AspectC, aspects structure and modularize concerns that crosscut functions, files and directories.

1.4.2 Language Design of AspectC

The design of AspectC is directly derived from the non-object-oriented mechanisms in AspectJ. As such, AspectC can be described in terms of 4 key mechanisms: *join points* are points in the execution of a program; *pointcuts* pick out specific join points and values at those points; *advice*[1] define actions to take at join points in a pointcut; and *aspects* modularize crosscutting behaviour in terms of pointcuts, advice, and standard C constructs, such as functions and structs.

Join points are points in a dynamic call graph. Our AspectC prototype currently supports two types of join points: *function call* and *function execution*. For any given function, its call join points correspond to all the points during the execution of the system when the function is explicitly called (i.e., not accessed through a function pointer), whereas its execution join points correspond to all the points during the execution of the system when the function body runs. Join points are dynamic in that they can be defined in terms of the control flow of other join points.

Pointcuts are a means of identifying and selecting join points. They can be composed and used anonymously or given names, and can extract values from join points. Pointcuts act as filters against the set of all join points and select a subset through matching. Matching takes place in terms of function signatures, control flow, lexical extents, and combinations, as detailed in Table 1.

---

[1] Advice will be used as a plural noun throughout this thesis.

Table 1: Matching of Pointcut Designators.

<table>
<tr><th>Syntax</th><th>Kind of join point</th><th>Additional criteria</th></tr>
<tr><td>`execution`(*Signature*)</td><td>execution</td><td>function execution join points where the function signature matches *Signature*</td></tr>
<tr><td>`call`(*Signature*)</td><td>call</td><td>function call join points where the function signature matches *Signature*</td></tr>
<tr><td>`cflow`(*Pointcut*)</td><td>any</td><td>all join points in the control flow of the join points specified by *Pointcut*</td></tr>
<tr><td>`within`(*File or Directory*)</td><td>any</td><td>join points when the code executing is defined in one of the files found in *File* or *Directory*</td></tr>
<tr><td>! *Pointcut*</td><td>any</td><td>join points that are not picked out by *Pointcut*</td></tr>
<tr><td>*Pointcut0* && *Pointcut1*</td><td>any</td><td>join points that are picked out by both *Pointcut0* and *Pointcut1*</td></tr>
<tr><td>*Pointcut0* || *Pointcut1*</td><td>any</td><td>join points that are picked out by either *Pointcut0* or *Pointcut1*</td></tr>
</table>

Pointcuts identify context, or "when" in terms of the execution of the primary modularity. Specifically, different kinds of context are exposed by different forms of pointcuts, as outlined in the Table 2.

Table 2: Kinds of context.

Kind of Context	Form of Pointcut
Static local	execution, call
Static immediate caller	*Pointcut* && within(..)
Dynamic	*Pointcut* && cflow(..)
Dynamic values	*Pointcut* && cflow(<..>(<value>,..))

Advice defines additional code to execute at join points. Advice can execute either when the execution reaches a join point and *before* proceeding, *after* returning from a join point, or *around* a join point, in which case advice gets explicit control over when and if a program proceeds with execution at the join point.

Both after and around advice introduce additional special keywords. After advice introduces a special variable, *returned*, through which after advice may access the return value of a function. Around advice introduces a special function, *proceed*, which explicitly requests execution of whatever would have run if the around advice had not been defined. Around advice can control the values of parameters passed onto *proceed.*

Aspects structure and modularize crosscutting concerns using pointcuts, advice, and standard C constructs. By way of an example, consider a simplified version of an aspect for sequential mode prefetching in Figure 2[2]. Here, we briefly overview the structure of the aspect but postpone detailed analysis of its content. Overall, only a small portion of our implementation of prefetching relies on the AspectC extensions to structure the crosscutting within the aspect, the rest is ordinary C code from the original implementation. This implementation should be considered as a refactoring [Opdyke 1992, Tokuda and Batory 1999, Fowler, Brant, *et al.* 2000] using AspectC. Refactoring is considered as a meaning-preserving transformation that restructures existing functionality in a system.

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_path*, with one *vm_map_t* parameter, *map*. The second line of this declaration provides the details. This pointcut refers to all function calls in the control flow of the execution of *vm_fault*, and picks out *vm_fault's* first argument. By using the name *map* in the function signature and using it again in the pointcut parameters, it says that the first argument of the function is published as the first argument of the pointcut. The ".." in this parameter list means that although there are more parameters in this list, they are not picked out by this pointcut.

---

[2] The full implementation, where parameter lists are not abbreviated, can be found in Figure 16.

The second declaration is another pointcut, named *ffs_read_path*, which allows advice in the aspect to access the vnode pointer parameter, *vp*, passed into *ffs_read*.

The before advice 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 executions of *vnode_pager_getpages*, and to give the body access to the *map* parameter of the surrounding executions of *vm_fault*.

Reading the header in more detail, the first line says that this advice will run *before* function executions designated following the ":", and lists five parameters available in the body of the advice. The second line specifies calls to the function *vnode_pager_get_pages*, and picks up the four arguments to that function. The third line uses the previously declared pointcut vm_fault_path, 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 of the advice is ordinary C code. The helper function *plan_and_alloc_sequential_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 around advice diverts the execution path to *ffs_read* when access is sequential, or to *proceed* with *ffs_getpages* otherwise.

The after advice executes under the control flow of the pointcuts *ffs_read_path* and *vm_fault_path*. That is, it executes only when control flow has been diverted along this special path from *vm_fault* to *ffs_read*. This after advice is responsible for ensuring that additional costs are not incurred when transferring the data from the buffer cache to the allocated VM pages. Since this is a special case page-aligned transfer, copying can be avoided by simply reassigning or flipping allocated VM pages with the appropriate file buffer cache pages.

In summary, this prefetching aspect uses pointcuts that cut across multiple subsystems – virtual memory, the virtual file system interface, and the FFS file system – to provide a structured and modular implementation for sequential mode prefetching for mapped files.

```
  aspect sequential_prefetching_for_mapped_files {

    pointcut vm_fault_path( vm_map_t map ):
        cflow(execution( int vm_fault( map, .. )));

    pointcut ffs_read_path( struct vnode* vp ):
        cflow(execution ( int ffs_read( vp, .. )));

    before( vm_map_t map, vm_object_t obj, vm_page_t* plist, int len, int fpage ):
        execution ( int vnode_pager_getpages( obj, plist, len, fpage ))
        && vm_fault_path( 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 ):
        execution( int ffs_getpages( obj, plist, len, fpage ))
    {
      int error;
      if ( obj->behaviour == SEQUENTIAL ) {
        struct uio* io_inf = io_prep( plist[fpage]->pindex, MAXBSIZE, curproc );
        error = ffs_read( obj->handle, io_inf, MAXBSIZE, curproc->p_ucred );
      } else
        error = proceed( obj, plist, len, fpage );
      return error;
    }

    after( struct uio* io_info, int size, struct buf** bpp ):
        execution( int bread(..) )
        && vm_fault_path(..)
        && ffs_read_path( struct vnode*, io_info, size, bpp )
    {
        flip_buffer_pages_to_allocated_vm_pages( (char *)bpp->b_data, size, io_info );
    }
 }
```

Figure 2: Partially simplified code for an example sequential prefetching aspect.

1.4.3 Design of AspectC Prototype Implementation

There is an intimate connection between the C programming language and the C preprocessor (Cpp), particularly in legacy systems where macros are used as efficient alternatives to eliminate reliance on compiler optimizations. In order to support this, and carry it forward to the relationship between AspectC and Cpp, AspectC compilation consists of four major steps: (1) a first pass with Cpp performing macro expansion,

leaving all macro definitions and other *#define* directives in place, (2) a parsing pass and a weaving pass with the AspectC compiler, (3) a second pass with Cpp to expand macros in advice, and (4) compilation with a standard C compiler.

1.4.3.1 Cpp Passes

The preliminary pass with Cpp means that the AspectC compiler only has to weave pure C code. Though it might be possible to avoid this first pass, some C programs cannot be parsed as Cpp can be used to define new syntax, and much of the preprocessor functionality would have to be re-implemented in the AspectC compiler.

AspectC semantics were defined so that advice bodies are macro expanded in the lexical context of the join point, so that advice can use local Cpp definitions when appropriate. For example, it is possible for an advice body to use a macro that is defined differently within different files. This is implemented by having them copied into place at the shadow of the join point. This is the case with many macros in file system code. These advice bodies must not be preprocessed until they are placed in the correct lexical context. To avoid prematurely expanding macros in advice bodies, advice that will be copied to matching join point locations is shielded from expansion by being temporarily replaced with a unique identifier. After weaving, the unique identifier is replaced with the original advice body, and the second pass of Cpp expands any macros within these bodies.

1.4.3.2 Parsing and Weaving

Parsing and weaving are done in two separate passes. When files are parsed, all advice, pointcuts and join points are collected. The parser was built using ANTLR, ANother Tool for Language Recognition[Parr 2003], and a GNU C parser written by John Mitchell and Monty Zukowski [Mitchell and Zukowski 2003]. We added the ability to recognize AspectC constructs and syntax to this parser, along with the various collectors. The

weaver then traverses the code, examining each join point to determine if there is advice with a pointcut that matches the join point.

Source transformations made during weaving are join point specific. Call join points replace all calls to the original function with a call to a stub function. For example, given the sample input file:

```
aspect a {
  before(): call(void foo()) {
    printf("before");
  }
}

void foo() {
   printf("foo");
}

void caller_of_foo() {
   printf("caller of foo");
   foo();
}
```

The weaver replaces all calls to *foo,* in this case in the function *caller_of_foo,* with calls to a stub function as follows:

```
void ACPP_a_BEFORE_1 ( ) {
  printf ( "before" ) ;
}

void foo ( ) {
  printf ( "foo" ) ;
}

void caller_of_foo ( ) {
  printf ( "caller of foo" ) ;
  ACPP_BEFORE_AFTER_CALL_foo_1 ( ) ;
}

void ACPP_BEFORE_AFTER_CALL_foo_1 ( ) {
  if ( 1 ) ACPP_a_BEFORE_1 ( ) ;
  foo ( ) ;
  return ;
}
```

Execution join points replace the body of the original function with a call to a stub function, moving the original body to a new function noted to be the original. For example, if *aspect a* was modified as follows:

```
aspect a {
  before(): execution(void foo()) {
    printf("before");
  }
}
```

The weaver replaces the body of *foo* as follows:

```
void ACPP_a_BEFORE_1 ( ) {
  printf ( "before" ) ;
}

void foo ( ) {
  ACPP_BEFORE_AFTER_EXEC_foo_1 ( ) ;
}

void ACPP_BEFORE_AFTER_EXEC_foo_1 ( ) {
  if ( 1 ) ACPP_c_BEFORE_1 ( ) ;
  REAL_EXEC_foo1 ( ) ;
  return ;
}

void REAL_EXEC_foo1 ( ) {
  printf ( "foo" ) ;
}

void caller_of_foo ( ) {
  printf ( "caller of foo" ) ;
  foo ( ) ;
}
```

As previously mentioned, during weaving advice bodies are copied to the same lexical context as the join points they match. As the examples above demonstrate, they appear as newly generated C functions, positioned before the function they advise.

An overview of weaver output for dynamic pointcut designators, and how management for control flow information is currently handled in the AspectC prototype, is discussed further in Section 4.3.

1.4.4 Runtime Architecture Interface

In the case of dynamic join points, the state that is pushed on the stack during *cflow* is a vector of pointers to the parameters involved. Since *cflow* pointcuts are tied to the dynamic control flow of the program, the state of the join points selected by these pointcuts needs to be tracked during program execution. All individual cflow pointcut designators are assigned a unique cflowID, used to identify an associated stack for pushing and popping of join point state. Definitions associated with AspectC's runtime interface are outlined in Table 3.

Table 3: Definitions associated with AspectC runtime support.

cflowID	*int*, globally unique identifier of cflow pointcut
void cflow_push(int cflowID, void **vals)	pushes the parameter vector onto the cflowID stack
void cflow_pop(cflowID)	pops the last parameter vector off the cflowID stack
int cflow_test(cflowID)	if there exists an entry on the cflowID stack returns 1, otherwise returns 0
void **cflow_get(cflowID)	returns the parameter vector of the topmost entry on the cflowID stack

1.4.5 *Integrating AspectC Runtime with the FreeBSD Kernel*

AspectC runtime support for *cflow* is integrated into the FreeBSD kernel. The runtime data structure is an open hash table. A pool of entries is statically allocated, and is sufficiently large to track the maximum number of processes in the system. Each entry tracks *cflow* information for a single process and is uniquely identified by the process ID allocated by the kernel.

Initialization of the runtime support happens when the kernel boots, setting up the hash table and the free queue. In this initial state, all entries are chained together in a free list. When a process is created, an entry is allocated from the free queue and inserted into the hash table. When a process exits, the entry is returned to the free queue.

The integration of the AspectC runtime with the kernel is crosscutting. It intersects the system at a point in the startup, and then again at process creation and process exit. Due to the crosscutting nature of this implementation, it seemed only fitting to implement it as an aspect, shown in Figure 3. This aspect has three advice: (1) runtime initialization, attached to a low level startup function, *main* in versions 2 and 3, but most functionality was moved to a new function, *mi_startup* in v4, (2) creation of per process *cflow* data strutures, attached to *make_proc_table_entry*, a helper function we refactored from *fork*, and (3) destruction of these data structures, attached to *exit.*

```
aspect acruntime {

  /* allocate space for runtime state */
  before(void):
    execution(void main())
  {
    cflow_initialize_runtime();
  }

  /* give a process its own cflow stacks */
  before(struct proc *p):
    execution(void make_proc_table_entry(struct proc*, p))
  {
    cflow_add_pid_entry(p->p_pid);
  }

  /* take back a cflow stack */
  before(struct proc *p, int rv):
    execution(void exit1(p, rv))
  {
    cflow_del_pid_entry(p->p_pid);
  }

}
```

Figure 3: Integration of the AspectC runtime library with the FreeBSD kernel.

We should note here that FreeBSD offers kernel support, *at_fork* and *at_exit*, for registering handlers at process creation and exit points in the system. It was necessary for us to use our own mechanism however, to allow dynamic advice to work on any of the execution paths these routines use. For example, both *at_fork* and *at_exit* use *malloc*, so if we had used *at_fork* to add an entry and *at_exit* to remove it, we would not have been able to put any advice that uses *cflow* on the *malloc* path.

Though this is not representative of an optimal implementation, as costs could be further reduced by both inlining these functions and storing *cflow* state directly within process data structures to eliminate the hash table lookup, Section 4.3 demonstrates that the overheads of even this naive implementation are not prohibitive for the crosscutting concerns considered here.

1.5 Thesis Overview

Chapter 2 presents a case study comparing the modularity of an aspect-oriented implementation of the three concerns, A2, with the original implementation, O2. We begin by establishing that the three concerns are crosscutting, and representative of other

crosscutting concerns. We then show that modularity of the three crosscutting concerns and their corresponding interacting concerns – the concerns they crosscut – is improved in the aspect-oriented implementation, and that non-interacting concerns are unharmed (Claim 1).

Chapter 3 presents a case study comparing the modularity of these same concerns in two subsequent versions of the system, A3 versus O3, and A4 versus O4, and the locality of change between the implementations. We begin by establishing that changes that we identified between versions are representative of the kinds of changes that impact concerns during evolution. We then show that the locality of change of the three crosscutting concerns and their corresponding interacting concerns is improved, and show that the improvements in modularity persist across the three versions (Claim 2).

Chapter 4 presents inferences regarding evolution and scale. First with respect to evolution, by which we mean the software developer work of analyzing, planning and making the changes, we infer that AOP can be used to improve evolvability of the three crosscutting concerns and their interacting concerns without harming non-interacting concerns (Claim 3). We then generalize to more concerns (Claim 4), more versions, and more systems (Claim 5), in support of our final inference, that these improvements in modularity can make evolution of OS code easier in an AOP implementation.

Chapter 5 provides a summary, conclusions and future work.

2 Case Study One: Single Version Modularity Analysis

This chapter begins with some background information on the structure of the FreeBSD virtual memory system, pager interface and file system interface. It is from within these subsystems that we extract our three crosscutting concerns: page daemon activation, mapped file prefetching and disk quota. For each concern, we analyze its implementation in O2 (FreeBSD v2, original code). In each case, we show that the crosscutting concern is *scattered* – present in more than one module – and *tangled* – mixed in with code that addresses more than one concern, and discuss the consequences of this non-modular implementation in the original kernel. We then present an alternative implementation of each concern in the aspect-oriented kernel, A2, and identify qualitative improvements in modularity on a case-by-case basis. We conclude with a discussion about the modularity of crosscutting, interacting and non-interacting concerns in A2 relative to O2, in order to more comprehensively address the validation of the following claim:

> ***CLAIM 1:*** *the modularity of the 3 crosscutting concerns and their corresponding interacting concerns in A2 is improved over O2, without harming non-interacting concerns.*

2.1 Case Study Background

The virtual memory (VM) system in FreeBSD dates back to Mach [Rashid, Tevanian, *et al.* 1987]. In this model, VM supports protected address spaces for applications into which *VM objects* can be mapped. VM objects include files and anonymous memory allocated during program execution. Each VM object is responsible for maintaining state about its own VM pages resident in memory. The contents of physical memory is essentially a cache of recently used pages from these VM objects, managed by a global page replacement algorithm that replaces the oldest, or least recently used page first. Four page queues are used to organize the contents of physical memory: the active, inactive, cache and free queues. Modified pages on the inactive queue have not been written to disk, whereas pages on the cache queue have been written to disk and are available for immediate reuse if necessary. Access to a page that is on the cache queue results in a low cost page fault, not serviced by disk.

Each VM object is associated with a *pager* that handles transferring pages between physical memory and backing store, such as disk. Pagers and file systems are structured according to an object hierarchy, where a form of polymorphic dispatch is facilitated through the use of function tables and macros. FreeBSD v2 supports four different pagers, as outlined in Figure 4. The *swap* and *default* pagers handle anonymous memory, the *vnode* pager handles memory backed by a file, and the *device* pager handles memory that maps to hardware. A pager instance is created at the same time as the VM object when a piece of anonymous memory, a file, or a device is mapped into a process address space, and continues to exist until the VM object is de-allocated.

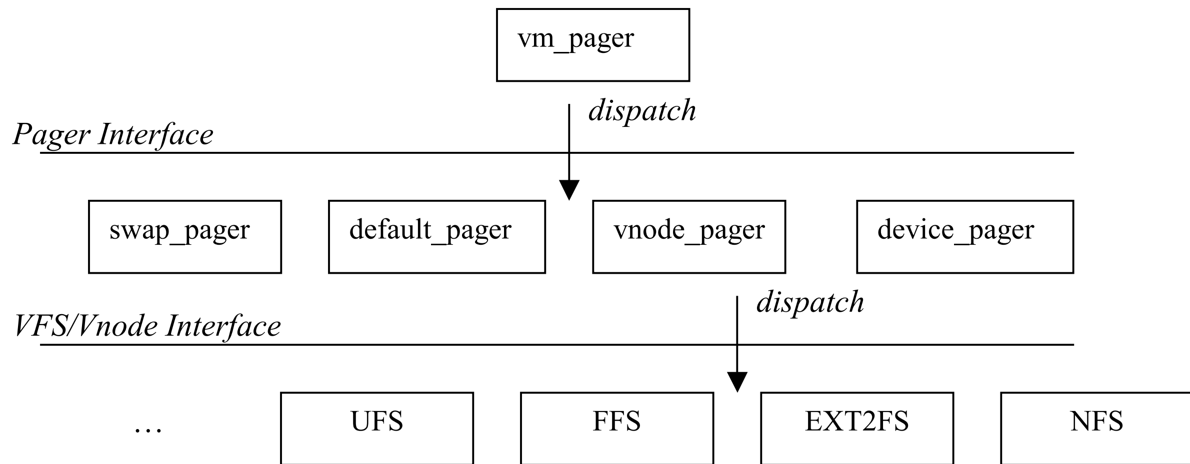

Figure 4: Pager and file system hierarchy in FreeBSD.

The case studies for this thesis use three crosscutting concerns: page daemon activation, mapped-file prefetching, and disk quota. We determined these concerns to be crosscutting by aligning their design intent with the functional decomposition of FreeBSD. This analysis involved exploring their implementation primarily by manual inspection using *grep*, and runtime diagnostics with a kernel debugger and our own *printf* statements.

In terms of design intent, we used high level descriptions for each as follows: the page daemon is activated to free up memory when the system is running low on available pages; mapped file pages are prefetched to amortize costs when faulted pages are retrieved from disk, and disk quotas track and restrict file system operations when disk blocks are released or consumed. We found that, in each case, the concern is engaged in multiple parts of the functional decomposition of the system, where engaged means a variable having to do with the concern is set, a value is passed on, or lines of code having to do with the concern are executed.

Having established that crosscutting concerns exist, we can consider any given concern in the system to be one of 3 types: concerns with the properties described above are crosscutting concerns, concerns that are crosscut are interacting concerns, and concerns

not crosscut are non-interacting concerns. In particular, Table 4 outlines each of these types for the three concerns in our case studies.

Table 4: Crosscutting, interacting and non-interacting concerns.

crosscutting	interacting	non-interacting
page daemon activation	VM, pagers and file buffer cache	the rest of the system
mapped-file prefetching	VM, pagers, file buffer cache and file systems	the rest of the system
disk quotas	file systems: UFS, FFS, EXT2FS	the rest of the system

Identifying the lines of code in the scattered and tangled implementation that constitute the crosscutting concern versus the interacting concern was fairly straightforward given the three crosscutting concerns considered here. Though interacting concerns may constitute further crosscutting concerns that may themselves merit further separation, for the sake of clarity, we consider interacting concerns only at the granularity of the primary modularity. That is, interacting concerns in our case are subsystems, such as VM, file buffer cache, and file systems.

We believe the three crosscutting concerns in our studies are representative for two reasons. First, they provide fundamental system elements that pose real and not idiosyncratic challenges to implement comprehensively in a traditional system. Second, we have identified several other concerns that exhibit these same properties and challenges, as discussed further in Chapter 4.

2.2 Original Implementation: O2

This section presents the implementation of each of the three concerns in the original version of FreeBSD v2, a kernel we call O2. For each concern, we start with a brief introduction, then an analysis of its scattered and tangled implementation, and conclude with a discussion regarding the consequences of its non-modular implementation.

2.2.1 *Page Daemon Activation*

The page daemon frees physical memory when the system is running low on available memory. Since the daemon imposes overhead for determining which pages will be replaced and for writing them back to disk if necessary, timing is important – part of the intent of activation is to ensure that the daemon is woken only when necessary.

2.2.1.1 *Analysis of O2 Implementation*

Page daemon activation crosscuts points during the execution of the system when additional available pages are, or could soon be, needed. An overview of this scattered implementation across operations from 4 files, *swap_pager.c*, *vm_fault.c*, *vm_page.c* and *vfs_bio.c*, is shown in a partial FreeBSD source tree in Figure 5. This Figure highlights both the subsystems – Swap and VM in the *vm* subdirectory, and the file buffer cache in the *kern* subdirectory – and the functions involved. For example, in Swap, *swap_pager.c*, 4 functions activate the daemon, and the other 22 functions in the file do not. Likewise, in the file buffer cache, *vfs_bio.c*, 1 of 50 functions activate the daemon.

As its name indicates, *vm_page_unqueue_nowakeup* in *vm_page.c* is the only exception in this list, and does not activate the daemon. We include it here to show that, in the case of the unqueue operation, the caller decides if activation is appropriate by choosing which version of the function to invoke. This means that the callers themselves are part of the interacting concerns and, though they are not shown in Table 5, they are discussed further in Section 2.3.1. The functions *vm_page_unqueue* and *vm_page_unqueue_nowakeup* are identical except for daemon activation. Table 5 details the scattering of this concern in terms of its 17 occurrences across the 9 functions from the 4 files involved.

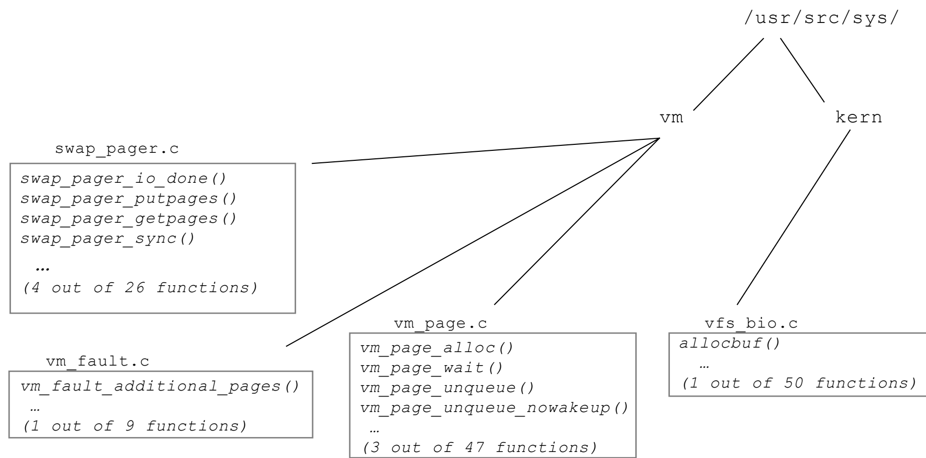

Figure 5: Modules and functions involved page daemon activation in O2.

Table 5: Places in the system where the page daemon is activated in O2.

File	Occurrences	Host Functions
vm/vm_fault.c	1	vm_fault_additional_pages
vm/swap_pager.c	7	swap_pager_io_done swap_pager_putpages swap_pager_getpages swap_pager_sync
vm/vm_page.c	8	vm_page_alloc vm_page_unqueue vm_wait
kern/vfs_bio.c	1	allocbuf
Total: 4	*17*	*9*

Activation is associated with paging activity that results in the consumption of available pages. In most cases, waking the daemon involves assessing system state and calling the function that primarily controls activation, *pagedaemon_wakeup*. This small function is defined in *vm_pageout.c*, along with the implementation of the rest of the page daemon, as follows:

```
void
pagedaemon_wakeup()
{
        if (!vm_pages_needed && curproc != pageproc) {
                vm_pages_needed++;
                wakeup(&vm_pages_needed);
        }
}
```

The daemon sleeps on the address of *vm_pages_needed*, so the call to *wakeup(&vm_pages_needed)* makes the daemon runnable. We found one instance of *wakeup(&vm_pages_needed)*, outside of *pagedaemon_wakeup*, in *vm_wait* as shown below:

```
void
vm_wait()
{
        int s;
        s = splvm();
        if (curproc == pageproc) {
                vm_pageout_pages_needed = 1;
                tsleep(&vm_pageout_pages_needed, PSWP, "vmwait", 0);
        } else {
                if (!vm_pages_needed) {
                        vm_pages_needed++;
                        wakeup(&vm_pages_needed);
                }
                tsleep(&cnt.v_free_count, PVM, "vmwait", 0);
        }
        splx(s);
}
```

Conditions under which *pagedaemon_wakeup* is called can involve a threshold assessment, consisting of both local and global state. To get an appreciation of the variation of thresholds involved, Figure 6 shows invocations of *pagedaemon_wakeup* from three different files: *vm_page.c*, *vm_fault.c* and *vfs_bio.c*.

In the function *vm_page_alloc* in *vm_page.c* for example, the first two invocations to *pagedaemon_wakeup* precede a *return(NULL)*, or a failed attempt to allocate a page, and the third invocation involves global counters for virtual memory, *cnt.v_cache_count, cnt.v_free_count, cnt.v_free_reserved* and *cnt.v_cache_min*. In the function *vm_fault_additional_pages* in the file *vm_fault.c*, the values *rahead* and *rbehind* are local state, passed into that function, used in the threshold calculation.

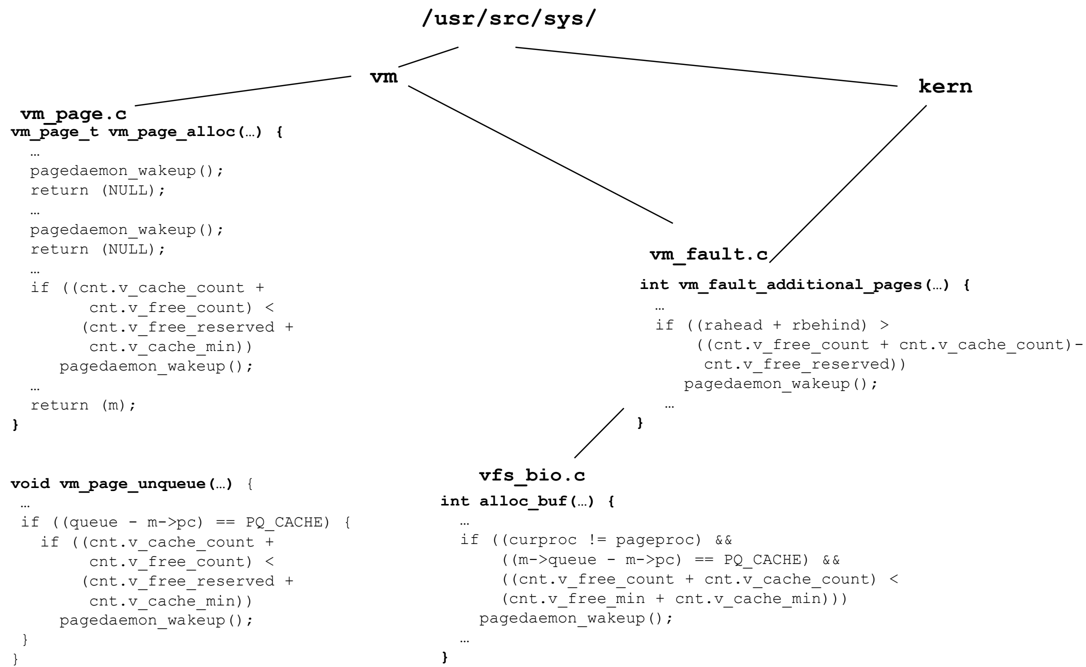

Figure 6: Analyzing scattered code for page daemon activation in O2.

2.2.1.2 *Consequences of Scattered and Tangled Implementation*

Identifying precisely when and why the page daemon may be activated is difficult because it is scattered throughout the code, and at times tangled with otherwise local state involved. The rationale behind the different thresholds is not immediately apparent to a non-expert. For example, the use of *cnt.v_free_min* within the file buffer cache versus *cnt.v_free_reserved* within VM. Additionally, it is not easy to comprehensively document these subtleties given that its implementation is scattered.

The allowable interface to activation is also relatively wide, in that activation does not always procede through *pagedaemon_wakeup*. Specifically, low level activation, such as in *vm_wait*, actually reduces the potential to coordinate through what might otherwise be a standard interface, *pagedaemon_wakeup*.

2.2.2 *Mapped-File Prefetching*

The intent of prefetching is to hide disk latency by bringing in additional pages that may be required in the near future. As a heuristic, an important part of the intent of prefetching is to carry out the plan only if it is cost effective to do so. The virtual memory system thus suggests pages for prefetching, but the file system decides whether or not to actually get them, combining predictions about what additional data is likely to be needed in the future with a analysis of what additional data would be most cost-effective to fetch at any given time.

2.2.2.1 *Analysis of O2 Implementation*

Prefetching crosscuts most execution paths that read from disk. Figure 7 overviews some of the scattered implementation of prefetching in terms of a partial source tree, and Table 6 summarizes this implementation.

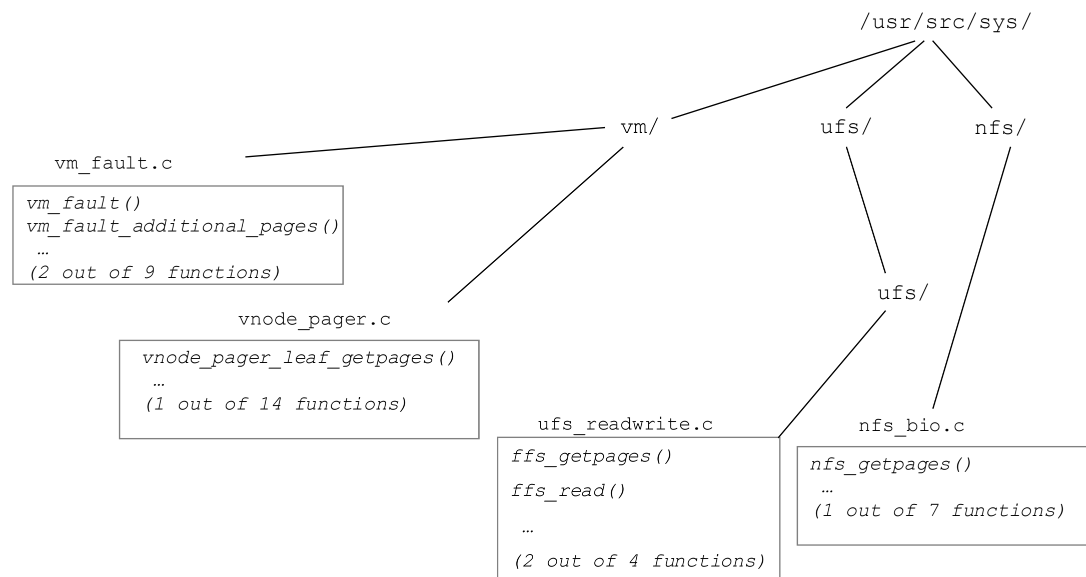

Figure 7: Scattered implementation of prefetching in O2.

Table 6: Files involved with prefetching functionality in O2.

File	Occurrences	Host Functions
vm/vm_fault.c	2	vm_fault vm_fault_additional_pages
ufs/ufs/ufs_readwrite.c	3	ffs_getpages
vm/vnode_pager.c	3	vnode_pager_leaf_getpages
nfs/nfs_bio.c	2	nfs_getpages
Totals: 4	*10*	*5*

When a file is mapped into virtual memory, it becomes a VM object, accessed as ordinary virtual memory, and faulted into memory as needed. That is, a page fault occurs when a process references a virtual address that is not resident and, over time, pages in the mapped file are demand-paged into memory.

In the absence of prefetching, handling a page fault is fairly straightforward. It is well supported by a layered architecture in which the virtual memory system is a client of the file system, which is in turn a client of the disk driver. A page fault starts in the virtual memory (VM) system as a request for a page associated with a VM object; it moves through to the local file system, for example FFS, and is translated into a block-based request associated with a file; it finally passes to the disk system where it is accessed in terms of a cylinder, head, and sectors. The division of responsibilities between these three layers is centered around the management of their respective representations of data. That is, the functionality within each layer primarily deals with controlling resources in terms of its own set of abstractions.

Upon close inspection, prefetching ultimately involves four key activities: prediction, cost analysis, planning and actually fetching the pages. Prediction requires context at the origin of the request to predict future requests. Future requests are anticipated to follow the pattern of past requests, such as sequential versus random access. Cost analysis involves lower level factors such as the cost of disk access, contiguity of data on disk, and the destination of the data. These are used to determine which disk blocks can be prefetched in the most cost-effective way. Planning involves combining prediction and

cost analysis information to determine which data should actually be fetched from disk, and can include decisions such as whether the prefetch should be synchronous as part of the current read, or asynchronous. Actually fetching the data normally just involves executing the plans, but there are cases where disk or other system state may change between planning and fetching, which can cause plans to be changed or cancelled.

In FreeBSD v2, VM objects have a declared access behaviour, which can be set to *random*, *normal* or *sequential* using the *madvise* system call. Prediction for objects with normal (the default) behaviour follows a simple locality heuristic that identifies addresses close to the faulted address (+/- a specific window) as more likely to be used next. Similarly, sequential behaviour identifies address following the faulted address. Cost analysis for bringing these pages in then looks at both available memory and contiguity on disk. Random behaviour dictates that prefetching would be fruitless.

2.2.2.2 *Consequences of Scattered and Tangled Implementation*

The most significant consequence of the current non-modular implementation of prefetching is that it is difficult to track its behaviour through the code. Even just identifying all the prefetching code in v2 takes a significant amount of work[3]. Consistency is compromised because prefetching is poorly localized – tracing the page-fault path described here for mapped files in v2 requires traversing 5 files, 2 levels of function tables, and 4 changes in identifier names, including repackaging of parameters into data structures.

In particular, *dynamic context passing*, when parameters are passed from high level functions down through lower level functions purely in service of prefetching and not the interacting concerns it crosscuts and *layering violations*, places where VM and file system

---

[3] Our inspiration to explore an AOP approach to prefetching is largely due to observing an experienced system programmer in our lab devote several days to tracking down all the sources of page allocation and de-allocation in the code.

abstractions are simultaneously accessed, complicate the code that prefetching crosscuts. Cost analysis inherently involves dynamic context passing and layer violations, since it looks at both VM layer information (available memory) and disk layer information (contiguity). Combining cost analysis with prediction is another source of layer violations. By the time the execution path reaches the file system layer, system state may have changed in a way that invalidates the prefetching plan. If the faulted page has become valid, the faulted page is no longer on disk, or the planned pages are no longer contiguous, the file system layer must de-allocate memory for virtual pages it decides not to fetch. This is an additional source of context passing and layer violations, since the file system layer must access the VM page map.

2.2.3 Disk Quota

Quota must track disk utilization, enforcing limits when appropriate for all users and groups. Quota introduces the need to monitor disk consumption, to abort otherwise legal file system operations if consumption would exceed quota, and to roll back consumption tracking in the event a file system operation fails for some other lower level reason.

2.2.3.1 Analysis of O2 Implementation

Quota crosscuts all operations that consume/release disk blocks. Figure 8 and Table 7 overview quota's scattered implementation across UFS, FFS and EXT2. Quotas are an optional feature in FreeBSD, configured through a combination of settings in both a kernel configuration file and on a per-file system basis, consisting of 29 compiler directives as outlined in terms of files and host functions in Table 7.

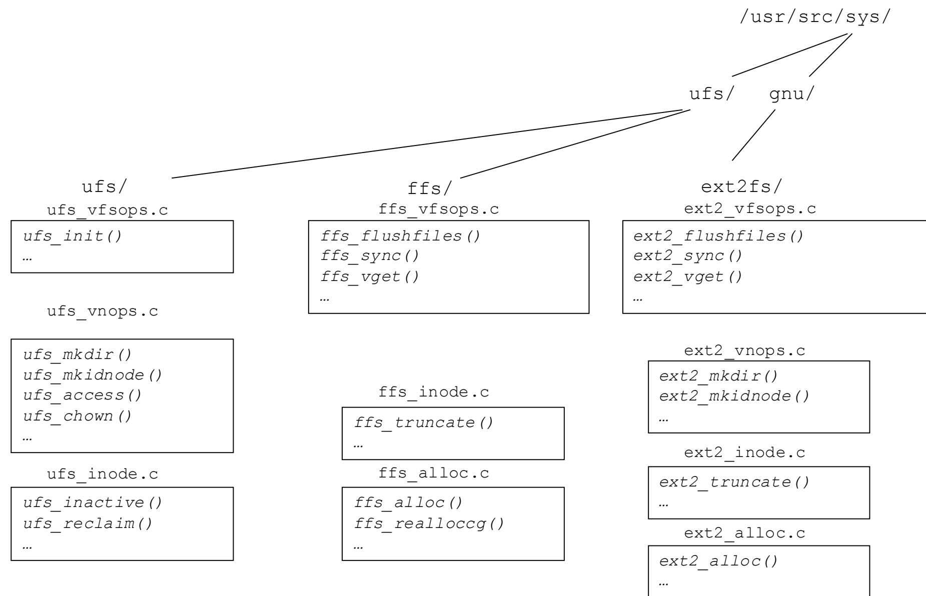

Figure 8: Scattering of quota implementation across UFS, FFS and EXT2FS.

Table 7: Files involved in quota functionality in O2.

File	Occurrences	Host Functions
ufs/ufs/ufs_vnops.c	7	ufs_access ufs_chown ufs_mkdir ufs_mkinode
ufs/ufs/ufs _inode.c	3	ufs_inactive ufs_reclaim
ufs/ffs/ffs _vfsops.c	3	ffs_flushfiles ffs_sync ffs_vget
ufs/ffs/ffs _alloc.c	5	ffs_alloc ffs_realloccg
ufs/ffs/ffs _inode.c	2	ffs_truncate
gnu/ext2fs/ext2_vfsops.c	4	ext2_flushfiles ext2_sync ext2_vget
gnu/ext2fs/ext2_alloc.c	3	ext2_alloc
gnu/ext2fs/ext2_inode.c	2	ext2_truncate
Total	*29*	*17*

2.2.3.2 *Consequences of Scattered and Tangled Implementation*

The major consequence of the current non-modular implementation of quota is that it is difficult to see quota as a whole, and identify the structural relationships that hold across the file systems. Further, redundant quota code creates the possibility for drift to occur between portions of quota code that should be identical.

The nature of drift we found did not appear to introduce anything of functional significance in quota. For example as we have previously seen, the quota code embedded in *flushfile* operations from FFS and EXT2, from two different files, respectively, is shown in Figure 9. The differences in the declaration of the variable and the assignment of the error condition are a merely matter of style, whereas the use of *#if* versus *#ifdef* is actually incorrect.

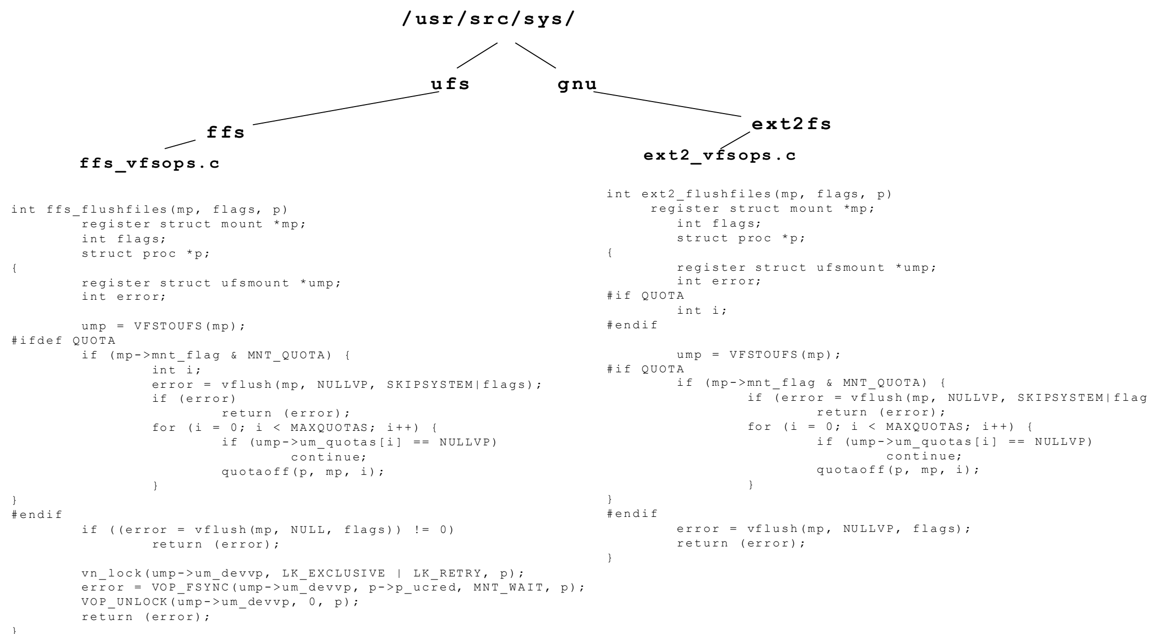

Figure 9: Implementation of Quota in *flushfile* operations from FFS and EXT2FS.

2.3 Aspect-Oriented Implementation: A2

This section presents a refactoring of the three crosscutting concerns in FreeBSD v2 using AspectC. To differentiate the aspect-oriented implementation from the original, we call the kernels A2 versus O2 respectively. In each case, we present possible variations of the implementations presented in AspectC, and discuss the merits of the alternatives.

2.3.1 Page Daemon Activation as an Aspect

The structure of page daemon activation consists of three things: the points in the system where available pages could be needed, the set of thresholds that determine if the daemon should be activated, and the relationship between two, such that the right threshold is applied in the right place at the right time. Available pages can become scarce under several different execution contexts in the system, spanning activity associated with Swap, VM and the file buffer cache. Page daemon activation thus crosscuts these different parts of the system, which we consider its interacting concerns within the primary modularity.

What we want to accomplish by modularizing page daemon activation as an aspect is to provide a locus of control. By localizing both functionality and external interaction, the subtleties of the thresholds gain significance by proximity, without losing context.

The Figure 10 highlights points in the execution of the system in which the page daemon may be activated, system-wide. We focus on one segment of this aspect, specifically its association with VM, points labelled 1-5 in Figure 10.

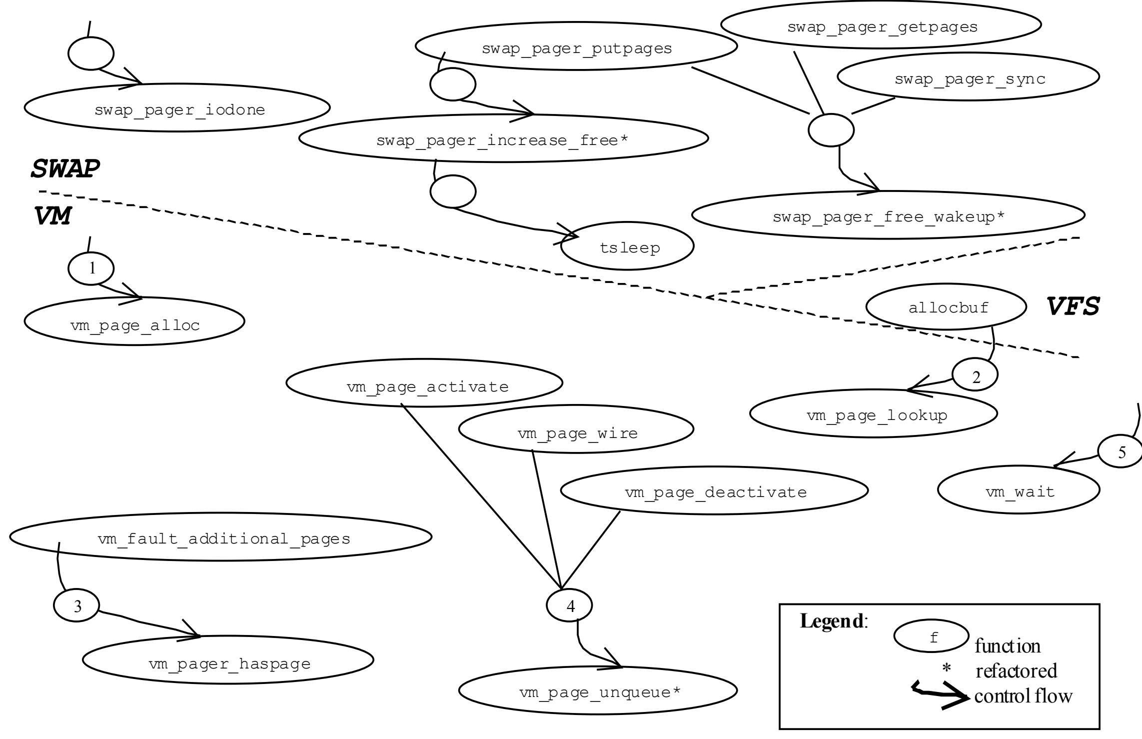

Figure 10: Contexts for calls to page daemon wakeup in O2.

In AspectC, we can use pointcuts to expose context during the execution of VM operations and use advice to attach thresholds appropriately. The portion of an aspect that defines named pointcuts for the contexts that crosscut VM and the file buffer cache is shown below, annotated with the corresponding numbers from Figure 10.

```
aspect pageout_daemon_wakeup {

(1)  pointcut allocate_pages(vm_object_t object, vm_pindex_t pindex, int page_req):
       execution(vm_page_t vm_page_alloc(object, pindex, page_req));

(2)  pointcut allocate_buffers(vm_object_t obj, vm_pindex_t pindex):
       execution(vm_page_t vm_page_lookup(obj, pindex))
       && cflow(execution(int allocbuf(struct buf*, int)));

(3)  pointcut fault(int rbehind, int rahead):
       execution(boolean_t vm_pager_has_page(vm_object_t, vm_pindex_t, int*, int*))
       && cflow(execution(int vm_fault_additional_pages(vm_page_t, rbehind, rahead,
                                                        vm_page_t*, int*)));

(4)  pointcut unqueue(vm_page_t m):
       execution(void vm_page_unqueue(m))
       && !cflow(execution(void vm_page_alloc(vm_page_t))
               || execution(void vm_page_free_toq(vm_page_t))
               || execution(void vm_page_cache(vm_page_t, int)));

… /* more to come */
```

These pointcuts not only name 4 execution contexts, but also expose values associated with each context.

The pointcut labelled 1, *allocate_pages*, exposes 3 parameters associated with the execution of the function *vm_page_alloc.* Just to review the syntax, the name of the pointcut is associated with a parameter list, and reading after the ":", is the definition of the pointcut in terms of more primitive pointcuts.

The pointcut labelled 2, *allocate_buffers,* is associated with the execution point originating in the file buffer cache, as identified by the *cflow* within the execution of *allocbuf*. The values exposed by this pointcut are those that are passed to *vm_page_lookup*.

The pointcut labelled 3, *fault*, exposes 2 parameters, and these parameters are dynamic values exposed from higher up the execution path. That is, *rbehind* and *rahead* are the values passed into *vm_fault_additional_pages*, making them available when *vm_pager_has_page* executes the control flow of that function.

The pointcut labelled 4, *unqueue,* exposes a parameter passed into *vm_page_unqueue*. Note that this pointcut specifies all executions of *vm_page_unqueue* that are not in the *cflow* (!cflow) of the 3 functions specified.

Advice using these pointcuts, and the values they expose, are shown below.

```
1  around(vm_object_t object, vm_pindex_t pindex, int page_req):
       allocate_pages(object, pindex, page_req) {
     vm_page_t allocd_page = proceed(object, pindex, page_req);
     if (allocd_page == NULL)
       pagedaemon_wakeup();
     else
       if (cnt.v_free_count + cnt.v_cache_count < cnt.v_free_reserved + cnt.v_cache_min)
         pagedaemon_wakeup();
     return allocd_page;
   }


   around(vm_object_t obj, vm_pindex_t pindex): allocate_buffers(obj, pindex) {
2    vm_page_t m = proceed(obj, pindex);
     if ((m != NULL) && !(m->flags & PG_BUSY)
         && ((m->queue - m->pc) == PQ_CACHE)
         && (cnt.v_free_count + cnt.v_cache_count < cnt.v_free_min + cnt.v_cache_min))
       pagedaemon_wakeup();
     return m;
   }

   after(int rbehind, int rahead): fault(rbehind, rahead) {
3    if ((rahead + rbehind) >
         (cnt.v_free_count + cnt.v_cache_count - cnt.v_free_reserved))
       pagedaemon_wakeup();
   }

   around(vm_page_t m): unqueue(m) {
4    int queue = m->queue;
     proceed(m);
     if (((queue - m->pc) == PQ_CACHE) &&
         (cnt.v_free_count + cnt.v_cache_count < cnt.v_free_reserved + cnt.v_cache_min))
       pagedaemon_wakeup();
   }
```

The 3 around advice all start by proceeding with the join point, and then accessing the return value from there[4]. For example, the first around advice activates the daemon if the allocation of a page either fails, or the appropriate threshold applies. The after advice uses the parameters as part of the threshold to determine if the daemon should be awoken. The original code from O2 as presented earlier is repeated here in Figure 11 for convenience in viewing this refactoring.

[4] In the current AspectC prototype, the only way to get at a return value is with around advice.

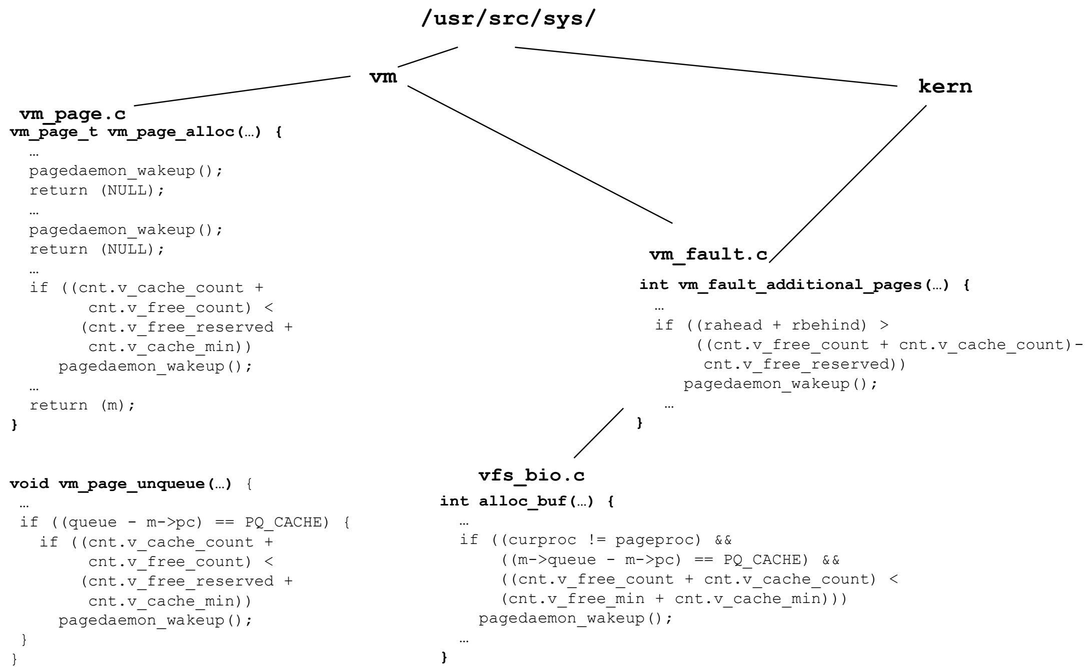

Figure 11: Original code for activation from O2.

We introduced a subtle but important change in A2 versus O2 when we refactored the activation within *vm_wait* to invoke *pagedaemon_wakeup*, enforcing it as a standard interface instead of allowing the original low level wakeup as previously shown, and repeated below.

```
void
vm_wait()
{
        int s;
        s = splvm();
        if (curproc == pageproc) {
                vm_pageout_pages_needed = 1;
                tsleep(&vm_pageout_pages_needed, PSWP, "vmwait", 0);
        } else {
                if (!vm_pages_needed) {
                        vm_pages_needed++;
                        wakeup(&vm_pages_needed);
                }
                tsleep(&cnt.v_free_count, PVM, "vmwait", 0);
        }
        splx(s);
}
```

```
aspect pageout_daemon_wakeup {

/* previously defined pointcuts and advice …*/

  pointcut sleep_in_wait(void* count, int priority, char* fname, int flag):
     execution(int tsleep(count, priority, fname, flag))
      && cflow(execution(void vm_wait()));

(5)  around(void* count, int priority, char* fname, int flag):
    sleep_in_wait(count, priority, fname, flag)
  {
    if (priority == PVM)
       pagedaemon_wakeup();
    return proceed(count, priority, fname, flag);
  }

}
```

This advice uses the value of *priority* to determine whether or not it should run, and only executes if this value indicates its priority is VM and not Swap (PVM versus PSWP)[5].

2.3.1.1 *Improvements in Modularity and Implementation Alternatives*

The page daemon activation aspect improves modularity by providing a locus of control for activation. The impact of this is twofold. First, whereas in O2 the thresholds are scattered, localizing them along with their execution context in A2 makes their relative significance easier to establish by proximity, without losing the necessary context information. Second, coordination through a standard interface can be enforced in A2, such that all activation proceeds through *pagedaemon_wakeup*.

One possible alternative implementation to the page daemon activation aspect presented here would be to redefine the *unqueue* pointcut in terms of the functions that intend to activate the daemon. That is, instead of listing the control flows we do not want activation to accompany, as we have done in the original pointcut:

---

[5] We take this opportunity to note that parameters such as *fname*, which provides the name of the calling function to *tsleep*, and even *priority*, are examples of passing *dynamic context* that might be better handled using *cflow* in an AOP implementation of their associated concerns.

```
pointcut unqueue(vm_page_t m):
    execution(void vm_page_unqueue(m))
    && !cflow(execution(void vm_page_alloc(vm_page_t))
            || execution(void vm_page_free_toq(vm_page_t))
            || execution(void vm_page_cache(vm_page_t, int)));
```

We could instead list the control flows in which we do want activation:

```
pointcut unqueue(vm_page_t m):
    execution(void vm_page_unqueue(m))
    && cflow(execution(void vm_page_activate(vm_page_t))
            || execution(void vm_page_wire(vm_page_t))
            || execution(void vm_page_deactivate(vm_page_t, int)));
```

The reason we chose the former was because we felt it was a more accurate reflection of the true special case, though this is hard to determine as there are equal number of callers to both versions of the original *vm_page_unqueue* and *vm_page_unqueue_nowakeup*, so there was not a clear majority. The ramification of this decision is discussed further in Section 3.2.1. Semantically, perhaps the most accurate way to describe what we want to capture is the movement of pages to the active queue, and the assumption we made is that the unqueue operation generally captures that movement. Though we tried to establish if any higher level callers could improve the semantics of this pointcut, it was not obvious how to simplify them further in the given implementation of the interacting concerns.

2.3.2 *Mapped-file Prefetching as an Aspect*

The most significant structural feature of prefetching is that it accompanies most execution paths to disk. As previously described, in planning prefetching, the low-level cost analysis factors are given more weight than the high-level prediction factors. Prefetching is synchronous and pages that are not contiguous on disk are not prefetched even if they appear within the predicted window[6]. As part of planning however, physical pages are allocated to hold the pages to be prefetched. Because this allocation requires

---

[6] Intuitively this is because the prediction in normal mode access is not strong enough to warrant the risk of additional disk waits.

locking the page map associated with the VM object, and because that page map is already locked in the VM layer as part of preparing to fetch the faulted page, it is advantageous to do planning while execution is still in the VM layer[7]. But it is not until later in the execution path that the low-level cost factors can be established.

What we would like to accomplish by modularizing prefetching as an aspect is to more explicitly structure dynamic context and layer violations along executions paths to disk by introducing it as a *path-specific-customization* [Coady, Kiczales, *et al.* 2001]. The points related to the prefetching aspect are thus designed to capture execution paths where prefetching should apply, and tailor the concern to the path – coordinating high and low level activity that crosscuts the system.

As previously identified, prefetching involves the interaction between four key activities: prediction, cost analysis, planning and actually fetching the pages. At a high-level, the context of the origin of the request is used to predict future requests. This prediction is based on normal, sequential or random access. Cost analysis involves lower level factors such as the cost of disk access, contiguity of data on disk, and the destination of the data. These are used to determine which disk blocks can be prefetched in a cost-effective way. Planning involves combining prediction and cost analysis information to determine which data should actually be fetched from disk. Actually fetching the data normally just involves executing the plans, but there are 3 cases where disk or other system state may change between planning and fetching, which can cause plans to be changed or cancelled.

Figure 11 illustrates the structure of prefetching for objects with normal access patterns. The corresponding aspect-oriented implementation is shown in Figure 13, where labels 1 through 4 correspond to the four advice in the aspect. The aspect structures coordination

---

[7] In a personal communication with Daryl Anderson from Duke University, it was established that he had experienced a problem with deadlock when lowering this VM locking along the execution path. The bug took days to identify.

between the high-level allocation for prefetched pages and their possible subsequent low-level de-allocation, further down the execution path. The significance of the dispatching across layers in the system is that these functions are not called directly, but instead accessed through tables of function pointers. Though specifying call sites for such functions is not possible in AspectC, their execution sites are available as join points.

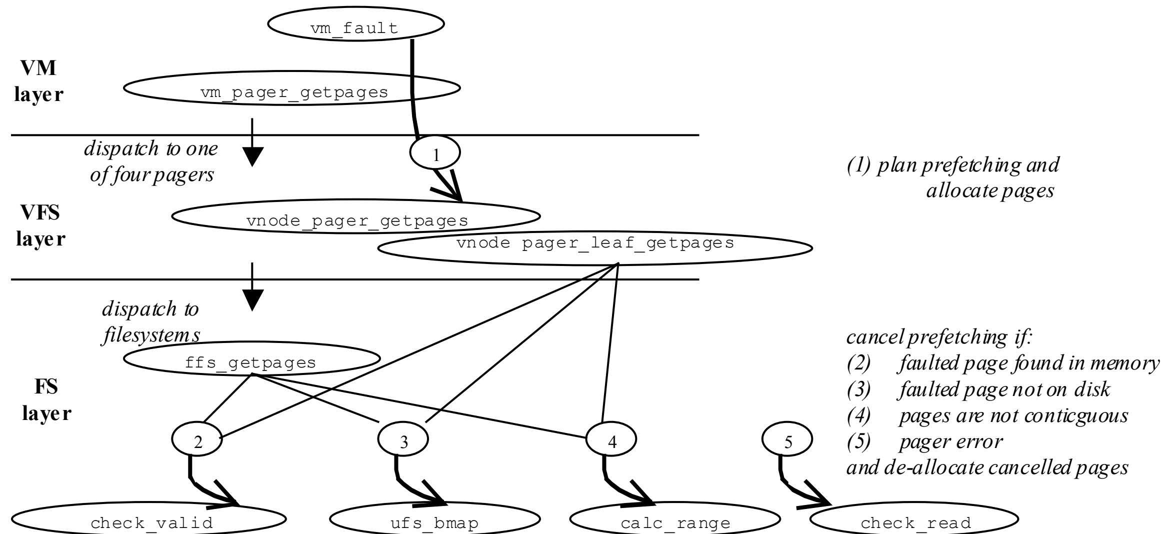

Figure 12: Prefetching for mapped files in FreeBSD v2.

The first two pointcuts name the high-level and the low-level parts of the execution paths involved, *fault_path* and *getpages_path* respectively, exposing the necessary parameters at each point. Here, the *getpages_path* demonstrates how multiple file systems can be handled, covering corresponding operations from different file systems, *ffs_getpages* and the generic version of the getpage function for file systems that do not provide their own (such as EXT2), *vnode_pager_leaf_getpages*, respectively. The advice then coordinate allocation/de-allocation along these control flows – the first advice allocates pages for prefetching, while the next four advice may de-allocate pages if it is no longer cost effective to retrieve them.

```
aspect mapped_file_prefetching {

   pointcut fault_path(vm_map_t map):
       cflow(execution(int vm_fault(map, vm_offset_t, vm_prot_t, int)));

   pointcut getpages_path(vm_object_t object, vm_page_t* pagelist, int length,
                          int faulted_page):
       cflow(execution(int ffs_getpages(object, pagelist, length, faulted_page))
             || execution(int vnode_pager_leaf_getpages(object, pagelist, length,
                                                        faulted_page);

   pointcut pager_error(int error, int uio_resid, struct vop_getpages_args* ap):
       execution(int check_read(error, uio_resid, ap));

(1) before(vm_map_t map, vm_object_t object, vm_page_t* pagelist, int length,
          int faulted_page):
   execution(int vnode_pager_getpages(object, pagelist, length, faulted_page))
   && fault_path(map) {
     if (object->declared_behaviour != RANDOM) {
       vm_map_lock(map);
       plan_and_alloc_prefetch_pages(object, pagelist, length, faulted_page);
       vm_map_unlock(map);
     }
   }

(2) around(vm_object_t object, vm_page_t* pagelist, int length, int faulted_page,
          int valid):
   execution(int ffs_valid(..))
   && getpages_path(object, pagelist, length, faulted_page){
     int valid = proceed();
     if (valid)
       dealloc_all_prefetch_pages(object, pagelist, length, faulted_page);
   }

(3) around(vm_object_t object, vm_page_t* pagelist, int length, int faulted_page,
          struct vnode *vp, ufs_daddr_t bn, struct vnode **vpp,
          ufs_daddr_t *bnp, int *runp, int *runb):
   execution(int ufs_bmap(vp, bn, vpp, bnp, runp, runb))
   && getpages_path(object, pagelist, length, faulted_page){
     int error = proceed(vp, bn, vpp, bnp, runp, runb);
     if (error || (*bnp == -1))
       dealloc_all_prefetch_pages(object, pagelist, length, faulted_page);
   }

(4) after(vm_object_t object, vm_page_t* pagelist, int length, int faulted_page,
         struct trans_args* t_args ):
   execution(int ffs_calc_size(t_args))
   && getpages_path(object, pagelist, length, faulted_page){
     dealloc_noncontig_prefetch_pages(object, pagelist, length, faulted_page, t_args);
   }

(5) around(int error, int uio_resid, struct vop_getpages_args* ap):
   pager_error(error, uio_resid, ap) {
     int i, success, npages;
     success = proceed(error, uio_resid, ap);
     if (!success) {
       npages = btoc(ap->a_count);
       for (i = 0; i < npages; ++i) {
         if (i != ap->a_reqpage)
           vnode_pager_freepage(ap->a_m[i]);
       }
     }
     return success;
   }

 }
```

Figure 13: Aspect for normal mode prefetching.

Looking at each advice in a little more detail, the advice labelled (1) 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*. 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, labelled (2) through (5), implement the four conditions under which the file system layer can choose not to prefetch. In each case, the implementation of the decision not to prefetch results in de-allocation.

The advice labelled (2) 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 advice labelled (3) 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_path* 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 advice labelled (4) 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_path* and

calc_range, and de-allocates any pages that were originally requested but not within the actual range that will be transferred.

The final advice, labelled (5), frees pages everytime *check_read* is not successful. This helper function, *check_read*, was refactored from *nfs_getpages*. Possible alternatives for its pointcut declaration are discussed further below.

2.3.2.1 Improvements in Modularity and Implementation Alternatives

Implementing prefetching as an aspect improves the modularity of this crosscutting concern by making its interaction with execution paths explicit, by providing consistency of naming within its functionality, and by making high-level and low-level coordination clear. The modularity of interacting concerns is simultaneously improved by better separation of dynamic context passing and layer violations.

One of the implementation alternatives we encountered when designing the prefetching aspect included the optional use of *cflow* to restrict execution context. When we factored the *check_read* helper function out of *nfs_getpages*, and attached de-allocation advice to this helper, we did not restrict the pointcut to apply only within the *cflow* of *nfs_getpages*, though no other function calls the helper in this version of the system. The ramifications of this decision have direct impact on how prefetching scales to new file systems, as discussed further in Section 3.2.2.

2.3.3 Disk Quota as an Aspect

The quota aspect is structured as a centralized set of operations for monitoring and restricting disk usage, simultaneously applied to corresponding file system operations from UFS, EXT2 and FFS that consume/release disk blocks. Figure 14 shows the contexts in which quota is activated within UFS, FFS and EXT2FS. We begin by looking at the segment of this aspect associated with the VFS layer, and show how consistent application of quota functionality can be applied across FFS and EXT2.

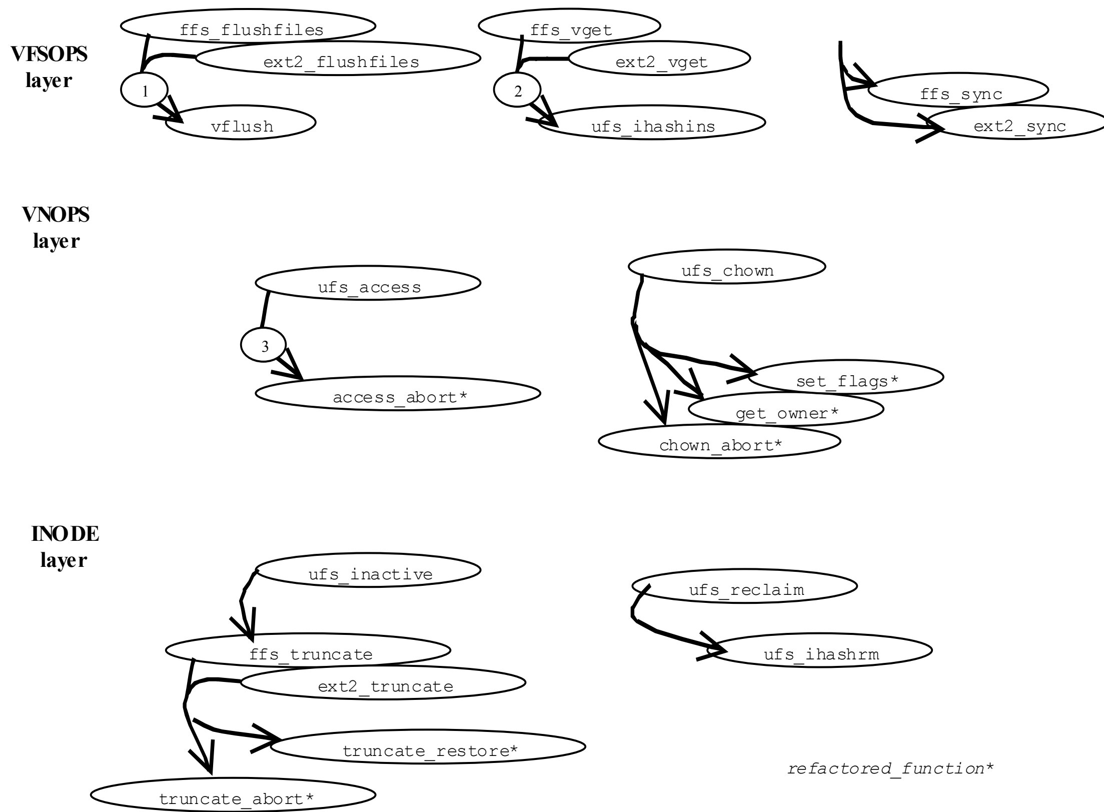

Figure 14: A subset of contexts for quota aspect in UFS, FFS and EXT2FS.

Specifically, 5 compiler directives in FFS and EXT2 associated with execution points labelled (1) and (2) from O2 are replaced by the portion of the aspect shown below. The pointcuts name the corresponding operations from the different file systems that are associated with shared quota operations. For example, the first pointcut, *flushfiles*, is associated with the flushfile operation in either FFS or EXT2. The around advice that uses this pointcut provides a single shared implementation of the associated quota operation.

```
aspect disk_quota {

  pointcut flushfiles(register struct mount *mp, int flags, struct proc *p):
    execution(int ffs_flushfiles(mp, flags, p))
    || execution(int ext2_flushfiles(mp, flags, p));

  pointcut vget(struct inode *ip):
    execution(void ufs_ihashins(ip))
    && cflow(execution(int ffs_vget(struct mount*, ino_t, struct vnode**)))
             || (execution(int ext2_vget(struct mount*, ino_t, struct vnode**)));

  around(register struct mount *mp, int flags, struct proc *p):
(1)  flushfiles(mp, flags, p)
  {
    register struct ufsmount *ump;
    ump = VFSTOUFS(mp);
    if (mp->mnt_flag & MNT_QUOTA) {
      int i;
      int error = vflush(mp, NULLVP, SKIPSYSTEM|flags);
      if (error)
        return (error);
      for (i = 0; i < MAXQUOTAS; i++) {
        if (ump->um_quotas[i] == NULLVP)
          continue;
        quotaoff(p, mp, i);
      }
    }
    return proceed(mp, flags, p);
  }

  before(struct inode *ip):
(2)  vget(ip)
  {
    int i;
    for (i = 0; i < MAXQUOTAS; i++)
      ip->i_dquot[i] = NODQUOT;
  }
 /* more disk quota aspect… */
}
```

The around advice demonstrates how, in the event of an error condition associated with quota, the advice may not proceed with the originally advised function. In the case of *flushfiles* this works well, as the function is aborted first thing. Functions in the VNOPs and INODE layers however, require further refactoring as abort takes place later in a function.

To facilitate more general aborting of functions based on quota conditions, we factored out *_abort* helper functions to which the quota abort advice attaches. For example, the *ufs_access* function starts with the following code:

```
int
ufs_access(ap)
        struct vop_access_args *ap;
{
        struct vnode *vp = ap->a_vp;
        …
#ifdef QUOTA
        int error;
#endif

        if (mode & VWRITE) {
                switch (vp->v_type) {
                case VDIR:
                case VLNK:
                case VREG:
                        if (vp->v_mount->mnt_flag & MNT_RDONLY)
                                return (EROFS);
#ifdef QUOTA
                        if (error = getinoquota(ip))
                                return (error);
#endif
                        break;
                }
        }
…
```

from which we refactored the *if(mode & VWRITE)* into a helper function, *access_abort*, called as follows:

```
int
ufs_access(ap)
        struct vop_access_args *ap;
{
        …
        if (error = ufs_access_abort(mode, vp))
                return (error);
```

and the quota aspect uses around advice, associated with the execution point labelled 3 in Figure 14, to piggyback its possible abort on this helper function in UFS:

```
aspect disk_quota {

…

  pointcut access_abort(struct vnode *vp, mode_t m):
    execution(int ufs_access_abort(vp, m));


(3) around(struct vnode *vp, mode_t m):
    access_abort(vp, m)
  {
    int error = proceed(vp, m));
    if (!error)
       error = getinoquota(VTOI(vp));
    return error;
  }
…
}
```

2.3.3.1 *Improvements in Modularity and Implementation Alternatives*

By making quota an aspect instead of introducing it with compiler directives, we can still unplug its implementation by leaving it out of the kernel's Makefile. The improvements in modularity are achieved through explicit representation of structural relationships that hold across file system operations, enforcing consistent application across parallel functionality, and the elimination of redundant code.

Implementation alternatives encountered during the design of the quota aspect include, as with prefetching, the optional use of *cflow* to restrict execution context. When possible, operations that must restore quota consumption due to a low level failure call the same helper function, to which quota restoration is attached. As is the case with prefetching, this decision has ramifications as the system evolves, as discussed further in Chapter 3.

2.4 Validation

The purpose of this case study was to provide validation for the following claim:

> ***CLAIM 1:*** *the modularity of the 3 crosscutting concerns and their corresponding interacting concerns in A2 is improved over O2, without harming non-interacting concerns.*

The analysis we performed was a qualitative assessment of the modularity of two implementations of the same three crosscutting concerns: the original implementation, O2, versus the aspect-oriented implementation, A2.

In O2, the implementation of the three crosscutting concerns are non-modular, that is, scattered and tangled, and their presence introduces isolated access to shared state, variability of interfaces used, subsystem violations, dynamic context passing, and redundant code, all compromising the modularity of the crosscutting and interacting concerns.

In A2, the implementation of the three crosscutting concerns has locality, is better separated from the interacting concerns, coordinates access to shared state, enforces narrower interfaces, makes non-local interactions explicit, such as detecting an execution sequence, passing a value down a call chain, or enforcing consistency of policy across parallel functionality, and eliminates redundant code. This change not only improves the modularity of the crosscutting concerns, but simultaneously improves the modularity of the interacting concerns as a result of better separation. Table 8 lists specific improvements in modularity for the concerns involved in each case.

Table 8: Summary of improved modularity in A2.

concerns	O2	A2
crosscutting: page daemon activation, *interacting*: swap, VM, file buffer cache	scattered activation thresholds, isolated access to shared state, wide interface	localized thresholds, coordinated access to shared state (enforced) narrow interface
crosscutting: mapped-file prefetching, *interacting*: VM, file systems	dynamic context passing, layer violations	high level and low level coordination along an execution path
crosscutting: disk quota *interacting*: file systems	redundant code, isolated enforcement across parallel functionality	shared code, shared contexts

With respect to the modularity of non-interacting concerns, we claim it remains constant between O2 and A2. A violation of this would arise if a concern that was non-interacting in O2 became interacting in A2. Potential for this exists if we defined any pointcut incorrectly, capturing more than what we intended to be its execution context. Due to the fact that we have no tool support for the AspectC prototype, we relied primarily on manual inspection of the woven code and testing to ensure this did not occur. In Section 4.2.1, we discuss some state of the art AOP tools available to help support the ability to reason about aspect composition with interacting concerns.

3 Case Study Two: Multi-Version Modularity Analysis

This chapter provides a case study designed to validate the following claim:

***Claim 2**: Improvements in modularity persist across 3 versions.*

The case study presented here provides an comparison of the modularity of page daemon activation, mapped-file prefetching and disk quota, in the original implementation versus the aspect-oriented implementation over time. That is, here we analyze the relative impact of evolution on each implementation as we roll forward into subsequent versions of the system. We start where Case Study One left off, with the three crosscutting concerns in FreeBSD v2, released in December 1998, in their original implementation, O2, and their aspect-oriented implementation, A2. We then consider the impact of evolution on the given modularity of the three crosscutting concerns in O2, versus the given modularity of the concerns in A2, as they roll forward into FreeBSD v3, released in September 1999, and FreeBSD v4, released in September 2001. We do this by comparing the original implementation of the concerns in O3 and O4, with their aspect-oriented implementations, A3 and A4. Each of A2, A3 and A4 has a subtly different implementation of the crosscutting concerns. The Appendix of this dissertation provides an overview of these subtle differences for the page daemon activation aspect.

Through manual inspection of the three versions of the system, we identified 11 specific change tasks that appeared to have the most significant impact on the crosscutting concerns during evolution. Each task is associated with the addition/removal/refactoring of related functionality that impacts one of the crosscutting concerns either directly, by changing the concern itself, or indirectly, by changing an interacting concern in a way that impacts the crosscutting concern.

The change tasks, along with the primary focus of each, are overviewed in Table 9, associated with page daemon activation (PDA), mapped-file prefetching (PREF) or disk quota (QUO), and within a particular evolution interval, from v2→v3, or from v3→v4. By *focus* we mean the functionality primarily associated with a given change, either crosscutting concern, interacting concerns or both.

Table 9: Change tasks overviewed.

change task	from	description	focus
PDA_CT1	v2→v3	Integration of Swap and VM	both
PDA_CT2	v2→v3	New function *vm_page_unmanage*	interacting
PREF_CT1	v2→v3	New sequential mode prefetching	crosscutting
QUO_CT1	v2→v3	New feature in file systems	both
QUO_CT2	v2→v3	New configurability for quotas in *ufs_init*	crosscutting
PDA_CT3	v3→v4	Revamp of *vm_alloc*	interacting
PDA_CT4	v3→v4	New function *vm_await*	both
PDA_CT5	v3→v4	Refactoring in *vm_page.c*	crosscutting
PREF_CT2	v3→v4	Removal of sequential mode prefetching	crosscutting
PREF_CT3	v3→v4	New *getpages* in new file systems	interacting
QUO_CT3	v3→v4	New feature in FFS	interacting

We start this analysis by comparing the locality of change in the aspect-oriented implementation to the original implementation. In short, the results of this study show that in the original implementation: of the 4 modules page daemon activation crosscuts, 2 of them change; of the 6 modules mapped-file prefetching crosscuts, 3 of them change; and of the 11 modules quota crosscuts, 5 of them change. These changes all either

directly or indirectly involve the crosscutting concern. In the aspect-oriented implementation, all changes focused on a crosscutting concern impact only the aspect involved, or one module. Changes to interacting concerns that require the aspect be applied in more or fewer places do not require subsequent modification of the crosscutting concern if they are within the original design intent of the aspect. Changes to non-interacting concerns remain unaffected.

We conclude this mult-version analysis by identifying the ways in which characteristics associated with the (non)modularity persist across the 3 versions in the aspect-oriented versus original implementation. In this analysis, we use the indicators of good modularity identified earlier in Section 1.2.4: consistency of functionality within the crosscutting concern, coordination of interaction of the crosscutting concern, and configurability of the crosscutting concern system-wide.

3.1.1 Page Daemon Activation Evolution

As FreeBSD evolved, the internals of many functions in the Swap subsystem, *swap_pager.c*, were significantly revamped to better integrate Swap with the rest of VM. Among many other changes, the 7 places where the page daemon was activated within Swap were eliminated between FreeBSD v2 and v3, as outlined in Table 10. Between v3 and v4, the reworking of the function to allocate pages, *vm_page_alloc*, trimmed two more places where activation had occurred. Other changes not evident in the Table are the introduction of two new VM functions that have interaction with daemon activation, and the refactoring of certain threshold calculations.

Table 10: Page daemon activation in FreeBSD v2, v3 and v4.

File	Occurrences			Host Functions
	v2	v3	v4	
vm/vm_fault.c	1	1	1	vm_fault_additional_pages
vm/swap_pager.c	7	0	0	swap_pager_io_done swap_pager_putpages swap_pager_getpages swap_pager_sync
vm/vm_page.c	7	7	5	vm_page_unqueue vm_page_alloc vm_wait vm_await (v4 only)
kern/vfs_bio.c	1	1	1	allocbuf
Totals: 4	*17*	*9*	*7*	*v2: 9* · *v3: 5* · *v4: 6*

Looking at the differences between the implementation of page daemon activation in v2 versus v3, and v3 versus v4, we identified five key change tasks associated with page daemon activation between v2 and v4:

PDA_CT1: Better integration of Swap with VM.
Evolution: v2 to v3
Result: Removal of 7 occurrences of activation from Swap.
Modification: *swap_pager.c*

PDA_CT2: Addition of a new VM function, *vm_page_unmanage*.
Evolution: v2 to v3
Result: Introduction of a new call to the unqueue operation that wakes the daemon, *vm_page_unqueue*.
Modification: *vm_page.c*

PDA_CT3: Maintenance of VM function, *vm_alloc*.
Evolution: v3 to v4
Result: Removal of 2 occurrences of activation from within VM.
Modification: *vm_page.c*

PDA_CT4: Addition of a new VM function, *vm_await*.
Evolution: v3 to v4
Result: Additional low level activation of page daemon wakeup.
Modification: *vm_page.c*

PDA_CT5: Addition of new VM helper function, *vm_paging_needed*.
Evolution: v3 to v4
Result: Refactoring of some activation threshold calculations within VM, helper function is in a new file *sys/vmmeter.h.*
Modification: *vm_page.c*

The change tasks show the ways in which 2 modules, *swap_pager.c* and *vm_page.c* are modified over time. In each of the 5 cases, the change impacts both the crosscutting concern and the interacting concern, as the two are not separate in this implementation. We can, however, identify the primary focus of the change, and distinguish between a change that involves interacting versus crosscutting concerns.

PDA_CT1 involves the removal of activation points due to better integration of Swap with VM. This change thus involves both the crosscutting and interacting concerns. PDA_CT2 introduces a new call to *vm_page_unqueue,* as opposed to *vm_page_unqueue_nowakeup*, meaning that the caller intends to wake the deamon. We identify this change as being primarily associated with an interacting concern. PDA_CT3 is a refactoring that reduces the number of exit points from *vm_page_alloc.* Again, this is a change primarily associated with an interacting concern. PDA_CT4 introduces a new asynchronous version of *vm_wait*, and a new low level activation of the daemon, similar to *vm_wait*. This is another change to an interacting concern. PDA_CT5 refactors some thresholds for activation in *vm_page.c,* but other thresholds are unchanged. This change primarily impacts the crosscutting concern itself, page daemon activation.

As a result of these 5 change tasks, of the 4 modules page daemon activation crosscuts, 2 are involved in change.

3.1.2 Mapped-File Prefetching Evolution

The most significant change to prefetching was between v2 and v3, when sequential mode prefetching in the native FreeBSD file system, FFS, was modified to be substantially

more aggressive than normal mode. As part of this change, control flow was diverted from the *getpage* operation, associated with mapped files, to the more general file system read operation where additional asynchronous prefetching would be applied, bringing pages into the buffer cache, and requiring the last position of access within the file be tracked within the read operation. This implementation of sequential mode prefetching was short lived however, as it was subsequently removed between v3 and v4, as outlined in Table 11. Further changes include the addition of more *getpages* functions from new file systems in v4.

Table 11: Prefetching functionality in FreeBSD v2, v3 and v4.

File	Occurrences			Host Functions		
	v2	v3	v4			
vm/vm_fault.c	2	2	2	vm_fault vm_fault_additional_pages		
ufs/ufs/ufs_readwrite.c	3 0	4 2	3 0	ffs_getpages ffs_read		
vm/vnode_pager.c	3	3	3	vnode_pager_getpages		
nfs/nfs_bio.c	2	2	2	nfs_getpages		
nwfs/nwfs_io.c	0	0	1	nwfs_getpages		
fs/smbfs/smbfs_io.c	0	0	1	smbfs_getpages		
Total: 6	*10*	*13*	*12*	*v2: 5*	*v3: 6*	*v4: 7*

The three most significant change tasks associated with maintenance and evolution of prefetching from v2 to v3 and v3 to v4 are:

PREF_CT1: Addition of optimized sequential mode prefetching in FFS.
Evolution: v2 to v3
Result: Change of interaction with file system read path.

PREF_CT2: Removal of optimized sequential mode prefetching in FFS.
Evolution: v3 to v4
Result: Undo PREF_CT1.

PREF_CT3: Addition of two new file systems that employ same prefetching as NFS.
Evolution: v3 to v4
Result: More low level prefetching in getpage operations.

PREF_CT1 and PREF_CT2 result in the addition and removal of an optimization for FFS only, which is focussed solely on prefetching and not interacting concerns. PREF_CT3, the addition of new file system *getpage* operations is focussed primarily on interacting concerns, but results in more places where prefetching applies. Of the 6 modules that prefetching crosscuts, 3 are involved in change.

3.1.3 Disk Quota Evolution

The implementation of the quota functionality grew incrementally, as a result of new features being added to the file systems: from v2 to v3, it spread to 16 new places, and from v3 to v4 to one new place in FFS, as outlined in Table 12.

Table 12: Quota in FreeBSD v2, v3, and v4.

File	Occurrences			Host Functions		
	v2	v3	v4			
ufs/ufs/ufs_vfsops.c	0	1	1	ufs_init		
ufs/ufs/ufs_vnops.c	7	15	15	ufs_access ufs_chown ufs_mkdir ufs_mkinode		
ufs/ufs/ufs _inode.c	3	3	3	ufs_inactive ufs_reclaim		
ufs/ffs/ffs _vfsops.c	3	3	3	ffs_flushfiles ffs_sync ffs_vget		
ufs/ffs/ffs _inode.c	2	3	3	ffs_truncate		
ufs/ffs/ffs _alloc.c	5	5	5	ffs_alloc ffs_realloccg		
ufs/ffs/ffs_balloc.c	0	0	1	ffs_balloc		
gnu/ext2fs/ext2_vfsops.c	4	4	4	ext2_flushfiles ext2_sync ext2_vget		
gnu/ext2fs/ext2_vnops.c	0	8	8	ext2_mkdir ext2_mkinode		
gnu/ext2fs/ext2_inode.c	2	2	2	ext2_truncate		
gnu/ext2fs/ext2_alloc.c	3	3	3	ext2_alloc		
Total	*29*	*38*	*39*	*v2: 17*	*v3:20*	*v4:21*

A list of the most significant change tasks involved with quota evolution between v2 to v3 and v3 to v4 is as follows:

QUO_CT1: Addition of new file server feature for assigning file ownership.
Evolution: v2 to v3
Result: 16 new compiler directives in *ufs_mkdir/mkinode*, *ext2_mkdir/mkinode*

QUO_CT2: Revamping of existing feature in UFS from *inode* to *vfsop*.
Evolution: v2 to v3
Result: Addition of new compiler directive for an existing feature in UFS.

QUO_CT3: Addition of new feature in FFS.
Evolution: v3 to v4
Result: Addition of new compiler directive for new feature in FFS.

Changes to disk quota between v2 and v3 resulted from the introduction of a new feature for file servers, which automatically assigned the ownership of a new file to be that of the enclosing directory.

Since ownership relates to disk consumption, this new feature was interleaved with quota code, introducing 14 new compiler directives for quota over 4 existing UFS and EXT2 operations. Also between v2 and v3, a UFS operation, *ufs_init*, was moved from *ufs_inode.c* to *ufs_vfsops.c* and a compiler directive was introduced to the *dqinit* operation as shown below.

```
int
ufs_init()                              /* in file ufs_inode.c v2 */
{
    static int first = 1;

    if (!first)
            return (0);
    first = 0;

#ifdef DIAGNOSTIC
    if ((sizeof(struct inode) - 1) & sizeof(struct inode))
            printf("ufs_init: bad size %d\n", sizeof(struct inode));
#endif
    ufs_ihashinit();
    dqinit();
    return (0);
}
```

```
int
ufs_init(vfsp)                              /* in file ufs_vfsops.c v3 */
   struct vfsconf *vfsp;
{
   static int done;

   if (done)
           return (0);
   done = 1;
   ufs_ihashinit();
#ifdef QUOTA
   dqinit();
#endif
   return (0);
}
```

Between v3 and v4, one new FFS operation was introduced, also requiring quota tracking. This function was *balloc*, used to define the structure of file system storage by allocating the physical blocks on a device given the inode and the logical block number in a file. Corresponding functionality does not exist in the EXT2FS.

As a result of these 3 change tasks, of the 11 modules disk quota crosscuts, 5 are involved in change.

3.2 Impact of Change Tasks on the Aspect-Oriented Kernels

In the original implementation, since the crosscutting and interacting concerns are not separate, changes to the crosscutting concern necessarily involved changes to the module(s) of the associated interacting concerns. In the aspect-oriented implementation, changes are better localized. That is, changes to the crosscutting concerns can be localized within one module, the aspect, and changes to the interacting concerns that result in more or fewer places where the aspect applies can be localized within the modules for those concerns.

The impact of the change tasks previously identified are summarized in Table 13, with two more columns added to show where the changes take place – the interacting concerns (IC) or the aspect (A). The Table shows that, in 7 of the 10 tasks that require any change, either the interacting concerns or the aspect requires change, but not both. This result reflects the improved independence between the crosscutting concerns and the code they crosscut in the aspect-oriented implementation. Details of the impact of

each change on the aspect-oriented implementation are presented in the subsections that follow.

Table 13: Changes (√) versus no change (captured), in AOP implementation.

change task	from	description	focus of change	change IC	change A
PDA_CT1	v2→v3	integration of Swap and VM	both IC and activation aspect	√	√
PDA_CT2	v2→v3	new function *vm_page_unmanage*	IC	√	*(captured)*
PREF_CT1	v2→v3	new sequential mode prefetching	prefetching aspect		√
QUO_CT1	v2→v3	new feature in file systems	both IC and quota aspect	√	√
QUO_CT2	v2→v3	new configurability for *ufs_init*	quota aspect	--	--
PDA_CT3	v3→v4	revamp of *vm_alloc*	IC	√	*(captured)*
PDA_CT4	v3→v4	new function *vm_await*	both IC and activation aspect	√	√
PDA_CT5	v3→v4	refactoring in *vm_page.c*	activation aspect		√
PREF_CT2	v3→v4	removal of sequential mode prefetching	prefetching aspect		√
PREF_CT3	v3→v4	new *getpages* in new file systems	IC	√	*(captured)*
QUO_CT3	v3→v4	new feature in FFS	IC	√	*(captured)*

3.2.1 Evolution of Page Daemon Activation Aspect

PDA_CT1: Better integration of Swap with VM.
Evolution: v2 to v3
Original: Removal of 7 occurrences of activation from Swap.
A+IC: Deletion of 4 pointcuts/advice

Better integration of Swap with VM required changing many concerns – daemon activation being only one of them. Of the four key functions involved with activation, *swap_pager_getpages/putpages* are revamped, *swap_pager_iodone/sync* are refactored into

other functions, and in all cases activation of the daemon is removed and left to be triggered by VM operations. The changes in the IC thus require removal of corresponding pointcuts and advice in the aspect, so both IC and the aspect are impacted by this change task.

PDA_CT2:	A new VM function, *vm_page_unmanage*.
Evolution:	v2 to v3
Original:	Introduction of a new call to the unqueue operation that wakes the daemon, *vm_page_unqueue*.
IC:	No change to the aspect, the existing pointcut applies to one new place, capturing this new functionality in IC.

The pointcut used to trigger activation when unqueuing available pages specifically captures special cases when the daemon is not activated:

```
pointcut unqueue(vm_page_t m):
    execution(void vm_page_unqueue(m))
    && !cflow(execution(void vm_page_alloc(vm_page_t))
            || execution(void vm_page_free_toq(vm_page_t))
            || execution(void vm_page_cache(vm_page_t, int)));
```

As previously discussed in terms of implementation alternatives in Section 2.3.1.1, this means that new calls to *vm_page_unqueue* are automatically absorbed and trigger activation. This change task thus requires modification of the IC only.

PDA_CT3:	Maintenance of VM function, *vm_page_alloc*.
Evolution:	v3 to v4
Original:	Removal of 2 occurrences of activation from within VM.
IC:	No change to the aspect, the existing pointcut applies to two fewer places.

During this revamping, several exit points were consolidated in *vm_page_alloc*. The number of return statements that return *NULL*, dropped from 4 to 2. The fundamental relationship in the pointcut however, still holds as originally captured:

```
pointcut allocate_pages(vm_object_t object, vm_pindex_t pindex, int page_req):
    execution(vm_page_t vm_page_alloc(object, pindex, page_req));

around(vm_object_t object, vm_pindex_t pindex, int page_req):
      allocate_pages(object, pindex, page_req)
 {
    vm_page_t allocd_page = proceed(object, pindex, page_req);
    if (allocd_page == NULL)
      pagedaemon_wakeup();
    else
      if (cnt.v_free_count + cnt.v_cache_count < cnt.v_free_reserved + cnt.v_cache_min)
        pagedaemon_wakeup();
    return allocd_page;
  }
```

This change task thus requires modification of the IC only.

PDA_CT4:	Addition of a new VM function, *vm_await*.
Evolution:	v3 to v4
Original:	Introduction of new low level activation of page daemon wakeup.
A+IC:	Modification of the pointcut associated with sleeping in wait.

The introduction of an asynchronous sleep operation resulted in many corresponding changes throughout VM, offering both the original *tsleep* version of functions and their new *asleep* counterparts. The core difference between *tsleep* and *asleep* involves blocking immediately versus blocking upon a subsequent call to *await*.

The impact this has on the page daemon activation aspect deals with the introduction of the new asynchronous version of the *vm_wait*, operation, *vm_await.* We reflect this change by modifying the associated pointcut, as follows:

```
        pointcut sleep_in_wait(void* count, int priority, char* fname, int flag):
           (execution(int tsleep(count, priority, fname, flag))
            && cflow(void vm_wait()))
/* new */  || (execution(int asleep(count, priority, fname, flag))
                && cflow(void vm_await()));
```

Another important way that the aspect-oriented implementation differs from the original is in terms of the low level activation of the page daemon introduced in the new function *vm_await*, associated with asynchronous VM operations. As in its synchronous counter part, *vm_wait*, this low level activation essentially inlines the body of the *pagedaemon_wakeup* function, used by all other instances of daemon activation in the original code. Assuming that the wakeup function was designed to be an interface for daemon activation, instances where it is bypassed circumvent the ability to standardize

activation through this interface. In the aspect, we restructured activation to avoid this end-run, enforcing coordination of all activation through *pagedaemon_wakeup*. As a result, we modified the pointcut, as above, but kept the advice the same:

```
around(void* count, int priority, char* fname, int flag):
  sleep_in_wait(count, priority, fname, flag)
  {
    if (where == PVM)
       pagedaemon_wakeup();
    return proceed(count, priority, fname, flag);
  }
```

This change task thus includes corresponding modification to both the IC and the aspect.

PDA_CT5:	Addition of new VM helper function, *vm_paging_needed*.
Evolution:	v3 to v4
Original:	Refactoring of some activation threshold calculations within VM.
A:	New helper functions for threshold calculations within aspect.

This change task provides a simple example of how inconsistencies arise in the original implementation. Taking 3 sample thresholds from 3 files in the original FreeBSD v3 implementation, the thresholds display a similar pattern, with subtle differences:

```
/* from file vm/vm_page.c */
if ((cnt.v_cache_count + cnt.v_free_count) <
     (cnt.v_free_reserved + cnt.v_cache_min))
            pagedaemon_wakeup();


/* from file vm/vm_fault.c */
if ((rahead + rbehind) >
     ((cnt.v_free_count + cnt.v_cache_count) - cnt.v_free_reserved)) {
            pagedaemon_wakeup();

/* from file kern/vfs_bio.c */
if ((curproc != pageproc) &&
     ((m->queue - m->pc) == PQ_CACHE) &&
     ((cnt.v_free_count + cnt.v_cache_count) <
     (cnt.v_free_min + cnt.v_cache_min))) {
            pagedaemon_wakeup();
}
```

This change task refactors thresholds within *vm_page.c* as shown below, but the other thresholds remain unchanged:

```
/* from file vm/vm_page.c */
if (vm_paging_needed())
     pagedaemon_wakeup();
```

When we replayed this change task within the aspect, localization of the functionality allowed us to more easily ensure refactoring of the threshold was internally consistent. We added some of our own refactoring to draw attention to the common use of *cnt.v_cache_count + cnt.v_free_count*, and the context-dependent use of *cnt.v_free_min* versus *cnt.v_free_reserved*. We should note here that though there were no comments pertaining to the selection of *cnt.v_free_min* versus *cnt.v_free_reserved* in the original code, the aspect affords the opportunity for the original implementer to document the rationale behind the diversity. In the end, the refactoring in this change task resulted in modification of the aspect only.

3.2.1.1 *Page Daemon Activation Change Task Summary*

Table 14 overviews the change tasks associated with the page daemon activation aspect. In summary, 3 of the 5 change tasks, PDA_CT2, PDA_CT3 and PDA_CT5, are made more independent, requiring changes to one of IC or the aspect but not both. In 2 of these 3 change tasks, PDA_CT2 and PDA_CT3, existing pointcuts automatically apply to changes in the ICs, while PDA_CT5 introduces refactoring internal to the aspect. The remaining 2 change tasks, PDA_CT1 and PDA_CT4, are of a more comprehensive nature and require modifications to both the code being crosscut, and the crosscutting concern. PDA_CT1 requires removal of pointcuts and advice when subsystems are integrated, and PDA_CT5 requires reconfiguration of an existing pointcut to include new functionality. Though PDA_CT2 and PDA_CT5 both introduce a new function, PDA_CT5 introduces an element of functionality not originally captured within the pointcuts of the aspect.

Table 14: Summary of page daemon activation change tasks.

change	locality of change
new function added to IC (PDA_CT2), refactoring of existing function in IC (PDA_CT3)	interacting concern only
refactoring of aspect functionality (PDA_CT5)	aspect only
integration of subsystems (PDA_CT2), new function added to IC (PDA_CT4)	both interacting concerns and aspect

3.2.2 Evolution of Prefetching Aspect

Figure 15 illustrates the structure of the optimization of sequential mode in FFS. Though the optimized version of sequential mode prefetching involved only a relatively small number of changes to the system, it introduced an important new interaction between VM, the file system, and the buffer cache that did not previously exist. At the level of the interfaces of the primary modularity involved, this interaction was impossible to detect, and furthermore was non-standard across file systems. This optimization was introduced in v3 and removed in v4. The other significant change to prefetching involves the introduction of new file systems, which handle low level deallocation associated with prefetching in the same way as existing systems.

PREF_CT1:	Addition of optimized sequential mode prefetching in FFS.
Evolution:	v2 to v3
Original:	Change of interaction with file system read path.
A:	Aspect for sequential mode, shown in Figure 16, with advice for each oval from Figure 15 labelled 1-3.

In the original implementation, this optimization involved adding a few dozen lines of code to several functions within VM and FFS to join them along a new execution path to disk. Our original decision to make this optimization a separate aspect is in keeping with the internal structural unit previously identified for prefetching – an execution path. Since sequential mode changes the execution path relative to normal mode, the new path seemed to naturally dictate a new aspect. Before examining the new aspect, we provide an overview of the design of sequential mode prefetching.

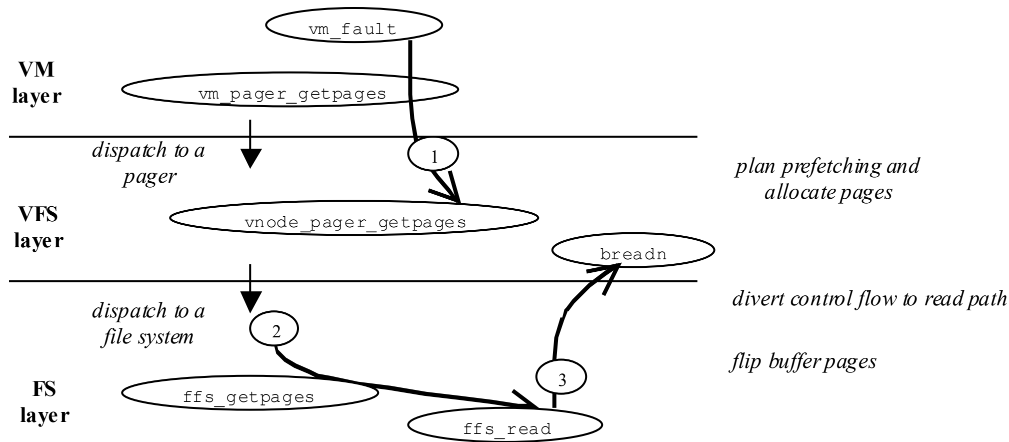

Figure 15: Sequential mode prefetching for FFS in FreeBSD v3.

For VM objects with sequential behaviour, prediction simply says that future accesses will directly follow the current access. The planning phase allows prediction to dominate cost analysis –predicted pages are prefetched even if they are not contiguous on disk, as prefetching is done asynchronously.

This aggressive sequential prefetching is handled by redirecting control flow through the file system read operation, *ffs_read,* instead of through the normally used operation for mapped file access, *ffs_getpages*. This path triggers yet another prefetching mechanism specific to the file system read service, which asynchronously prefetches according to a sequential access pattern. The pages read from disk must eventually end up in the pages allocated higher up the execution path to the VM object. A page-aligned exchange that flips pages between the file buffer cache and the VM allocated pages is not part of a typical file system read operation, but avoids an expensive copy operation and thus is required on this particular execution path.

```
aspect sequential_mapped_file_prefetching {

   pointcut fault_path(vm_map_t map):
       cflow(execution(int vm_fault(map, vm_offset_t, vm_prot_t, int)));

   pointcut ffs_read_path( struct vnode* vp, struct uio* io_info, int size,
                          struct buff** bpp ):
       cflow( execution( 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 ):
(1)    execution( int vnode_pager_getpages( object, pagelist, length, faulted_page ))
       && fault_path( 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 ):
       execution( int ffs_getpages( object, pagelist, length, faulted_page ))
   {
     if ( object->behaviour == SEQUENTIAL ) {
       struct vnode* vp = object->handle;
(2)    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 ):
       execution( int breadn(struct vnode *, daddr_t, int,
                             daddr_t*, int*,int, struct ucred*, struct buf**) )
       && fault_path(vm_map_t, vm_offset_t, vm_prot_t, int)
       && ffs_read_path( struct vnode*, io_info, size, bpp )
(3)
   {
     flip_buffer_pages_to_allocated_vm_pages( (char *)bpp->b_data, size, io_info );
   }
 }
```

Figure 16. AspectC code for sequential prefetching in A3.

Figure 16 lists the aspect for sequential mode prefetching in A3. Similar to normal mode prefetching, the two pointcuts use *cflow* to make values from higher-levels of the page fault handling path available to prefetching code in lower-levels. Note that this before advice (5) operates independently of the before advice in the aspect for normal mode prefetching, even though they both advise the same function. The around advice (6) diverts the execution path to *ffs_read* when access is sequential, or directs it to *proceed* with *ffs_getpages* otherwise. Finally, the after advice (7), which does the page aligned transfer, executes under the control flow of the pointcuts *ffs_read_path* and *fault_path*. That is, it executes only when control flow has been diverted along this special path.

Given that both the configuration of this new execution path and the new functionality that accompanies it are localized within the aspect, this change task requires changes to the aspect only.

PREF_CT2:	Removal of optimized sequential mode prefetching in FFS.
Evolution:	v3 to v4
Original:	Undo PREF_CT1.
A:	Aspect for sequential mode is removed

In the original implementation, removing this optimization involved removing the lines of code that joined the new path from VM through FFS accordingly. In the aspect-oriented implementation, changes are to the aspect only.

PREF_CT3:	Addition of two new file systems that employ similar prefetching.
Evolution:	v3 to v4
Original:	More low level prefetching in *getpage* operations.
IC:	No change to the aspect, the existing pointcut applies to two more places.

This change task requires that low level deallocation coordinate with high level VM functionality within low level functions in new file systems for Netware and Samba. As this deallocation is done the same way as previously described for NFS, the pointcut and advice apply to these new systems as well:

```
pointcut pager_error(int error, int uio_resid, struct vop_getpages_args* ap):
    execution(int check_read(error, uio_resid, ap->a_count));


around(int error, int uio_resid, struct vop_getpages_args* ap):
   pager_error(error, uio_resid, ap)
{
     int success, npages, i;
     success = proceed(error, uio_resid, ap->a_count);

     if (!success) {
       npages = btoc(ap->a_count);
       for (i = 0; i < npages; i++) {
         if (ap->a_reqpage != i)
              vnode_pager_freepage(ap->a_m[i]);
       }
     }
 }
```

The assumption made when the aspect was built was that similar file systems would intentionally offer this same interface for composition with the prefetching aspect, as was

discussed in terms of implementation alternatives in Section 2.3.2.1. As a result, the aspect applies to the new functionality without requiring modification. Only the IC is modified.

3.2.2.1 Prefetching Change Task Summary

Table 15 overviews the change tasks associated with the mapped-file prefetching aspect. In summary, all 3 change tasks are made more independent, requiring changes to one of aspect or IC but not both. One change, PREF_CT3, is captured by existing pointcuts and advice and in the remaining change tasks, PREF_CT1 and PREF_CT2, a new aspect is introduced and removed.

Table 15: Summary change tasks and impact on the prefetching aspect.

change	locality of change
new functions added in IC (PREF_CT3)	interacting concern only
addition/removal of aspect (PREF_CT1, PREF_CT2)	aspect only

3.2.3 Evolution of Quota Aspect

Similar to the page daemon activation and prefetching aspects, one of the change tasks associated with quota is comprehensive and requires changes to both IC and aspect (QUO_CT1), and another involves new functionality already captured in an existing pointcut (QUO_CT3). Unlike the other aspects however, we argue that one of these changes would not have occurred in the aspect-oriented implementation (QUO_CT2).

QUO_CT1: Addition of new file server feature for assigning file ownership.
Evolution: v2 to v3
Original: New compiler directives in *ufs_mkdir/mkinode*, *ext2_mkdir/mkinode*
A+IC: Introduction of new pointcuts/advice to quota aspect.

In the original implementation, this change task resulted in redundant code being added to both UFS and EXT2. Judging from the identical comments in the code, it appears to

have been cut and pasted between the two file systems, with minor modifications added to reflect taste. In the quota aspect, we introduced this code in one place, within the aspect, and specified that it should be applied to both operations, explicitly ensuring consistency across these two contexts. An additional feature of the aspect with respect to coordination is that this explicit sharing also makes structural properties within the primary modularity explicit. Until this new feature was added, sharing of quota is declaratively coordinated between FFS and EXT2. This addition thus reveals a new structural property that exists between UFS, EXT2, this new feature, and quota, and requires changes to both IC and the aspect.

QUO_CT2: Revamping of existing feature in UFS from *inode* to *vfsop*.
Evolution: v2 to v3
Original: Addition of new compiler directive for an existing feature in UFS.

We argue here that, in the original implementation, leaving the compiler directive out of the implementation of *ufs_init* when *dqinit* is called was most likely an oversight. As previously mentioned, in v2 the implementation of *ufs_init* was:

```
int
ufs_init()                              /* in file ufs_inode.c v2 */
{
    static int first = 1;

    if (!first)
            return (0);
    first = 0;

#ifdef DIAGNOSTIC
    if ((sizeof(struct inode) - 1) & sizeof(struct inode))
            printf("ufs_init: bad size %d\n", sizeof(struct inode));
#endif
    ufs_ihashinit();
    dqinit();
    return (0);
}
```

Whereas in v3 it became:

```
   ufs_ihashinit();
#ifdef QUOTA
   dqinit();
#endif
   return (0);
```

As initializing the quota module is part of the quota concern, it would be much more difficult to justify including this in the IC when implementing the quota aspect in v2. We thus consider this change task to be a special case that would not have occurred in the aspect-oriented implementation.

QUO_CT3:	Addition of new feature in FFS.
Evolution:	v3 to v4
Original:	Addition of compiler directive and quota functionality for new feature in FFS.

The new feature in FFS requires user quotas to be restored in the event that a new operation fails. Existing pointcut and advice to restore quotas applies, and consequently this change task involves the IC part of the kernel only.

3.2.3.1 Quota Change Task Summary

Table 16 summarizes quota change task activity. The three change tasks associated with quota cover a range where one involves a new function in the IC where the quota aspect would already apply (QUO_CT3), one involves changes to both IC and A (QUO_CT1), and one arguably would not have happened in the aspect-oriented implementation (not shown, QUO_CT2).

Table 16: Change tasks associated with the quota aspect.

change	locality of change
new function in IC (QUO_CT3)	interacting concern only
new feature in IC (QUO_CT1)	both interacting concerns and aspect

3.2.4 Improved Modularity Persists Across Versions

As we studied the impact of change tasks on the different kernels, we were also able to assess how the improved modularity in A2 held up over these changes into A3 and A4.

That is, we were able to confirm that the modularity of the three concerns and their corresponding interacting concerns was still improved in A3 and A4 over the original implementation in O3 and O4.

Table 17 outlines the concerns and the change tasks that exemplify a particular kind of problem encountered as a result of an impact of a change task in the original implementation. The following subsections revisit each change task, focusing on the material impact aspects have on qualities we associated with good modularity for crosscutting concerns in Section 1.2.4 – consistency, coordination, and configurability – on a case-by-case basis. Consistency and coordination refer to the internal representation of the functionality and interaction of a crosscutting concern, whereas configurability refers to the effort required to make a change to either of its functionality or interaction, system-wide.

Table 17: Summary of modularity problems.

concern	consistency	coordination	configurability
page daemon wakeup	PDA_CT5: factoring of threshold calculations evolve differently across subsystems	PDA_CT3: isolated introduction of low level daemon activation	Cumulatively, change tasks are non-modular, involving swap and VM.
mapped file prefetching	PREF_CT1: sequential mode prefetching evolves differently across file systems	PREF_CT3: isolated support for high level functionality within low level functionality	Cumulatively, change tasks are non-modular, involving VM and multiple file systems.
disk quota	QUO_CT1: identical evolution implemented differently across file systems	QUO_CT1: isolated enforcement of disk based restrictions across file systems	Cumulatively, change tasks are non-modular, involving multiple file systems.

3.2.4.1 *Consistency*

Change tasks that highlight the ways in which consistency problems arise in the scattered and tangled implementation include factoring and optimization of a concern in one context but not another, and drift between redundant code when a concern appears in multiple places. Aspects better support internal consistency for crosscutting concerns by providing localization of functionality. Within each aspect, factoring and optimizations are consistent, or at least more explicitly special-cased, and a single advice can be specified to run in multiple contexts.

Refactoring of thresholds in page daemon activation evolution provides a simple example of how inconsistencies arise in the scattered and tangled implementation. This task was inconsistent across files in the original implementation, but easier to make consistent when the functionality was localized in the aspect.

Optimized sequential access mode prefetching in FFS[8] compromises consistency from the perspective of the generalized protocol that exists between VM and file systems with respect to mapped files. It substantially changes the protocol for one file system. Additionally, in the scattered and tangled implementation, it requires that new functionality be added to a part of the file system typically not involved with mapped files. The change can only be detected upon close inspection of the code involved. As an aspect, this special case is made declarative, and stands in contrast to the normal mode prefetching aspect. Though still representing a departure from the norm, the inconsistency is made explicit and overt.

The addition of a new feature in file servers during disk quota evolution resulted in redundant code being added to both UFS and EXT2. In the quota aspect, we introduced

---

[8] Personal communication with the developer who introduced the optimization revealed that, from his perspective, the removal of this optimization was a sign of serious system degradation.

this code in one place, within the aspect, and specified that it should be applied to both file system operations, explicitly ensuring consistency across these parallel operations.

3.2.4.2 Coordination

Change tasks that highlight the ways in which coordination problems arise in the scattered and tangled implementation include the ability to bypass standard interfaces, the requirement to accommodate corresponding high level heuristics in low level functionality, and the necessity to decentralize structural properties of shared protocols. Aspects better structure internal coordination for crosscutting concerns by providing localization of interaction with primary modularity. Within each aspect, functionality is structured to enforce control flow through standard interfaces, to match up corresponding high and low level specialization along execution paths, and to centralize structural properties from a concern-centric perspective.

Low level page daemon activation essentially inlines the body of the *pagedaemon_wakeup* function, circumventing the ability to coordinate activation through this interface. In the aspect, we structured activation to avoid this end-run, enforcing coordination of all activation through the standard interface. Localization of interaction thus makes it easier to ensure a narrower interface.

Deallocation in prefetching introduces support for high level VM functionality within low level functions in new file systems. In the aspect, this activity is explicitly coordinated along execution paths, combining high level planning and allocating with this lower level functionality.

As previously established, consistency within the disk quota aspect is ensured due to the fact the code appears once and can declaratively run in multiple contexts. Coordination within the aspect involves sharing of structural information that makes additional properties within the primary modularity explicit. Until the new feature for file servers was added in QUO_CT1, sharing was coordinated only between FFS and EXT2, so the

addition thus reveals a new structural property between UFS, EXT2, the new feature for file servers, and quota.

3.2.4.3 Configurability

In the original implementation, the cumulative changes involved in the evolution of each concern are non-modular. They require making changes to more than one function, in more than one file, and often in more than one subdirectory. Aspects better enable configurability by providing localization of both functionality and interaction with primary modularity. As a result, evolving the aspects involves changes to the aspects, and not the primary modularity they crosscut. The cumulative changes involved in the evolution of each aspect are thus modular.

Table 18 contrasts configurability in terms of evolutionary changes in the original versus the aspect-oriented implementation. In the original implementation, configurability during evolution involved adding/removing individual lines of code (LOC) to/from multiple functions. In the aspect-oriented implementation, evolutionary changes were on the level of adding/removing/modifying modularized and structured elements of the implementation that were not previously configurable in the original implementation, such as contexts, paths, and shared functionality.

Table 18: Configurability of crosscutting concerns over the 11 changes tasks.

concern	original	AOP
page daemon activation	+/- LOC 2 of 4 files	+/- pointcuts/advice 1 of 1 file
prefetching	+/- LOC 3 of 6 files	- aspect change to Makefile
disk quota	+ LOC 5 of 11 files	+ pointcuts/advice 1 of 1 file

3.2.4.4 Discussion

The previous sections have highlighted the ways in which modularity of concerns is maintained across releases in the aspect-oriented implementation. Consistency is improved primarily as a result of localization of functionality – naming, factoring, special cases and even coding style and comments can be concern-centric, and are no longer dictated in terms of the parts of the system they crosscut. Coordination is improved due to localization of interaction with primary modularity – use of standardized interfaces, staged protocols, and structural properties can be explicitly enforced between otherwise isolated contexts. Configurability is improved due to localization of both functionality and interaction with primary modularity – changing what parts of the primary modularity the aspect sees, what values it shares with the code it crosscuts, and what functionality it provides can all be done within one module.

A possible criticism of this comparison is that it is an inadequate basis upon which to strongly conclude that these improvements are inherent to the aspects. Since FreeBSD is a highly evolved code base, we believe we have been comparing to the best possible competing implementation. But for the sake of argument, we now consider properties of the two approaches in general terms, to more concretely identify inherent differences.

Traditional modular encapsulation of any of these concerns could better promote internal consistency with respect to factoring, naming and redundant code. Refactored as traditional modules, the interfaces involved could roughly align one-to-one with contexts

in the primary modularity, requiring the caller to decide which function within the module to call. This is similar to the variation in unqueue operations in *vm_page.c*, where the caller selects either *vm_page_unqueue* or *vm_page_unqueue_nowakeup* depending on whether or not the page daemon might need to be activated. Alternatively, the module could have a narrower interface but require that context be passed in, and dispatch to the correct functionality based on that context. But this modularization introduces new management overheads associated with interaction, with diminishing returns in terms of independent development.

SPIN [Bershad, Savage*, et al.* 1996] demonstrated that a wide interface can be exceedingly difficult to manage due to the extensive semantic properties involved [Savage 1999][9]. One of the difficulties associated with the management of a wide interface is unstructured coordination given a traditional implementation. In an aspect-oriented implementation, the semantic properties can be made explicit and structured in a modular fashion.

Without language mechanisms to modularize and structure both functionality and interaction with primary modularity, we could not have structured and subsequently evolved in a modular way the context/threshold pairs in page daemon activation, the path-specific customization in prefetching, or the parallel protocol in disk quota. These internal structures inherently reduce the otherwise implicit dependencies with interacting concerns, improving independence of crosscutting concerns to a degree unattainable in any traditional approach.

3.3 Validation

The purpose of this case study was to provide validation for the following claim:

---

[9] The fine-grained extensibility supported by SPIN resulted in an interface an order of magnitude wider than the typical Unix interface.

***Claim 2**: Improvements in modularity persist across 3 versions.*

The analysis was twofold. First, we identified the locality, in terms of number of modules involved in changes to the crosscutting concern, of the differences of impact the 11 change tasks had on the aspect-oriented versus original implementations over successive releases: v2 to v3, and v3 to v4. Second, we analyzed modularity in terms of support for consistency, coordination and configurability over the 3 versions of each implementation.

In total, of the 5 modules page daemon activation crosscuts, changes to page daemon activation occur in 2 modules in the original implementation, versus 1 module – the aspect – in the aspect-oriented implementation. Similarly, of the 6 modules mapped file prefetching crosscuts, changes to prefetching occur in 3 modules in the original implementation, versus 1 module in the aspect-oriented implementation. Finally, of the 11 modules disk quota crosscuts, changes to quota occur in 5 modules in the original implementation, versus 1 module in the aspect-oriented implementation.

A more detailed analysis shows that, in original kernels, O2, O3 and O4, all change tasks require changes to functions that implement both crosscutting and interacting concerns. In aspect-oriented kernels, A2, A3, and A4, 3 of the 11 change tasks required changes to both crosscutting and interacting concerns, 7 change tasks required changes to just one of the crosscutting or interacting concerns, and 1 change task would arguably not have been required.

As with the previous case study, we show that the localization of the differences between non-interacting concerns remains constant, due to the fact that we have not introduced any unintentional interaction.

A possible criticism of these results is that, in the aspect-oriented implementation, we had perfect foresight of the change tasks. We addressed this issue by discussing precisely why we felt our implementation had a high likelihood of being reasonable with references to possible alternative implementations in Section 2.3. Though the end result of choosing any of these alternatives would have changed the ability of the aspect to

apply in more or fewer places as the interacting concerns change, they would have no impact on the number of modules that needed to change in order to make a change to the crosscutting concern itself, which would still be localized to the aspect.

4 Inferences of Evolution and Scale

This chapter provides validation for the remaining claims in the thesis, inferring results from the case studies, and generalizing the specific case of the three concerns over 3 releases of FreeBSD to include more concerns, more releases and more systems. Before generalizing however, we consider adoption of AOP in general and as it relates to these case studies, and costs associated with the AspectC runtime. In order to provide additional support for claims of generalizability, we consider specific examples of more concerns and more systems.

4.1 Evolution

Based on the results of the previous case studies on single and multi-version modularity, we infer the following.

> ***CLAIM 3**: AOP can be used to improve evolvability of 3 crosscutting concerns and their interacting concerns without harming non-interacting concerns.*

If non-modular concerns are hard to evolve, and modularity is improved in A2 versus O2 (Claim 1), then we can infer that evolution would have been easier in the aspect-oriented implementation due to better modularity. We support this claim by inference, as we did

not study the process of evolution of the entire system, but instead the impact of a subset of changes. Further studies on the process of evolution are required to support this claim directly.

One source of additional support for this claim comes from identifying the common source of modularity improvements for the three concerns and their corresponding interacting concerns. Modularization of activation points for the page daemon (Section 2.3.1), execution paths for prefetching (Section 2.3.2), and parallel functionality for disk quota (Section 2.3.3) require locality of both functionality and external interaction to be able to structure them effectively. Simultaneous improvement in localization of interacting concerns is accomplished by the removal of the crosscutting concern in each case.

Another source of additional support for this claim comes from the analysis of locality of change (Section 3.2), and the ways in which modularity persists across versions (Section 3.2.4). In the original implementation, the change tasks require changes to functions that implement both crosscutting and interacting concerns. In the aspect-oriented implementation, 4 of the 11 change tasks modify interacting concerns only, though the aspect applies in more or fewer places as a result of the change; 3 change tasks require a change to aspects only; 3 change tasks require changing both crosscutting and interacting concerns; 1 change arguably would not have happened.

One possible criticism of this inference is that we are basing it on a small subset of change tasks. Though the set is small, it is based on the impact of real change between FreeBSD v2, v3 and v4 and includes changes focussed primarily on the interacting concerns, the crosscutting concerns, or both.

In the case where change tasks involve both, another possible criticism is that, relative to the same cases in the tangled implementation, more than one module may need to be modified. We believe that, in such cases, this is appropriate, as these changes involve

either new features or integration of old features that impact aspects, interacting, and non-interacting concerns.

4.2 Adoption of AOP: Tools and Refactoring

Having inferred that the evolution A2 to A3 and A3 to A4 would be easier than O2 to O3 and O3 to O4, we now turn to the issue of adopting AOP. Support for aspect-oriented programming includes several projects targeting the development of language extensions and tools. Though currently there is no tool support for AspectC, this survey includes a brief look at simple tool support for AspectJ to demonstrate the role of tools in AOP.

We also consider the process of retroactively refactoring crosscutting concerns from within an existing system, as we did in our case studies. We further qualify refactoring in terms of the type of separation involved: simple separation versus refactoring to expose parameters or functionality along an execution path. An overview of the refactoring involved to separate each of the three FreeBSD concerns is provided.

4.2.1 Tool Support

Programming environment tools are an important feature of aspect-oriented programming, designed to aid navigation and compositional reasoning between the separated aspect and the primary modularity it crosscuts. In this section, we briefly highlight two simple features of tools currently available for AspectJ, and identify examples where they would have been most useful during the evolution of the FreeBSD aspects. Tools aid the developer in ensuring that non-interacting concerns are not harmed.

A simple extension to Emacs is shown in Figure 17. This screenshot shows what a developer would see when editing code of a *SpaceWar* game. The Figure shows a screenshot with the cursor on a method, *fire*, which is advised by an aspect, *EnsureShipIsAlive*. The editor supports both navigation and compositional reasoning

between the aspect and the primary modularity it crosscuts by providing a means of easily switching between primary modularity and aspect code within the editor. When viewing primary modularity with the tool enabled, a visual tag indicates which methods are advised, and which aspects advise them.

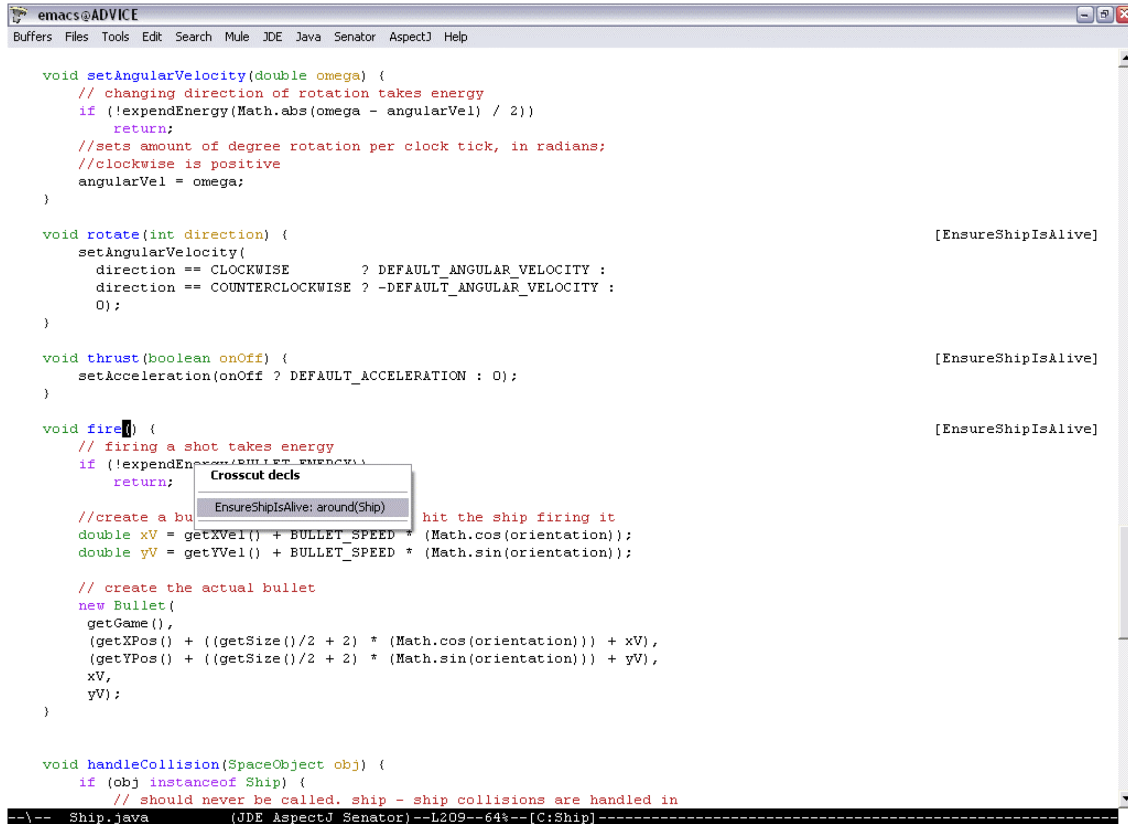

Figure 17: A simple extension to Emacs for AspectJ.

This kind of support would have been useful to map the evolutionary changes from the contexts of the page daemon activation within the primary modularity to the aspect that controls activation. In particular, the swap functions in which the contexts for activation were removed during evolution would be labelled in the editor as advised by the page daemon activation aspect. Figure 18 shows a mock-up of an Emacs window to demonstrate what it would look like to a developer if we had this same tool support for AspectC. The label for the aspect would remind the programmer to check the advice during the revamping of the interacting concern within the primary modularity.

```
…

/*
 * swap_pager_iodone1 is the completion routine for both reads and async writes
 */
void
swap_pager_iodone1(bp)    crosscut decls:                          [page_daemon_activation]
        struct buf *bp;     page_daemon_activation after()
{
        bp->b_flags |= B_DONE;
        bp->b_flags &= ~B_ASYNC;
        wakeup(bp);
}

…
```

Figure 18: Mocked-up example of Emacs tool for AspectC.

More sophisticated graphical environments are available in projects such as Eclipse [Eclipse.org 2003]. Though systems programmers tend not to be early adopters of heavy weight programming environments, the CDT (C/C++ Development Tools) project [CDT 2003] for Eclipse was designed specifically with Linux systems programmers in mind.

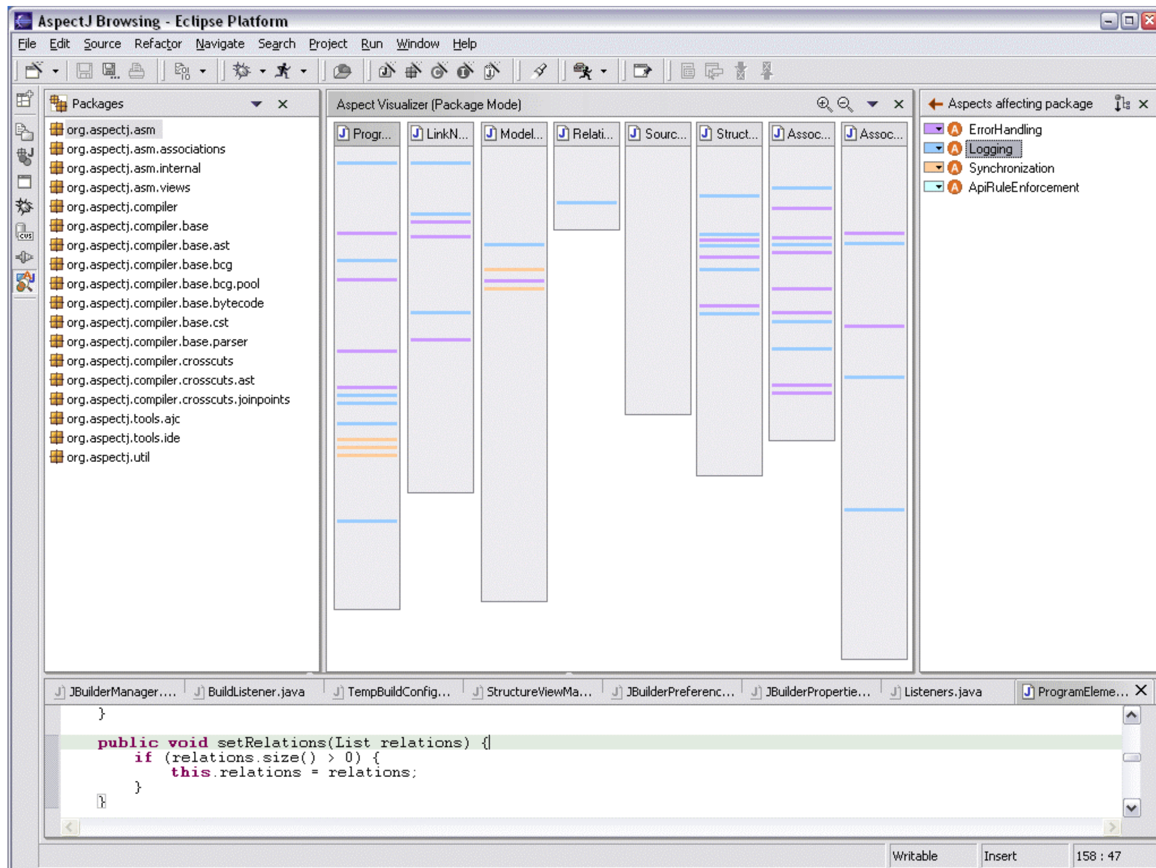

Figure 19: Support for compositional views of crosscutting for AspectJ within Eclipse.

Figure 19 demonstrates how tool support for AspectJ in Eclipse provides higher level and comprehensive views of crosscutting. The screenshot shows four aspects, each corresponding to a different colour (not shown in this black and white snapshot). The vertical rectangles represent files in the primary modularity, and the horizontal coloured lines represent specific lines of code where advice applies.

Views such as this shown in Figure 19 would have been most useful when reasoning about the possible interaction between page daemon activation and prefetching aspects that crosscut the same sections of VM code.

Others tools that aid in visualization of the implementation of concerns include the AspectBrowser [Griswold, Kato, *et al.* 1999], Aspect Mining Tool, AMT [Hannemann and Kiczales 2001] , which offer graphical interfaces for analysis techniques, and the Feature Exploration and Analysis Tool, FEAT [Robillard and Murphy 2002], which supports general tasks associated with locating, describing, and analyzing the code implementing any given concern.

4.2.2 Separating Concerns

The refactoring required to separate the FreeBSD concerns considered in our case studies arguably could have been different depending upon the original functional decomposition provided by the original OS. Though these efforts would not be part of an aspect-oriented implementation that started at the level of design, close inspection of the nature of the refactoring encountered helps identify important issues to consider in non-retroactive cases.

Without exception, the aspects considered here involve two distinct types of refactoring: (1) simple separation, where the code for the aspect is removed from the primary modularity without further fine-grained changes required, and (2) separation with exposure, where new helper functions expose otherwise local values that must be shared between the primary modularity and the aspect. Whereas simple separation seems to

correspond largely to what can be considered as the degree of scattering of a concern, separation with exposure corresponds more to what can be considered as the degree of tangling involved with a given decomposition. We thus examine the relative efforts associated with each type of refactoring in terms of the relative effort required to reduce scattering and tangling respectively.

4.2.2.1 Reducing Scattering

Simple separation reduces scattering without requiring more fine-grained exposure of function-local values involved. In some cases, this requires further decomposition of the primary modularity to include helper functions for advice to attach. An assumption we made about introducing such helper functions was that they were generally available for use outside of our own requirements, and thus could be called from anywhere in the system. We did not, however, restrict the application of advice to these helper functions to be within the *cflow* of any specific functions. That is, if an outside caller were to use the helper function, the advice will be triggered as well.

Though a different functional decomposition of the original system may have coincidentally exposed all functionality for advice to attach without further refactoring, such a granularity of decomposition in legacy systems may be unlikely to expect due to historical influences. The approach to functional decomposition used in early releases of FreeBSD can be characterized as mostly large, coarse-grained functions that rely heavily on macros for sub-function computation for performance reasons. During evolution from FreeBSD v2 to v4 however, there is a noticeable shift to more frequent use of smaller helper functions, with file scope, to replace macros and further improve readability of subsystems that have since been revamped.

4.2.2.2 Reducing Tangling

Relative to the effort required to reduce scattering, the refactoring required to reduce tangling is complicated by the need to decompose functionality to expose otherwise local

values shared between the aspects and the primary modularity involved. Table 19 overviews some examples of exposure introduced during the refactoring of each concern.

Table 19: Examples of exposure of otherwise function-local values for each concern.

concern	shared exposure	primary modularity involved
page daemon activation	values uses to establish thresholds	swap
prefetching	list of pages to retrieve	all *getpage* operations in pagers and file systems
quota	values to be rolled back	vetoed file system operations

For page daemon activation, some of the values used in threshold calculations with swap are declared local to functions or static to files associated with swap, and hence must be exposed to accommodate the aspect. This appears to violate what might be an important assumption about what elements of this subsystem can expect privacy. Local values associated with the pager and file system crosscut by prefetching and quota must be similarly exposed. But had these systems been designed with the intension of aspect composition from the start, the decision as to what values should be strictly local versus what values need to be shared with aspects would presumably be an additional influence during the design of the primary modularity.

4.2.2.3 *Separation of page daemon activation*

Separating page daemon activation from the code it crosscuts in FreeBSD v2 involves stripping the 17 invocations, and the conditions associated with those invocations, from the 9 functions listed in Table 5 (page 30). In the case of one function, *vm_page_unqueue*, stripping this invocation made this function identical to another function, *vm_page_unqueue_nowakeup.* This refactoring thus required further identifying which callers invoked the former version of the function, and explicitly including those callers as part of the context associated with that particular activation.

Four functions call *vm_page_unqueue,* and four other functions call *vm_page_unqueue_nowakeup*. During refactoring, we replaced calls to the no longer available *nowakeup* version of this function with calls to newly refactored *vm_page_unqueue,* moving the decision of whether or not to wake the daemon into the aspect. Though subtle, this change is significant in that the caller of the unqueue operation can no longer decide whether or not the daemon may be activated, making the aspect the locus of control.

Other refactoring involved accommodating advice and non-local access to elements within the primary modularity. Accommodating advice resulted in refactoring parts of larger functions to include smaller helper functions to which advice could be attached. Accommodating non-local access meant changing local file declarations – static declarations and *#define* directives – to global declarations. This was necessary for two static values and one *#define* within swap.

4.2.2.4 Separation of prefetching

Separating prefetching from the code it crosscuts in FreeBSD v2 involves stripping the 11 sections of code from the 5 functions listed in Table 6 (page 34). Refactoring resulted in the creation of several new helper functions to which advice could be attached. The process of separation was very similar to that previously described for page daemon activation.

4.2.2.5 Separation of quota

Separating quota from the code it crosscuts in FreeBSD v2 involves stripping the 29 sections of compiler directives from the 17 functions listed in Table 7 (page 37). Refactoring resulted in the creation of several new helper functions to which advice could be attached.

The most significant way in which this aspect differs from the others considered so far is that it regularly aborts file system operations. That is, it is very common to see quota code do something like the following in a file system operation:

```
#ifdef QUOTA
  if (error = getinoquota(ip))
    return (error);
#endif
```

This typically accompanied other code that would detect error conditions and possibly abort as well. To refactor code like this, we created what was essentially an exception handling helper function that would test all error conditions and then call the main body if none of them hold true. We attached around advice for quota to the newly refactored helper function. This way, if there was a quota exception, the around advice could be designed to not proceed to the main body of the aborted function.

4.3 Costs: Microbenchmarks for AspectC Runtime

Like AspectJ, most AspectC constructs are static, that is, resolved at compile time. They introduce no more overhead than a call to an inlineable function containing the advice body. Though the pre-processor could inline these directly, it currently does not, in order to help make the pre-processor output more readable.

But *cflow* is a dynamic construct and hence has runtime overhead associated with it. We follow the AspectJ implementation model for *cflow*, in which the overhead is distributed across executions of join points that are *cflow*-tested.

```
pointcut unqueue_available(vm_page_t m):
    execution(void vm_page_unqueue(m))
    && !cflow(execution(void vm_page_alloc(vm_page_t))
              || execution(void vm_page_free_toq(vm_page_t))
              || execution(void vm_page_cache(vm_page_t)));
```

In the pointcut shown above, a *cflow_push* and *cflow_pop* are effectively added to the code for each of the three VM operations listed. The example below shows how AspectC replaces the original body of *vm_page_cache* with a call to a function that performs the

push, calls the original body, and performs the pop. The process-local stack associated with the *cflow* is specifically dedicated to this pointcut and assigned a unique identifier, *0* in the example below.

```
void vm_page_cache ( vm_page_t m ) {
  ACPP_CFLOW_EXEC_vm_page_cache_1 ( m ) ;
}

void ACPP_CFLOW_EXEC_vm_page_cache_1 ( vm_page_t ACPP_PARAM_3 ) {
  void * tmparray1[] = { & ACPP_PARAM_3  } ;
  cflow_push ( 0 , tmparray1 ) ;
  REAL_EXEC_vm_page_cache1 ( ACPP_PARAM_3 ) ;
  cflow_pop ( 0 ) ;
  return ;
}

void REAL_EXEC_vm_page_cache1 ( vm_page_t m ) {
  /* original body of vm_page_cache */
}
```

A *cflow_test* is effectively added to *vm_page_unqueue*, as part of testing whether the advice should run. The code demonstrating this use of *cflow_test* is shown below. If the advice requires access to parameters from the *cflow*, *cflow_get* is also called.

```
void vm_page_unqueue ( int m ) {
  ACPP_BEFORE_AFTER_EXEC_vm_page_unqueue_1 ( m ) ;
}

void ACPP_BEFORE_AFTER_EXEC_vm_page_unqueue_1 ( int ACPP_PARAM_1 ) {
  REAL_EXEC_vm_page_unqueue1 ( ACPP_PARAM_1 ) ;
  if ((1) && (! (((cflow_test(0)) || (cflow_test (0))) || (cflow_test(0)))))
    ACPP_page_daemon_activation_AFTER_1 ( ACPP_PARAM_1 ) ;
  return ;

}

void REAL_EXEC_vm_page_unqueue1 ( int m ) {
  /* original body of vm_page_unqueue */
}
```

Our current implementation of the push/pop/test/get runtime routines is trivially naive. An open hash table tracks this information on a per-process basis. A pool of entries, sufficiently large to track the maximum number of processes in the system, is statically allocated at boot time. Each entry tracks the necessary *cflow* information for a single process, uniquely identified by the process identifier (PID). Possible alternative implementations for kernel runtime support include the opportunity to statically allocate these stacks in the process table, or access information directly from the runtime stack.

We chose to implement the naïve approach, which is more in keeping with the runtime support available for user-level AspectC, simply because it was the easiest, most modular, implementation.

Table 3 provides micro-benchmarks for our prototype AspectC runtime. These were taken on a 700MHz Pentium-III processor running FreeBSD v4. Baseline tests within this environment report the costs of forking a process from user-level to be 165.2 μseconds, and the roundtrip time of switching between user and kernel mode and back again to be 0.7 μseconds. This roundtrip mode switch time is the minimum overhead associated with all system calls. For system calls that involve going to disk, such as *read,* the roundtrip time from user level is approximate 8000 μseconds.

The first two rows in Table 3 show the costs of adding and deleting hash table entries during process initialization and tear-down. Relative to the cost of forking from user-level, *add_pid* introduces 0.5% additional overhead, the bulk of which is re-initialization of the hash table entry with *bzero*. These costs could be eliminated if we were to allocate stacks statically in the process table, or access the runtime stack directly.

The next four rows in Table 3 show the per-call costs of the *cflow* push/pop/test/get routines. Relative to the null system call, these operations introduce 10-12% additional overhead per *cflow* push/pop/test/get. Relative to the read system call, they add 0.001% each. Though this is not representative of an optimal implementation, as costs could be further reduced by inlining these functions, by storing *cflow* state directly within process data structures to eliminate the hash table lookup on process id, or perhaps even by walking the runtime stack when necessary, it demonstrates that the overheads of even a naive implementation are not prohibitive.

Table 3. Runtime costs, per process and per associated cflow operation.

granularity	cflow function	overhead (μseconds)
per-process	`add_pid`	0.777
	`del_pid`	0.141
per-cflow-op	`push`	0.079
	`pop`	0.080
	`test`	0.086
	`get`	0.073

Even with its currently naïve implementation, the AspectC runtime overheads are not prohibitive for the concerns considered here. These results are not surprising, and do not constitute a conclusive assessment – each of these concerns involves an inherently expensive disk operation that dominates performance. Though more testing has to be done to conclusively assess the impact of AOP on performance, the benefits of improved evolution potentially merit at least small tradeoffs in terms of performance.

4.4 Scale

We now consider the scalability of this approach to more concerns, more versions and more systems.

CLAIM 4: *the modularity of crosscutting concerns and corresponding interacting concerns in OS code can be improved in an AOP implementation without harming non-interacting concerns.*

If FreeBSD is representative of a highly evolved OS code base, and the three concerns are crosscutting and representative, and the modularity of A2 is better than O2, then we can infer these results would scale to some degree, to apply to more concerns, and more systems.

The second claim that involves scalability is as follows:

CLAIM 5: *the modularity benefits are expected to persist across more than 3 versions and more than 1 system*

If the 11 change tasks are representative, and the modularity persists across 3 versions, we can infer that these results would scale to some degree, and apply to more versions and more systems.

Though evolution inevitably entails changes that span concerns, the 3 of the 11 change tasks that involved aspects, interacting concerns, and non-interacting concerns appeared to be no worse in the aspect-oriented kernels. Multiple concerns/modules were involved in each case. The propensity for consistency, coordination, and configurability of crosscutting concerns to carry through multiple successive versions of an OS is much higher in an AOP implementation, where crosscutting structure can be made explicit.

Additional support for these claim can be gathered by examining more OS concerns from the perspective of AOP. Future trends indicate crosscutting alternative resource management strategies require configurability and portability. We consider 3 future concerns in the subsections that follow. Integration and manageability of new concerns is considered further in Future Work (Section 5.1.1).

4.4.1 GMS

The global memory service (GMS) [Feeley, Morgan, *et al.* 1995, Feeley 1996] manages cluster-wide memory, avoiding costly disk accesses by paging from remote memory instead of disk and minimizing the total cost of all memory references in the cluster. Part of the intent of GMS is to capture all points where certain page management decisions are made, and integrate GMS operations as appropriate for that execution context.

4.4.1.1 Key GMS Activities

GMS extends the notion of "oldest", or least recently used page, to the cluster. That is, global information is tracked and used it to make page replacement decisions that are

optimal in a global sense. Idle pages in the network act as backing-store for active nodes through an augmented least recently used criteria that consider them as older. All nodes run the same memory management algorithm and attempt to make choices that balance local and global needs. Any given node contains both local pages and global pages, stored on behalf of other nodes.

GMS was originally implemented by modifying the memory management portion of the OSF/1 operating system on the DEC Alpha platform, and later ported to FreeBSD v2.2 on the i386 platform. A GMS module introduces functionality associated with global pages on the node, and all page-replacement decisions are made by the page daemon in conjunction with GMS. Each kernel was modified by inserting calls to GMS at points where pages are either added to/removed from memory. These calls track information and forward disk reads to the remote GMS on an appropriate node.

4.4.1.2 System Integration

A subset of GMS operations from the i386 implementation are overviewed in Table 20. The entire integration with FreeBSD uses 45 compiler directives spread over 9 files.

Three of the entries in the Table cannot be directly implemented in terms of the primary functionality involved. Their implementation requires refactoring existing functionality in order to facilitate composition with the aspect. These three new functions (marked with *) implement the specific portions of the tangled functionality that the aspect targets. Two of the three cases involve creating small helper functions from code that was originally part of *vm_fault*, which itself is over 700 lines.

For example, the original code to integrate GMS with prefetching code within *vm_fault* involves manipulating variables with local scope and jumping over a portion of the function:

```
  if (object->type != OBJT_DEFAULT && (!change_wiring || wired)) {
    int rv;
    int faultcount;
    int reqpage;
    int ahead, behind;
#ifdef GMS
    if(gms_os_getpage(&m,0,PSIZE,0))){
      ahead = behind = 0;
      rv = VM_PAGER_OK;
      goto gms_getpage_success;
    }
#endif
```

Refactoring *vm_fault* to use a helper function, *vm_prefetch*, parameterizes these otherwise local values, exposing them to around advice which then may or may not proceed with the body of the helper, depending on the result of the GMS *getpage* operation.

Table 20: A subset of GMS operations and where they occur in FreeBSD v2.

GMS Operation	File	Modified Function	Structural Relationship
gms_os_lclru_pg_activated gms_os_lclru_pg_deactivated	i386/i386/pmap.c	pmap_insert_entry pmap_clear_reference	after call to splvm before function body
gms_os_putpage_wait	vm/swap_pager.c vm/vm_page.c vm/vm_pageout.c	swap_pager_putpages vm_page_alloc vm_pageout_scan	before call to pmap_qenter before call to vm_page_remove after call to vm_page_list_find
gms_os_getpage	vm/vm_fault.c vm/vfs_bio.c	vm_fault getblk	around call to vm_prefetch* after call to allocblk*
gms_os_discardpage gms_os_grp_discard	vm/vfs_bio.c vm/vm_fault.c vm/vm_object.c	vfs_vmio_release vm_hold_free_pages vm_fault vm_object_terminate	before call to vm_page_free before call to vm_page_free before call to vm_fault_free* before call to pager_deallocate

* Functions introduced as result of refactoring to facilitate aspect composition.

4.4.1.3 *Implementation*

As previously noted, unlike the page daemon aspect, GMS reacts to actions in the primary modularity with different operations. To further capture this internal structure, we designed several small aspects for GMS, coupling related operations such as activate/deactivate, putpage/getpage and discard page/group discard. Most of the entries in Table 20 can be implemented without modification to the original code. For example:

```
aspect lclru_activation {

  after(vm_offset_t pa): call(int splvm())
  && cflow(execution(int pmap_insert_entry(pmap_t, vm_offset_t, vm_page_t, pa))){
       gms_os_lclru_pg_activated(pa);
  }

  before(vm_offset_t pa): execution(int pmap_clear_reference(vm_offset_t pa))){
       gms_os_lclru_pg_deactivated(pa);
  }
}
```

This aspect captures activation/deactivation, which happens only after the priority level has been set when inserting a *pmap* entry, and deactivation, which happens before the reference has been cleared.

4.4.2 *Resource Containers*

Resource Containers (RCs) [Banga 1999, Banga, Mogul, *et al.* 1999] allow fine-grained resource management in server systems. A resource container is an abstraction that is logically associated with all system resources being used by an application to service a specific activity, such as servicing a particular client connection. By separating the notion of resource principals from processes/protection domains, RCs can control resource consumption at all levels of the system. Part of the intent of RCs is to capture all points in the system where certain resource allocation/release functionality occurs, and to track and coordinate this activity.

4.4.2.1 *Key RC Activities*

RCs were originally implemented in Digital UNIX by Gaurav Banga as part of the ScalaServer project. A port was later developed for FreeBSD by Mohit Aron [Aron 2000]. Consumption of resources is charged to RCs rather than processes, and a dedicated RC module ensures scheduling of resources in terms of RCs. An important feature of the internal structure of RCs is the ability to coordinate charges for single RCs consistently.

4.4.2.2 *System Integration*

Table 21 overviews a subset of Resource Container operations, the modifications required to integrate them with existing functionality, and the corresponding structural relationships between the modifications and the new behaviour.

Table 21: A subset of Resource Container operations and where they occur in FreeBSD v4.0

RC Operation	File	Modified Function	Structural Relationship
rc_proc_runnable	kern/kern_switch.c	set_runqueue	before function body
rc_proc_yield			
rc_proc_block	kern/kern_synch.c	mi_switch	before function body
rc_proc_finish	i386/i386/vm_machdep.c	cpu_exit	before call to cpu_switch
rc_chargecpu	kern/kern_clock.c	Statclock	before function body
rc_attach_socket	kern/uipc_socket.c	socreate	after call to soalloc
	kern/uipc_socket2.c	sonewconn3	before call to soreserve
re_detach_socket	kern/uipc_socket.c	sofree	before call to sodealloc

4.4.2.3 *Implementation*

In the unstructured implementation, modifications to intercept all calls to *set_runqueue* and *mi_switch* involved renaming the original functions *O_set_runqueue* and *O_mi_switch* respectively, and replacing the original function bodies with the resource container code followed by a call to the new functions.

The following AspectC code accomplishes the same outcome, introduces no more overhead than the tangled implementation, and can be specified from within an aspect, without requiring changes to the existing functionality:

```
before(struct proc* p):
  execution(void set_runqueue(p))
     /* if <…> rc_proc_runnable(p)
   else re_proc_yield(p) */
}

before(): execution(mi_switch())
     /* rc_proc_block */
}
```

Using *call* in conjunction with the *cflow* of an *execution* means that advice runs only at selected call cites, such as that associated with the call to *cpu_switch* within the control flow of an execution of *cpu_exit*:

```
before(): call(void cpu_switch())
 && cflow(execution(int cpu_exit(..))) {
     /* rc_proc_finish */
}
```

Further kernel modifications for resource containers not listed in the Table include changes to the *proc* and *socket* structs to introduce additional fields used for binding resources. In *sys/proc.h*, the final two fields in the *proc* struct introduce this support for resource container functionality:

```
struct  proc {
  /* original fields */
  ..
  /* new fields for RCs */
  struct RC *p_res_binding;
  LIST_HEAD(p_sched_binding, SB_Elmt)
     p_sched_binding;
}
```

In keeping with AspectJ, AspectC could augment the existing structs from within an aspect through the following declarations:

```
  struct RC* proc.p_res_binding;
  LIST_HEAD(p_sched_binding, SB_Elmt) proc.p_sched_binding;
```

These fields, and the Resource Container code that manipulates them, can now be kept together in the same modular unit[10].

4.4.3 Bossa

Bossa is a kernel-level event-based framework that facilitates the implementation and integration of new scheduling policies [Barreto, Douence, *et al.* 2002, Lawall, Muller, *et al.* 2002, Muller and Lawall 2002, Muller, Lawall, *et al.* 2003] based on a domain specific language approach [Consel and Marlet 1998, Muller, Consel, *et al.* 2000]. The intent of the Bossa run-time system is to capture all points when scheduling decisions are made, and to raise a specific event type that corresponds to the current execution context.

Scheduling code spans interrupt handlers, device drivers, and all places in the system where process synchronization occurs. One of the challenges in the development of Bossa, a domain specific language for schedulers, is to precisely identify all the scheduling points, or circumstances under which the scheduler is activated, throughout the OS. Extending the scheduler to respond to Bossa-defined scheduling events requires access to the context of the scheduler invocation. To get an idea of how extensive the challenge is to track this context, the number of calls to *tsleep* system-wide is close to 500 in v4. Changing an OS to raise scheduling events thus requires invasive modifications to hundreds of places in the system, compromising the modularity of the extension. Aligning the extension as a scheduling concern structured within an aspect, similar to the device blocking aspect, thus improves the modularity of the extension.

---

[10] If they want, an AspectC programmer can look at the output of the pre-processor, which shows the result of weaving the advice. This woven code will be similar to the original tangled code. Tool support such as that developed for AspectJ in many environments can make looking at the woven code almost never necessary.

4.4.3.1 *Background: Bossa, Scheduling Decisions and Events*

In Bossa, high-level abstractions provided by a domain specific language are designed to simplify policy specification and verification. The Bossa run-time system uses an event-based model to integrate new policies into an existing kernel.

The job of a scheduler is to dispatch, suspend, and terminate processes. Scheduling decisions combine 3 things: attributes associated with processes, criteria used to select a process for execution, and the context in which this process selection takes place. Points where scheduling decisions are made arise under many different circumstances during execution. For example, the arrival of a high priority process in the ready queue of a priority-based scheduler, due either to the creation of a new process or the process becoming runnable because of a timer expiration.

4.4.3.2 *Key Bossa Activities*

Bossa requires that a kernel be augmented with functionality to raise scheduling events associated with process creation, termination, timeout, clock ticks, blocking and unblocking. To allow a policy to define a single behaviour for a collection of related events, the events are organized in a hierarchy that reflects both the type and context of each event. For example, *block.** applies to all blocking events, and *block.io.network* applies only to blocking events from network I/O. The re-implementation of kernel scheduling points to raise events must evolve with each new release of the OS, and should be relatively easy to port across different OSes.

Within a Linux 2.2.16 kernel, this modification crosscuts device drivers, file systems, system calls and kernel daemons. Table 22 summarizes the number of Bossa event notifications per OS subsystem in an implementation of a Bossa/Linux kernel where 14 drivers have been modified.

Table 22: Modifications required in Linux to implement the Bossa event-based notification.

OS subsystem	Event	Mechanisms
Time Service	3	Clocktick, kernel timers
Memory management	6	Page fault
Networking	9	Sockets, TCP/IP statck
File system	14	Buffer cache, VFS
Device drivers	17	Floppy, IDE disk
Process management	19	Fork, synchronization
Total	*68*	

4.4.3.3 System Integration

Table 23 overviews 5 of the 68 events from the Bossa hierarchy presented in [Muller, Lawall, *et al.* 2003]. The columns highlight the files and functions that require modification to raise the event, and the structural relationship the modification has with existing Linux 2.4.18 functionality. These structural relationships are critical, as they precisely describe the execution context in which each event is raised, thereby determining the event type.

Table 23: A subset of Bossa events and where they are raised in Linux v2.4

Bossa Event	File	Modified Function	Structural Relationship
process_new_fork	kernel/fork.c	do_fork	replaces call to wake_up_process
process_new_initial_process	kernel/main.c	start_kernel	after call to kernel_thread
process_yield	kernel/sched.c	sys_sched_yield	before body of function
process_end	kernel/exit.c	release_task	before call to schedule
block	arch/i386/kernel/ptrace.c drivers/block/raid5.c drivers/char/generic_serial.c drivers/char/serial.c drivers/char/tty_io.c fs/buffer.c	syscall_trace _wait_on_stripe block_til_ready block_til_ready do_tty_write create_buffers	In each case, this event is raised before call to schedule.

4.4.3.4 Implementation

Using AspectC, these kinds of structural relationships can be made more explicit and modular. For example, we could structure each of the highest level event types as an aspect:

```
aspect process_event_types {

   around(): call(void wake_up_process(..))
    && cflow(execution(int do_fork(..))) {
        /* raise process_new_fork[11] */
   }

   after(): call(void kernel_thread(..))
    && cflow(execution(void start_kernel(..))){
        /*raise process_new_initial_process*/
   }

   before():
    execution(void sys_sched_yield(..)) {
        /* raise process_yield */
   }

   before(): call(void schedule())
    && cflow(execution(int release_task(..))) {
        /* raise process_end */
   }
}
```

---

[11] Note that replacing the call to *wake_up_process* can be done with *around* advice that does not call *proceed*.

The first advice in a *block* event aspect also targets calls to *schedule*, but within different control flows:

```
aspect block_event_types {

pointcut within_char_serial_dev():
 execution(int block_til_ready(..))
 && (within("drivers/char/generic_serial.c")
     || within("drivers/char/serial.c"));

before(): call(void schedule())
 && (cflow(execution(int syscall_trace(..))
       || execution(int _wait_on_stripe(..))
       || execution(int do_tty_write(..))
       || execution(int create_buffers(..)))
     || within_char_serial_dev()) {
     /* raise block event */
}
…
```

4.4.4 Challenges with Existing Concerns

Existing concerns that could to benefit from improved locality of change include additions to existing concerns such as the changes required to conform with NFS v3 given NFS v2, and security for communication in protocols such as IP. As benefits associated with aspect-oriented programming have been previously established, here we precisely identify some of the challenges associated with refactoring a tightly tangled implementation for better separation as aspects, and the tradeoffs involved.

With respect to version 2 versus version 3 of FreeBSD's implementation of the Network File System (NFS) [Osadzinski 1988], the NFS v2 code base is approximately 10,000 lines, to which the integration of v3 adds over 100 small, scattered clusters of code, differentiated from v2 by appropriate compiler-directives and system-wide identifiers. Similarly, FreeBSD's implementation of IPv4 security is configured through 39 *#ifdef*s throughout IPv4 code. Perhaps even more so than file systems, protocol modules tend to have fairly narrow interfaces, resulting in some coarse grained functional decompositions, with potential for high degrees of tangling. Drawing from the source

code for IP security, examples associated with refactoring highly tangled existing concerns such as these include:

The use of *gotos* within IPSEC:

```
#ifdef IPSEC
  if (ipsec_gethist(m, NULL))
    goto pass;
#endif
```

Tight integration with local variables and control structures:

```
#ifdef IPSEC
  if (n && ipsec4_in_reject_so(n,
             last->inp_socket)) {
    m_freem(n);
    ipsecstat.in_polvio++;
  } else
#endif /*IPSEC*/
```

Interleaving with other *#ifdef*s that also access local variables and use gotos:

```
#ifdef IPSEC
#ifdef INET6
  if (isipv6) {
    if (inp != NULL &&
        ipsec6_in_reject_so(m,
          inp->inp_socket)) {
      ipsec6stat.in_polvio++;
      goto drop;
    }
  } else
#endif /* INET6 */
  if (inp != NULL && ipsec4_in_reject_so(m,
                       inp->inp_socket)) {
    ipsecstat.in_polvio++;
    goto drop;
  }
#endif /*IPSEC*/
```

Fine grain calculations amidst large functions:

```
#ifdef IPSEC
  ipoptlen += ipsec_hdrsiz_tcp(tp);
#endif
```

When refactoring existing concerns such as these, the modularity benefits gained by refactoring must outweigh the effort associated with retroactive separation. Crosscutting concerns that are highly scattered and tangled in the original implementation may not be beneficial to separate due not only to the refactoring costs, but also to a lack of significant improvement in consistency, coordination, or configurability in the aspect-oriented implementation. In particular, in the case of NFS v3, the issue of further extending the aspect-oriented implementation for NFS v4 offers a possible challenge problem for future work (Section 5.1.1). The possible trade off involves modularity benefits versus the complexity of reasoning about the composition.

4.5 Inference of Main Claim

We claim that AOP can be used to improve evolvability of OS code by supporting better modularity of crosscutting concerns and interacting concerns without harming non-interacting concerns. We base this inference on the generalizations of scale presented in the previous subsections. Specifically, if we expect modularity benefits to scale to more concerns, more versions and more systems, then we can infer improvements to evolution as we did for the three concerns and the 3 versions.

Important questions remain unanswered as to the degree to which the number of aspects can be increased, and if this will eventually cause problems with comprehensibility or aspect interaction in the system. In the event this happens, we need to identify the nature of the problems with real world examples, and investigate the ability of tool support and possibly further language mechanisms to address these problems.

5 Summary and Conclusions

In Chapter 1 we introduced the argument structure for the thesis. We include this overview again in Figure 20 for convenience of reference. Our experimental set up was presented in Chapter 1. Chapters 2 and 3 presented case studies associated with Claims 1 and 2 respectively. Results of these studies are primarily qualitative and observation based. Chapter 4 presented the remaining claims – inferences of evolution and scale. We now summarize support for these claims, highlighting the analysis, observations and issues with each, as they work in support of the main claim of the thesis.

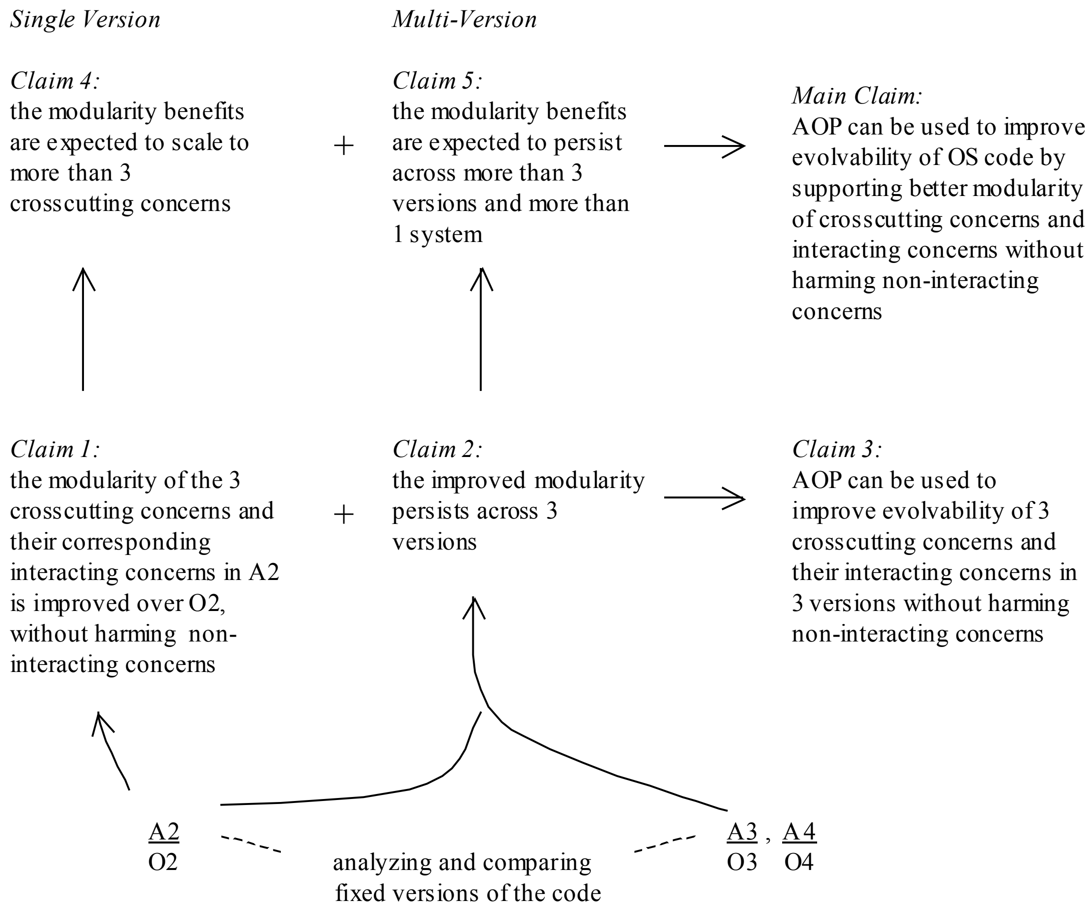

Figure 20: Claim structure for the dissertation.

***Claim 1**: The modularity of the 3 crosscutting concerns and their corresponding interacting concerns in A2 is improved over O2, without harming non-interacting concerns.*

The analysis performed to support this claim was the case study comparing the implementation of the three crosscutting concerns in O2 versus A2.

Observations included the ways in which the implementation of the three crosscutting concerns in O2 is non-modular, and their presence introduces subsystem violations, redundant code, and dynamic context passing, compromising the modularity of the interacting concerns as well. Improvements in A2 with respect to the structure of the aspects include locality, common abstractions with the same naming/factoring, less redundant code, narrower interfaces, and explicit non-local interactions, such as detecting

an execution sequence, passing a value down a call chain, and enforcing consistency of policy across parallel functionality. We showed that these improvements not only impacted the modularity of the crosscutting concerns, but also simultaneously improved the modularity of the interacting concerns in A2 as a result of better separation, and did not harm non-interacting concerns.

Two issues arose from the case study associated with this claim. First, the degree to which the comparison is fair, and second, the degree to which we can be sure non-interacting concerns are not harmed.

For fairness of comparison, we relied on the fact that FreeBSD is a high quality implementation, and that the code base represents one of the best possible implementations of these concerns given traditional techniques.

In terms of safety of non-interacting concerns, we looked at two levels of insurance. First, we examined the precision of pointcut definitions and the ability to reason about the composition of the aspect with the system when looking at the aspect alone. Second, we discussed the kind of additional support tools in AOP can provide, but as AspectC does not have tool support, we relied on manual inspection of the woven code and testing.

***Claim 2**:* *The improved modularity persists across 3 versions.*

The analysis performed to support this claim was the case study identifying change tasks impacting the three crosscutting concerns over successive versions of FreeBSD that compared the number of files modified as a result of change in both aspect-oriented and original implementations, and established the impact of the changes on modularity.

Observations made were that localization of change was better in A2 and A3 than in O2 and O3. In the original kernels, all change tasks involving the crosscutting concern require changes to interacting concerns. In the aspect-oriented kernels, 3 of the 11 change tasks required changes to both crosscutting and interacting concerns, 7 change tasks

required changes to just one of the crosscutting or interacting concerns, and 1 change task would arguably not have been required.

The key issue discussed with respect to this claim involved foresight in the AOP implementation, in that we knew the change tasks in advance. In order to more fairly assess an AOP implementation, we provided possible alternative implementations, along with their associated impact, and identified why the one we choose had high likelihood of being chosen by a developer without foresight.

Other issues included the small size of the subset of change tasks, and the number of files involved when change tasks span concerns. The fact that we only looked at a subset of changes associated with evolution prevents us from making conclusions about the process of complete evolution, but not the impact of these particular changes. Change tasks that span concerns can involve more files in the aspect-oriented implementation, but we argued that this was appropriate given that the changes included multiple concerns – either new features or integration of old features that impact aspects, interacting, and non-interacting concerns.

***Claim 3**: AOP can be used to improve evolvability of 3 crosscutting concerns and their interacting concerns in 3 versions without harming non-interacting concerns.*

We supported this claim by inference from previously supported claims. Specifically, if evolution of non-modular concerns is hard, and we have improved modularity in the aspect-oriented implementation, and this improved modularity persists across 3 versions, then we can infer that evolution would have been easier in an aspect-oriented implementation due to better modularity.

By way of analysis, we identified common sources of modularity improvements for all three concerns and their interacting concerns. Our observations indicated that localization of both functionality and interaction supported explicit coordination between activation points for the page daemon, execution paths for prefetching, and parallel functionality for disk quota. Simultaneous improvements in modularity of interacting

concerns resulted from the removal of subsystem violations introduced by the crosscutting concerns.

Issues that arose from this claim involved the small sample size, the inability to structure for unanticipated change, and both adoption and costs of AOP. In terms of sample size, we argued that since the traits were common in all 3 aspects, we believe they will generalize. With respect to unanticipated change, we showed that by using AOP to more effectively structure according to design intent, changes in interacting concerns that are consistent within that intent do not require corresponding changes to the crosscutting concerns – as was the case in 4 of the 11 change tasks. Finally, issues of adoption and runtime costs were presented, demonstrating how tool support can help ensure non-interacting concerns are unharmed, showing that a naïve implementation has low performance costs, and that a more optimal implementation appears reasonable.

Claim 4*: The modularity benefits are expected to scale to more than 3 crosscutting concerns.* We supported this claim primarily by generalization of the results of the case study associated with Claim 1. If FreeBSD is representative of OS code, and the three concerns are crosscutting and representative, and the modularity of the aspect-oriented implementation is better than the original implementation, these results should generalize and scale to apply to more concerns and more systems.

We further enforced this argument by looking at structure, portability and manageability of future concerns in more systems. Our observations regarding future trends indicate crosscutting alternative resource management strategies require configurability and portability: GMS crosscuts VM and file buffer cache, instruments kernels using compiler directives, and has been ported from OSF/1 to FreeBSD; Resource containers crosscut networking and process management, instruments kernels with loadable modules and LOC, and has been ported from Digital Unix to FreeBSD; and Bossa crosscuts VM, networking, filesystems, device drivers, and process management, instruments kernels with LOC to raise events, and is being ported from Linux to FreeBSD, and soon to

Windows NT. We showed how each of these strategies has internal structure, and how their non-modular integration makes that structure difficult to manage effectively.

Issues discussed with respect to this claim involve the comprehensibility of increasing numbers of aspects and the potential for interaction. Though we could not fully address this issue, we noted that the three concerns in the case studies had common interacting concerns – both page daemon activation and prefetching crosscut VM, and both prefetching and quota crosscut file systems – and interaction was not a problem. The increasing importance for tool support in AOP under these conditions was also discussed.

Claim 5*: the modularity benefits are expected to persist across more than 3 versions and more than 1 system.*

We supported this primarily by generalization of the results of the case study associated with Claim 2. Specifically, if FreeBSD is representative of OS code, and the 11 change tasks are representative of change, and modularity improvements persist across versions in the aspect-oriented implementation, then these results should generalize and scale to apply to more versions and more systems.

By way of analysis, we considered structure, integration and manageability of additions to existing concerns in more systems. Our observations showed that within several existing subsystems, upgrades such as moving from NFSv2 to v3, and incorporating security strategies into an IP stack, require incremental and configurable adoption, currently provided by compiler directives, global variables, and entire alternative branches of newly integrated source code. Though these additions have internal structure, their non-modular implementations make that structure difficult to manage effectively, and further compromise modularity of the existing of subsystems they are tightly tangled within.

Issues discussed involved the inevitably of system evolution to entail changes that span multiple concerns, and additions to eventually become standard. We conclude that,

based on the 3 of the 11 change tasks that involved aspects, interacting concerns, and non-interacting concerns in our case studies, these comprehensive changes appeared to be no worse in the aspect-oriented kernels.

> ***Thesis Statement**: AOP can be used to improve evolvability of OS code by supporting better modularity of crosscutting concerns and interacting concerns without harming non-interacting concerns.*

We ultimately supported the thesis of this work through inference based on the scalability of improvements in modularity to more concerns, more versions and more systems. Though we did not perform a study on the process of evolution of a full system, we believe these results are a good first step towards understanding the material impact of AOP in systems code.

5.1 Conclusions

Evolving OS code is hard. It involves extending, integrating, optimizing, re-optimizing, and maintaining system functionality and requires understanding not only the concerns within the system, but their interactions – which are inherently complex. Modularity aids evolution by providing structure for comprehensibility and locality of change. It is easier to evolve a system if responsibilities are better separated between concerns.

This dissertation shows support for the claim that aspect-oriented programming can be used to improve evolvability in operating system code by providing better modularity. We presented a case study showing improved modularity when three crosscutting concerns in FreeBSD were implemented using AspectC. We also presented a case study showing improved locality of change when the three crosscutting concerns were exposed to change tasks over two evolutionary periods, and showed that improvements in modularity persisted across 3 versions of the system. We inferred from these results that evolution of the aspect-oriented kernels would be easier than the original, due to improved modularity. We then generalized these results to the modularity of more

concerns, more versions and more systems in support of our main claim – that aspect-oriented programming can be used to improve evolvability in operating system code by providing better modularity of crosscutting concerns and interacting concerns without harming non-interacting concerns.

5.1.1 Future Work

In its current state, AspectC is still a prototype that has been released to several developers strictly for experimental purposes. Several features, such as wildcards for pointcut declarations, remain to be implemented, along with optimizations for runtime support. Its source is soon to be publicly available, and several developers have already expressed interest in enhancing its current implementation.

Three key steps for future work that could build on the results in this thesis, and each other, include targeting more existing concerns in current systems, integrating of new concerns into current systems, and finally, designing a fully aspect-oriented operating system from the ground up. Additionally, realistic evolutionary studies including the process developers go through, and the need to establish metrics to show benefits associated with improved modularity, are two further areas that require work.

The advantage of continued refactoring of existing concerns is that we can study scalability and performance relative to the original implementation. Verifying behaviour, analyzing interaction, examining the development process, and isolating overheads are all import next steps that may best be accomplished in this setting. Performance is of particular interest within more fine-grained concerns such as scheduling. To establish the tradeoffs, it is important that this evaluation be done in a setting where the aspects can go head-to-head with a tradition implementation. Several projects are already underway in this arena, including µDyner [S'egura-Devillechaise, Jean-MarcMenaud, *et al.* 2003], used to dynamically weave prefetching policies into web

caches, and Task Scheduler Logic (TSL) [Reid and Regehr 2003] used for reasoning about scheduling and concurrency.

Introducing new functionality to existing systems requires the ability to manage diversity effectively. Competing strategies for resource management may be more apparent, and hence more manageable, with explicit crosscutting structure. The need to build more crosscutting structure upon crosscutting structure, as could be the case with NFS v3 and v4, offers many open avenues for future work.

Rolling forward with a new systems from scratch would inevitably incorporate lessons learned from the previous two steps, more refactoring of existing systems and more experience with new concerns. Establishing what interfaces would look like when a system's primary modularity has been designed specifically to accommodate aspects would revisit the tension Lampson noted between of exposure of power and hidden functionality.

Within each of these steps, further understanding the differences in the process of evolution in an aspect-oriented implementation versus the original could be gained through developer studies. Metrics that establish improvements might then be measurable in terms of number of changes required, number of bugs introduced, and amount of time required to execute change tasks associated with evolution.

5.1.2 Contributions

This dissertation makes three contributions.

First, it shows how the non-modular implementation of three crosscutting concerns in OS code currently compromise evolvability.

Second, it shows how AspectC can be used to improve modularity of three crosscutting concerns and their interacting concerns in OS code, without harming non-interacting concerns, and that this modularity persists over two subsequent versions of the system.

From this we inferred that AOP can be used to improve evolvability of the three concerns and their interacting concerns without harming non-interacting concerns.

Third, it shows that the benefits of modularity should generalize to more than the three concerns, and persist over more than 3 versions of an OS. From this we inferred that AOP can be used to improve evolvability of OS code by supporting better modularity of crosscutting concerns and interacting concerns without harming non-interacting concerns.

Bibliography

M. Aksit and B. Tekinerdogan, *Solving the Modeling Problems of Object-Oriented Languages by Composing Multiple Aspects Using Composition Filters*, OOPSLA AOP'98 workshop position paper, 1998.

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 (TOCS), 14(1), 1996, pp. 41--79.

Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska and Henry M. Levy, *Scheduler activations: Effective kernel support for user-level management of parallelism*, ACM Transactions on Computer Systems (TOCS), 10(1), 1992.

AOSD.net, http://aosd.net/, 2003.

Apple.com, *The Power of Unix*, http://www.apple.com/macosx/jaguar/unix.html, 2003.

Mohit Aron, *Resource containers and LRP*, http://www.cs.rice.edu/CS/Systems/ScalaServer/code/rescon-lrp/README.html, 2000.

John W. Backus, Friedrich L. Bauer, Julien Green, C. Katz, John McCarthy, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden and Michael Woodger, *Report on the algorithmic language ALGOL 60*, Communications of the ACM, 3(5), 1960.

Gaurav Banga, Operating System Support for Server Applications, Computer Science, Rice University, PhD thesis, 1999.

Gaurav Banga, Jeffrey C. Mogul and Peter Druschel, *Resource containers: A new facility for resource management in server systems*, Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI), 1999.

L.P. Barreto, R. Douence, G. Muller and M. Südholt, *Programming OS Schedulers with Domain-Specific Languages and Aspects: New Approaches for OS Kernel Engineering*, Workshop on Aspects, Components, and Patterns for Infrastructure Software at Aspect-Oriented Software Development, 2002.

Computer Systems Research Group UC Berkeley, *4.4BSD-Lite CD-ROM Companion*, The USENIX Association and O'Reilly and Associates Incorporated, 1994.

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*, Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15), 1996.

J. Bieman and L. Ott, *Measuring Functional Cohesion*, IEEE Transactions on Software Engineering, 20(8), 1994.

James M. Bieman and Byung-Kyoo Kang, *Measuring Design-Level Cohesion*, Software Engineering, 24(2), 1998.

Richard R. Burton, L.M. Masinter, Daniel G. Bobrow, Willie Sue Haugeland, Ronald M. Kaplan, B.A. Sheil and Alan Bell, *Interlisp-D Overview*, Xerox PARC SSL-80-4, 1981.

Pei Cao, *LRU-SP: An Allocation Algorithm For Application-Controlled Caching*, Proceedings of the 1997 Grace Hopper Celebration of Women in Computing Conference, 1997.

Pei Cao, Edward W. Felten and Kai Li, *Application-Controlled File Caching Policies*, Proceedings of the USENIX Summer Technical Conference, 1994.

CDT, *Eclipse C/C++ Development Tools*, http://www.eclipse.org/cdt/, 2003.

Yvonne Coady, Gregor Kiczales, Mike Feeley and Greg Smolyn, *Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code*, Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on th Foundations of Software Engineering (FSE-9), 2001.

C. Consel and R. Marlet, *Architecturing software using a methodology for language development*, Proceedings of the 10th International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP/ALP), 1998.

O.J. Dahl and K. Nygaard, *SIMULA- An Algol Based Simulation Language*, Communications of the ACM, 9(9), 1966.

E.W. Dijkstra, *The structure of THE-multiprogramming system*, Communications of the ACM, 11(5), 1968.

Edsger W. Dijkstra, *Go To Statement Considered Harmful*, Communications of the ACM, 1968.

Edsger W. Dijkstra, *A Discipline of Programming*, Englewood Cliffs, United States: Prentice Hall, 1976.

Peter Druschel, *Efficient Support for Incremental Customization of OS Services*, Proceedings of the Third International Workshop on Object Orientation in Operating Systems, 1993.

Peter Druschel, Vivek S. Pai and Willy Zwaenepoel, *Extensible Kernels are Leading OS Research Astray*, Proceedings of the Sixth Workshop on Hot Topics in Operating Systems (HotOS-VI), 1997.

Peter Druschel, Larry L. Peterson and Norman C. Hutchinson, *Beyond Microkernel Design: Decoupling Modularity and Protection in Lipto*, Proceedings of the Twelfth International Conference on Distributed Computing Systems, 1992.

Eclipse.org, *The Eclipse Home Page*, www.eclipse.org, 2003.

John K. Edwards and Pei Cao, *User-Oriented Resource Scheduling in UNIX*, University of Wisconsin CS-TR-96-1318, 1996.

T.J. Emerson, *A Discriminant Metric for Module Cohesion*, Proceedings of the 7th International Conference on Software Engineering (ICSE-7), 1984.

Dawson Engler, Benjamin Chelf, Andy Chou and Seth Hallem, *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*, Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI), 2000.

Dawson R. Engler and M. Frans Kaashoek, *The Exokernel approach to extensibility (panel statement)*, Proceedings of the 1st USENIX Symposium on Operating System Design and Implementation (OSDI), 1994.

Dawson R. Engler and M. Frans Kaashoek, *Exterminate all operating system abstractions*, Proceedings of the 5th Workshop on Hot Topics in Operating Systems HotOS-V, 1995.

Dawson R. Engler, M. Frans Kaashoek and James O'Toole Jr., *Exokernel: an Operating System Architecture for Application-Level Resource Management*, Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), 1995.

M.J. Feeley, Implementing Global Memory Management in a Workstation Network, Computer Science, University of Washington, Ph.D. Thesis, 1996.

M.J. Feeley, W.E. Morgan, F.H. Pighin, A.R. Karlin, H.M. Levy and C.A. Thekkath, *Implementing Global Memory Management in a Workstation Cluster*, In Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), 1995.

Marc Fiuczynski and Brian Bershad, *An Extensible Protocol Architecture for Application-Specific Networking*, Proceedings of the 1996 Winter USENIX Conference, 1996.

Martin Fowler, John Brant, William Opdyke and Don Roberts, *Refactoring: Improving the Design of Existing Code*, Addison–Wesley, 2000.

Gideon Glass and Pei Cao, *Adaptive Page Replacement Based on Memory Reference Behavior*, Proceedings of SIGMETRICS, 1997.

W. G. Griswold, Y. Kato and J. J. Yuan, *AspectBrowser: Tool Support for Managing Dispersed Aspects*, Department of Computer Science and Engineering, University of California, San Diego CS99-0640, 1999.

Jan Hannemann and Gregor Kiczales, *Overcoming the Prevalent Decomposition in Legacy Code*, Advanced Separation of Concerns Workshop at the Internation Conference on Software Engineering, 2001.

W. Harrison and H. Ossher, *Subject-Oriented Programming (a critique of pure objects)*, Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications, 1993.

Norm Hutchinson and Larry Peterson, *The x-Kernel: An architecture for implementing network protocols*, IEEE Transactions on Software Engineering, 17(1), 1991.

Daniel H. H. Ingalls, *The Smalltalk-76 Programming System Design and Implementation*, In the Proceedings of the Fifth ACM Symposium on Principles of Progamming Languages, 1978.

M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti and Kenneth Mackenzie, *Application Performance and Flexibility on Exokernel Systems*, Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP), 1997.

Gerry Kane, *MIPS RISC Architecture*, Prentice-Hall, 1988.

A Kay, *The Early History of Smalltalk*, In Proceedings of 2nd ACM SIGPLAN History of Programming Languages Conference, ACM SIGPLAN Notices, 28(3), 1993.

Brian Kernighan and Dennis Ritchie, *The C Programming Language*, Prentice Hall, First Edition, 1978.

Brian Kernighan and Dennis Ritchie, *The C Programming Language*, Prentice-Hall Software Series, Second Edition, 1988.

Gregor Kiczales, *Towards a New Model of Abstraction in Software Engineering*, Proceedings of IMSA'92 Workshop on Reflection and Meta-level Architectures, 1992.

Gregor Kiczales, *Beyond the Black Box*, IEEE Software, 13(1), 1996.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold, *An overview of AspectJ*, Proceedings of 15th European Conference on Object-Oriented Programming (ECOOP), 2001.

Gregor Kiczales, John Lamping, Chris Maeda, David Keppel and Dylan McNamee, *The Need for Customizable Operating Systems*, Proceedings of the Fourth Workshop on Workstation Operating Systems, 1993.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin, *Aspect-Oriented Programming*, European Conference on Object-Oriented Programming (ECOOP), 1997.

Gregor Kiczales, Jim des Rivieres and D Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

S.R. Kleiman, *Vnodes: An Architecture for Multiple File System Types in Sun UNIX*, Proceedings of the 1986 USENIX Conference, 1986.

Keith Krueger, David Loftesness, Amin Vahdat and Thomas Anderson, *Tools for the developement of application-specific virtual memory management*, Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA), 1993.

A. Lakhotia, *Rule-Based Approach to Computing Module Cohesion*, Proceedings of the 15th International Conference on Software Engineering, 1993.

B.W. Lampson and R.S. Sproull, *An open operating system for a single user machine*, Operating Systems Review, 13(5), 1979.

Butler W. Lampson, *Hints for Computer System Design*, Operating Systems Review, 15(5), 1983.

J. Lawall, G. Muller and L.P. Barreto, *Capturing OS expertise in an Event Type System: the Bossa experience*, Proceedings of the ACM SIGOPS European Workshop, 2002.

Edward K Lee and Chandramohan A. Thekkath, *Petal: Distributed Virtual Disks*, ASPLOS, 1996.

Edward K. Lee and Chandramohan A. Tekkath, *Frangipani: A Scalable Distributed File System*, Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), 1997.

Greg Lehey, *The Complete FreeBSD*, Walnut Creek, 3rd Edition, 1999.

L.L. Lehman and L.A. Belady, *Program Evolution*, APIC Studies in Data Processing, Volume 3, 1985.

Barbara Liskov, *A History of CLU*, Massachusetts Institute of Technology MIT-LCS-TR-561, http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-561.pdf, 1992.

Barbara Liskov and Stephen Zilles, *Programming with Abstract Data Types*, Proceedings of ACM SIGPLAN Conference on Very High Level Languages, 1974.

D. MacQueen, *Modules for Standard ML*, University of Edinburgh ECS LFCS 86-2, 1986.

C. Maeda and B. Bershad, *Service without Servers*, Proceedings of the Fourth Workshop on Workstation Operating Systems, 1993.

Chris Maeda, *Flexible System Software Through Service Decomposition*, OOPSLA, 1994.

Chris Maeda, Service Decomposition: A Structuring Principle for Flexible, High Performance Operating Systems, Computer Science, CMU, PhD Thesis, 1997.

P. Maes, *Concepts and Experiments in Computational Reflection*, Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 1987.

Dylan McNamee and Katherine Armstrong, *Extending the Mach external pager interface to allow user level page replacement policies*, University of Washington UWCSE 90-09-05, 1990.

John Mitchell and Monty Zukowski, *A complete GNU C parser and translator*, http://www.antlr.org/resources.html, 2003.

G. Muller, C. Consel, R. Marlet, L.P. Barreto, F. Mérillon and L. Réveillère, *Towards Robust OSes for Appliances: A New Approach Based on Domain-Specific Languages*, Proceedings of the ACM SIGOPS European Workshop, 2000.

G. Muller, J. Lawall, L.P. Barreto and J.F Susini, *A framework for simplifying the development of kernel schedulers: design and performance evaluation*, Ecole des Mines de Nantes, Technical report 03/02/INFO, 2003.

G. Muller and J.L. Lawall, *Towards a Scheduling Framework for Dynamically Downloaded Multimedia Applications*, Proceedings of the Microsoft Summer Research Workshop, 2002.

Gail C. Murphy, Lightweight Structural Summarization as an Aid to Software Evolution, Computer Science, University of Washington, PhD Thesis, 1996.

Gail C. Murphy, Albert Lai, Robert J. Walker and Martin P. Robillard, *Separating features in source code: An exploratory study*, Proceedings of the 23rd International Conference on Software Engineering (ICSE), 2001.

William F. Opdyke, Refactoring Object-Oriented Frameworks, Department of Computer Science, University of Illinois at Urbana–Champaign, PhD Thesis, 1992.

A Osadzinski, *The Network File System (NFS)*, Computer Standards & Interfaces, 8, 1988.

Harold Ossher, William Harrison, Frank Budinsky and Ian Simmonds, *Subject-Oriented Programming: Supporting Decentralized Development of Objects*, Proceedings of the 7th {IBM} Conference on Object-Oriented Technology, 1994.

Jens Palsberg, Cun Xiao and Karl Lieberherr, *Efficient Implementation of Adaptive Software*, Transactions on Programming Languages and Systems, 17(2), 1995.

D.L. Parnas, *On the Criteria To Be Used in Decomposing Systems into Modules*, Communications of the ACM, 15(12), 1972.

David Lorge Parnas and Paul C. Clements, *Software State-of-the-Art: Selected Papers*, in T. DeMarco and T. Lister, eds., A rational design process: How and why to fake it., Dorset House Publishing., 1990.

Terence Parr, *ANother Tool for Language Recognition (ANTLR)*, www.antlr.org, 2003.

Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky and Jonathan Chew, *Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures*, Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, 1987.

D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray and S. C. Purcell, *Pilot: An operating system for a personal computer*, ACM Operating Systems Review, SIGOPS, 1979.

D.P. Reed, J.H. Saltzer and D.D. Clark, *Active Networking and End-to-End Arguments*, IEEE Network, 1998.

Alastair Reid and John Regehr, *Task/Scheduler Logic: Reasoning about Concurrency in Component-Based Systems Software*, Workshop on Aspects, Components and Patterns for Infrastructure Software, at the 2nd International Conference on Aspect-Oriented Software Development, 2003.

Martin P. Robillard and Gail C. Murphy, *Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies*, Proceedings of the 24th International Conference on Software Engineering (ICSE), 2002.

Frank Rubin, *'GOTO Statement Considered Harmful' Considered Harmful*, Communications of the ACM, 30(3), 1987.

Alex Salkever, *Windows XP: A Firewall for All*, Business Week Online, 2001.

J.H. Saltzer, D.P. Reed and D.D. Clark, *End-to-End Arguments in System Design*, ACM Transactions on Computer Systems (TOCS), 2(4), 1984.

Stefan Savage, *Comment in Session 4b: The Thin Red Line*, Digest of Proceedings of the Seventh IEEE Workshop on Hot Topics in Operating Systems, 1999.

Marc S'egura-Devillechaise, Jean-MarcMenaud and Gilles Muller, *Web Cache Prefetching as an Aspect: Towards a Dynamic-Weaving Based Solution*, Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), 2003.

Christopher Small and Margo Seltzer, *A Comparison of OS Extension Technologies*, Proceedings of the USENIX Conference, 1996.

Brian Cantwell Smith, Procedural Reflection in Programming Languages, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, PhD Thesis, 1982.

W.P. Stevens, G.J. Meyers and L.L. Constantine, *Structured Design*, IBM Systems Journal, Volume 13, 1974.

P.L. Tarr, et al., *N Degrees of Separation: Multi-Dimensional Separation of Concerns*, Proceedings of the International Conference on Software Engineering (ICSE), 1999.

D.L. Tennenhouse and D.H. Wetherall, *Towards an Active Network Architecture*, ACM Computer Communications Review, 26(2), 1996.

Lance Tokuda and Don Batory, *Evolving object-oriented designs with refactorings*, In Proceedings14th IEEE International Conference on Automated Software Engineering, 1999.

UBC, *AspectC Homepage*, www.cs.ubc.ca/labs/spl/aspects/aspectc.html, 2002.

Uresh Valhalia, *UNIX Internals, The New Frontiers*, Prentice Hall Inc, 1996.

Michael VanHilst and David Notkin, *Decoupling Change from Design*, Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1996.

Alistair C. Veitch and Norman C. Hutchinson, *Kea - A Dynamically Extensible and Configurable Operating System Kernel*, Proceedings of the 1996 Third International Conference on Configurable Distributed Systems (ICCDS), 1996.

Alistar Veitch, A Dynamically Reconfigurable and Extensible Operating System, University of British Columbia, PhD Thesis, 1998.

Alistar Veitch, *A conversation after several glasses of wine at an ASI reception.*, 1999.

Werner Vogels, *File System Usage in Windows NT 4.0*, 17th ACM Symposium on Operating System Principles (SOSP), 1999.

R. Walker and G. Murphy, *Implicit Context: Easing Software Evolution and Reuse*, Proceedings of the Conference on Foundations of Software Engineering (FSE), 2000.

Robert James Walker, Essential Software Structure through Implicit Context, Computer Science, University of British Columbia, PhD thesis, 2003.

Niklaus Wirth, *Programming in Modula-2*, Springer-Verlag, 3rd, 1985.

W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson and F. Pollack, *Hydra: The kernel of a multiprocessor operating system*, Communications of the ACM, 17(6), 1974.

W. Wulf and Mary Shaw, *Global variable considered harmful*, SIGPLAN Notices, 8(2), 1973.

Yasuhiko Yokote, *The apertos reflective operating system: The concept and its implementation*, Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA), 1992.

APPENDIX

This appendix provides a sample listing of the page daemon activation aspect used in the case studies. This should not be considered a how-to of aspect-oriented programming, as the AspectC prototype did not support a full range of pointcut mechanisms. Where appropriate, portions of the aspect is shaded and identified as begin specific to aspect-oriented implementations in FreeBSD v2 (A2) or v4 (A4). Non-shaded code is common to all versions.

```
/* helper functions */
int pages_available()    { return cnt.v_free_count    + cnt.v_cache_count; }
int vm_page_threshold()  { return cnt.v_free_reserved + cnt.v_cache_min;   }
int vfs_page_threshold() { return cnt.v_free_min      + cnt.v_cache_min;   }


aspect pageout_daemon_activation {

  pointcut allocate_swap():
    execution(void swap_free_needed_wakeup())
    && cflow(execution(int swap_pager_get_pages(vm_object_t, vm_page_t, int))
             || execution(int swap_pager_putpages(vm_object_t, vm_page_t, int,
                             boolean_t, int*))
             || execution(void swap_pager_sync()));

  pointcut free_needed():
    execution(void swap_free_needed_wait())
    && cflow(execution(int swap_pager_putpages(vm_object_t, vm_page_t, int,
                                               boolean_t, int*)));
  pointcut free():
    execution(int tsleep(void*, int, char*, int))
    && cflow(execution(void swap_free_needed_wait()));

  pointcut io():
    execution(void swap_pager_iodone(struct buf*));
```

A2 only

```
  pointcut unqueue(vm_page_t m):
    execution(void vm_page_unqueue(m))
    && !cflow(execution(void vm_page_alloc(vm_page_t))
              || execution(void vm_page_free_toq(vm_page_t))
              || execution(void vm_page_cache(vm_page_t, int)));
```

```
pointcut allocate_pages(vm_object_t object, vm_pindex_t pindex, int page_req):
  execution(vm_page_t vm_page_alloc(object, pindex, page_req));

pointcut fault(int rbehind, int rahead):
  execution(boolean_t vm_pager_has_page(vm_object_t, vm_pindex_t, int*, int*))
  && cflow(execution(int vm_fault_additional_pages(vm_page_t, rbehind, rahead,
                                                    vm_page_t*, int*)));

pointcut allocate_buffers(vm_object_t obj, vm_pindex_t pindex):
  execution(vm_page_t vm_page_lookup(obj, pindex))
  && cflow(execution(int allocbuf(struct buf*, int)));

pointcut sleep_in_wait(void* count, int where, char* fname, int flag):
  execution(int tsleep(count, where, fname, flag))
  && cflow(execution(void vm_wait()))
  || execution(int asleep(count, where, fname, flag))
     && cflow(execution(void vm_await()));
```

A4 only

```
after(): allocate_swap() || io() {
  if (swap_pager_needflags & SWAP_FREE_NEEDED_BY_PAGEOUT) {
    swap_pager_needflags &= ~SWAP_FREE_NEEDED_BY_PAGEOUT;
    pagedaemon_wakeup();
  }
}
```

A2 only

```
before(): free_needed()
{
   pagedaemon_wakeup();
}

after(): free()
{
   pagedaemon_wakeup();
}

after(): io()
{
  if (!swap_pages_pending_clean() ||
      ((cnt.v_free_count + cnt.v_cache_count) < cnt.v_free_min &&
      nswiodone + cnt.v_free_count + cnt.v_cache_count >= cnt.v_free_min))
    pagedaemon_wakeup();
}

around(vm_page_t m): unqueue(m)
{
  int queue = m->queue;
  proceed(m);
  if (((queue - m->pc) == PQ_CACHE) && (pages_available() < vm_page_threshold()))
    pagedaemon_wakeup();
}

around(vm_object_t object, vm_pindex_t pindex, int page_req):
  allocate_pages(object, pindex, page_req)
{
  vm_page_t allocd_page = proceed(object, pindex, page_req);
  if (allocd_page == NULL)
    pagedaemon_wakeup();
  else
    if (pages_available() < vm_page_threshold())
      pagedaemon_wakeup();
  return allocd_page;
}

after(int rbehind, int rahead): fault(rbehind, rahead)
{
  if ((rahead + rbehind) > (pages_available() - cnt.v_free_reserved))
    pagedaemon_wakeup();
}
```

```
  around(vm_object_t obj, vm_pindex_t pindex): allocate_buffers(obj, pindex)
  {
    vm_page_t m = proceed(obj, pindex);
    if ((m != NULL) && !(m->flags & PG_BUSY)
        && ((m->queue - m->pc) == PQ_CACHE)
        && (pages_available() < vfs_page_threshold()))
      pagedaemon_wakeup();
    return m;
  }

  around(void* count, int where, char* fname, int flag):
    sleep_in_wait(count, where, fname, flag)
  {
    if (where == PVM)
      pagedaemon_wakeup();
    return proceed(count, where, fname, flag);
  }
}
```