# AN ANALYSIS OF SOFTWARE QUALITY AND MAINTAINABILITY METRICS WITH AN APPLICATION TO A LONGITUDINAL STUDY OF THE LINUX KERNEL

By

Lawrence Gray Thomas

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

In

Computer Science

August, 2008

Nashville, Tennessee

Approved:

Dr. Stephen R. Schach

Dr. Larry Dowdy

Dr. Julie Adams

Dr. Richard Alan Peters II

Dr. Ralph M. Butler (MTSU)

To my beloved wife, Peggy

and

To the memory of my mother, Nina Irene Gray, Vanderbilt Class of 1947

# ACKNOWLEDGEMENTS

This work was funded in part by a grant from Microsoft Corporation. I am also grateful to Klocwork Inc., for generously providing their software tools used in this research.

I am especially indebted to Dr. Stephen R. Schach, Associate Professor of Computer Science and Computer Engineering at Vanderbilt University, who has been invaluable in supporting my research, and who as worked actively to provide me with the protected academic time and other necessary resources to pursue my research.

Nobody has been more important to me in the pursuit of this goal than my loving wife, Peggy. I am thankful for her unending encouragement, support, inspiration, and patience.

# TABLE OF CONTENTS

Appendix

# LIST OF TABLES

## LIST OF FIGURES

# CHAPTER I

# INTRODUCTION

## 1.1 What is Quality?

The word “quality” can evoke different responses from different people in different contexts. There is no single, correct measure of “quality.” In a software engineering context, “quality” could mean any, all, or none of the following:

- Adherence to specifications in code design documents;
- Adherence to user requirements (which may or may not match the design documents);
- Number of code faults;
- Efficiency (run time to perform a given task in a given running environment).

Clearly, the word “quality” gives us no hint as to how we might go about measuring it. Most people would say that they “recognize quality when they see it,” but they might have difficulty in defining it. “Quality” is an inherently nebulous term.

Similarly, “size” is nebulous, because there are many ways to measure size. The size of a person could be measured by their height, their weight, their BMI (Body Mass Index, which is computed from *both* height and weight), or a volumetric measurement of the total number of cubic inches their body occupies. In the same manner, the “size” of software can be measured by LOC (Lines of Code), number of modules (files), number of routines (functions and/or methods), number of classes (in the case of object-oriented software), number of variables (or the number of bytes of variables) declared (measures of the "data size" of the software), or any of several other measurements.

There are many such descriptive-yet-nebulous terms that can be applied to software, such as "quality," "maintainability," "reliability," "size," "complexity," "user-friendliness," and a host of others. These terms all refer to a vague aspect that describes an item, but they are all inherently impossible to quantify with a single, correct measurement. Before proceeding, we must establish a semantic foundation on which to build further discussion.

### 1.2 Quality Factors, Quality Metrics, and Validation

The IEEE defines a software *quality factor* (or a *quality attribute*) as "a feature or characteristic that affects an item's quality." As we have already seen, "an item's quality" is a nebulous term. There are many such factors applicable to software that represent a spectrum ranging from "desirable" to "undesirable." As stated above, such factors may include maintainability, quality, complexity, reliability, and user-friendliness, among others. Unfortunately, quality factors are inherently nebulous, and as such, impossible to quantify objectively.

On the other hand, many aspects of software *can* be quantified objectively, by measuring some property of the source code itself (and/or other software artifacts, such as documentation, specifications, etc.). Such a measurement is called a *quality metric*, and is formally defined as “a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which the software possesses a given quality attribute” [IEEE, 1990].

The key distinction to make here is the phrase “can be interpreted as.” *Quality factors* comprise a set of nebulous (non-directly-measurable) properties (quality, maintainability, complexity, etc.), and *quality metrics* comprise a set of concrete, directly-measurable properties (*metrics* such as lines of code (LOC), Cyclomatic Complexity [see Section 2.3.2], and Halstead Volume [Section 2.3.3]). These two sets are non-intersecting.

The only way we can relate a factor and a metric (elements from two distinct and disjoint sets) is to make a plausible argument as to how we interpret the way the behavior of the metric predicts the behavior of the factor. If we make a weak argument, then counter-examples should be relatively easy to produce. Unfortunately, because there is no way to directly measure a factor, there is also no way to *prove* conclusively that any given metric is a good predictor of a given factor.

Some quality metrics *seem*, however, to be good predictors of some quality factors, whereas others do not. Counting the number of lines of code (LOC) might lead us to believe that a longer program is inherently more difficult to maintain than a shorter one, but there are any number of counter-examples of a large program that is universally recognized as being easy to maintain, and vice versa. In such a case, LOC might be a poor predictor of maintainability. On the other hand, a program with a million lines of code is surely more difficult to maintain than a program with 100 lines. Therefore, some generalities will hold, even though the possibility exists to contrive a counterexample. In the case of sequential versions of the same program, the code should be similar enough from version to version that these generalities will still apply.

We therefore define a new term, a *process metric,* as a measurement similar to a quality metric, but one whose input is not a software artifact, but rather some aspect of the software life-cycle process, such as "development cost" or "mean time to make maintenance changes." We can then measure aspects of the development life cycle; indeed, project managers have been doing this for decades. Such metrics are key tools in the successful management of a software project.

A natural consequence of this definition is the relationship between *quality factors*, *quality metrics*, and *process metrics*. Because we cannot measure quality factors directly, we must make measurements on one of the two types of metrics.

Showing a statistically significant correlation between a quality metric and a process metric is known as the *validation*[1] of the quality metric with respect to the process metric. Such an example might be (hypothetically) that McCabe's Cyclomatic Complexity [McCabe, 1976] (a quality metric) has been shown to have a strong correlation with mean time to make maintenance changes (a process metric) [at first glance, it seems plausible that a complex program would take longer to maintain]. In the case of such a relationship

[1] In [Schneidewind, 1992], the author proposes a methodology for validating software metrics, but the author deviates from the IEEE terminology by mixing the definitions of factors and metrics.

between the two metrics, we would say that McCabe’s Cyclomatic Complexity has been *validated* against mean time to make maintenance changes, at least for the source code examined, and for the maintenance changes made and documented within the project being examined.

Any validation is inherently empirical, and must be considered within the context under which it was performed. Showing a correlation in one case does not necessarily mean that the correlation must necessarily hold in all cases, nor should correlation be taken to imply causality. Only after repeated validations under varying contexts can a “generic validation” begin to form.

Now consider a new type of correlation – one between two quality metrics (we will be confining our analysis of quality metrics to those measurable directly from source code and, as such, we will be using the term *code metric* synonymously with *quality metric*, as interpreted with a source-code-only input). Suppose (again, hypothetically) that McCabe’s Cyclomatic Complexity has already been validated against mean time to make maintenance changes. Further suppose that, in the same context, the LOC metric shows a strong correlation to McCabe’s Cyclomatic Complexity metric. The existence of this new correlation means that LOC is now validated with respect to the mean time to make maintenance changes. This is an *indirect validation*.

## 1.3 Coupling

One code metric that has been the subject of considerable study is *coupling* [Selby, Basili, 1991], [Briand, Daly, Porter, Wust, 1998], [Wang, Schach, Heller, 2001], [Schach et al., 2002], [Yu, Schach, Chen, Offutt, 2004].

Coupling is defined as "the degree of interaction between two modules" [Schach, 2007]. There are many different forms of coupling. In an object-oriented environment, for example, two classes might be coupled if one class calls a method or uses a property from another class. In non-object-oriented environments, coupling usually refers to the data interactions between two modules. The higher the interaction, the more the two modules are coupled. One particularly strong form of coupling, which we examine in detail in this dissertation, is *common coupling*.

Common coupling exists when two modules share a global variable (hereafter, *GV*). The term "common coupling" is a vestige of FORTRAN, in which variables were made global by placing them into a block of memory visible from multiple functions by using the keyword COMMON. Because multiple functions could reference the same block of COMMON variables, and the activity of one function, by potentially modifying the contents of the COMMON variables, could influence the operation of other functions sharing the common variables, such functions were said to be common coupled.

In C, variables are made global by declaring them outside of all functions (i.e., at the module level), and by using the `extern` keyword on the declaration, as in `"extern int gv;"` The presence of the module-level declaration and the `extern` keyword are necessary and sufficient to create an instance of a GV, but they are not sufficient to create an instance of common coupling.  For common coupling to exist between two modules, both must reference the GV in the executable code inside a function.

Suppose the following:

- modules A and B both declare GV x
- x does not appear in module B beyond its declaration (i.e., it is declared but not used)
- x is used (somewhere) in the executable code of module A

In this case, we would say that variable x is *visible* in both modules A and B, but *accessed* (or *used*) in only module B.  The fact that x is declared but never used in B is, at best, poor programming practice, and at worst, a maintenance problem waiting to happen with some subsequent modification, but we would not be able to say that modules A and B are common coupled  [Schach, 2007, pp.187-189].

Coupling has been validated against several other process metrics, including:

- Stability (i.e., the ease with which a software item can evolve while preserving its design) [Grosser, Sahraoiu, Valchev, 2002];
- Faults in source code [Selby, Basili, 1991], [Basili, Briand, Melo, 1996],
- The impact of changes (maintenance) [Briand, Wust, Lounis, 1999];
- Maintenance cost [Ince, 1988];
- Development time and error rate [Ferneley, 2000]; and

- Maintenance effort [Epping, Lott, 1994].

As a consequence of this extensive validation, common coupling is the "gold standard" against which other process metrics can be indirectly validated.

## 1.4 Previous Validation Studies

Previous examples of metric validation have focused on either:

1) Gathering the metric at multiple points within a single application. For example, in a given release of a particular product, it was noted that modules with high (>10) values of McCabe's Cyclomatic Complexity (MCC) were associated with decreased reliability [McCabe, 1976]; or
2) Gathering metrics within multiple (often dissimilar) applications simultaneously:
    - [MacCormack, 2006] (Linux, Apache, Mozilla);
    - [Oman, Hagemeister, 1994] (8 unidentified suites of Hewlett-Packard applications);
    - [Chou et al., 2001] (Linux, OpenBSD);
    - [Nakakoji et al., 2002] (GNUWingnut, Linux Support, SRA-PostgreSQL, Jun);
    - [Mockus, Fielding, Herbsleb, 2002] (Apache and Mozilla); and
    - [Paulson, Succi, Eberlein, 2004] (GCC, Apache, Linux, 3 unidentified closed-source systems).

Both of the above approaches seem to fail to take into account two significant factors:

1) The cross-application differences (such as application domain, choice of programming language, etc.) may be so great as to make comparisons of a given metric between any two arbitrary applications meaningless; and
2) The trend over time of the value of a given metric for a (single) given application may prove to be significant, even if single measurements of a given metric may not be.

Some metrics that are simple to compute and understand are of questionable utility. Despite the fact that these metrics typically are computed by nearly every CASE tool that provides source-code metrics, there is considerable controversy regarding the validity of these metrics. We analyzed several of these metrics in consecutive versions of three series of the Linux kernel, and determined the degree of correlation between these metrics and common coupling. To the extent that one or more of these other metrics correlates strongly with common coupling, then these other metrics are said to be *indirectly validated* as ways to measure the metrics with respect to which common coupling has been previously validated.

### 1.5 Research Objectives

In [Schach et al., 2002], the authors examined common coupling within the context of a portion of the Linux kernel, for as many released versions as were in existence at the time. Newer CASE tools exist to programmatically obtain the GV information on a much broader code base. Among the goals of this dissertation, with respect to this prior work, were to apply CASE tools to:

1) Verify the conclusions of Schach et al.;
2) Extend the previous analysis to Linux versions released subsequent to the prior work, as well as to larger subsets of the kernel's code base; and
3) Repeat the process using release date, rather then sequential release number, as the independent variable for determining the rates of growth in both size and the number of instances of common coupling in the Linux kernel.

In addition to the above extension of the previous work on the rate of growth of common coupling in the Linux kernel, this dissertation leverages the capabilities of the CASE tools to gather more metrics on the Linux kernel than just common coupling, and tests the correlations in multiple sequential versions of the Linux kernel between common coupling and the following code metrics:

1) LOC;
2) McCabe's Cyclomatic Complexity;
3) Oman's Maintainability Index (MI);
4) Halstead Volume and Effort; and
5) A "risk" metric unique to the CASE tool being used.

Specifically, we wanted to see if we could draw correlations between some of these older metrics in sequential versions of the Linux kernel. Even though many researchers have claimed that these controversial metrics are of marginal value *in general*, or perhaps of no value at all, we wanted to determine if they could be validated against common coupling on sequential versions of the *same product*.

This analysis was performed on the Linux kernel for several reasons:

- It is a non-trivial body of code with which to work;
- It has been the object of study for numerous previous analyses;
- The source code for every version ever released is readily available; and
- The Linux kernel runs on a large number of systems, so any conclusions we may draw are broadly applicable in the field.

There is no reason to believe that such an analysis could not also be performed on closed-source software, or on some other open-source software, such as MySQL, Apache, or Mozilla. As mentioned above, however, any validation must be interpreted within the context of its running environment. The relationships that exist within successive versions of the Linux kernel may or may not also hold for individual or multiple sequential versions of other applications.

## 1.6 Problem Statement

In a nutshell, the problems this work addresses are to:

1) Re-calculate of the results of [Schach et al., 2002], using release date, rather than version number, as the independent variable, but using the original study's data.
2) An extension of the work of [Schach et al., 2002], considering a larger code base, more versions of the kernel (primarily those released since the previous study), and using preprocessed code.

3) Perform longitudinal studies of three series of the Linux kernel (2.00.x, 2.02.x, and 2.04.x) in an attempt to validate any or all of the following metrics with respect to Common Coupling: Oman's Maintainability Index metric, McCabe's Cyclomatic Complexity, and/or Halstead's Volume and Effort metrics.

## 1.7 Contributions of This Work

The contributions of this work are:

1. The authors of [Schach et al., 2002] used the Linux kernel version number as their independent variable, and found that the common coupling in the Linux kernel was growing exponentially. When we re-evaluated Schach et al.'s data with respect to release date, rather than version number, we obtained a linear growth rate. Repeating this study with our standardized configuration, when applied to the preprocessed source code for three separate kernel series, we again found that the growth of common coupling in the Linux kernel is linear with respect to time (see Chapter 8).
2. Previous studies have concentrated on examination of the source code in only the `/linux/` subdirectory. That approach captures only a tiny portion of the source code base. We have examined the entire source code base and constructed complete, fully-configured kernels, actually capable of running on real hardware (see Chapter 5).

3. Because the Linux kernel is so highly configurable, the set of configuration options has a profound impact on the code base selected for compilation. We have developed a framework for selecting a consistent configuration across multiple versions of the Linux kernel, which makes longitudinal studies more readily comparable (see

Chapter 5). This method can be extended to other operating systems, and to software product lines that are hardware-dependent.

4. A number of "classical" metrics have been argued to be of questionable value as general predictors of software quality and maintainability. We performed a validation study using a longitudinal analysis of the Linux kernel to determine if metrics of questionable general validity might have increased merit when applied to sequential versions of a single product. We concluded that only LOC (lines of code) was useful as a predictor of common coupling, a metric that has previously been validated with respect to various quality and maintainability attributes. McCabe's Cyclomatic Complexity, Halstead's Volume and Effort, and Oman's Maintainability Index all failed to consistently have a statistically significant correlation with common coupling, the "gold standard" (see Chapter 9).

5. Although we used many GPL (GNU Public License[2]) open-source programs and one commercial closed-source programs in our analysis, we also created a suite of programs to act as "glue" between the other tools, and to provide a GUI-based interface to the database to facilitate selection, graphing, and computation of statistics on all metrics in the database (see Appendix A).

6. Because many of the older kernels had code that depends on compiler features that have since been deprecated, the older kernels cannot be built using newer versions of

[2] An overview of the GPL can be found at http://en.wikipedia.org/wiki/GPL.

gcc. We have identified a set of tools that allows the kernels to be built and the changes that must be made to the source code to achieve this (see Appendix A).

7. As discussed in Section 2.3.1, inter-file Common Coupling is quadratic in subsets of the number of files sharing each of the respective global variables used in a program. Such instances are typically counted only once per file (i.e., multiple occurrences of the same Global Variable in the same file are counted as only once instance). We discovered a direct linear correlation between the total number of instances of global variables throughout the entire source code base and inter-file Common Coupling count.

## 1.8 Organization of the Remainder of the Dissertation

The remainder of this dissertation is organized as follows:

- In Chapter 2, we examine and discuss the relevant literature that forms the foundation for this research, including sections on each of the respective software metrics to be examined.
- Chapter 3 discusses the Linux kernel source code and the third-party tools we used in this work.
- Chapter 4 discusses in detail our motivations for revisiting the work of [Schach et al. 2002].
- Chapter 5 discusses the process we developed for selecting a configuration for empirical research on the Linux kernel.

- Chapter 6 presents a more detailed categorization of global variables, and the CASE tool we developed to aid in the classification.
- Chapter 8 presents our results from the reexamination of the work in [Schach et al. 2002].
- Chapter 9 presents and discusses the results of our validation study of several metrics against common coupling.
- Chapter 10 Presents a summary of our conclusions, and discusses potential future work.
- Appendix A presents our workflow for obtaining our source code metrics, examines some of the identified challenges that must be addressed to complete this research, and introduces the tools we created to facilitate the collection, management, and analysis of our metrics

# CHAPTER II

# RELATED WORK

## 2.1 Open-Source Software and the Origins of Linux

The kernel for the Linux Operating system [`www.kernel.org`] is perhaps the crown jewel of the open-source software movement. There are several attributes of the Linux kernel project that make it an attractive topic for investigation. Among them are:

- It has been developed by a relatively large number of contributors;
- It has been developed over a relatively long time;
- The kernel has been developed with a sense of urgency and intensity (it has not languished for long periods of time);
- The installed base of Linux (particularly for Web servers) makes it a significant player in the operating system market;
- Because a number of distributions have been available at little or no up-front cost, it has always been economically attractive; and
- The entire source code for all versions since Version 1.0.0 (March 1994) is freely available on the Internet.

The modern Free/Open-Source Software movement (F/OSS) was launched by Richard Stallman with the 1983 announcement of the GNU project itself [gnu announcement, 2007] and the publication in 1985 of the GNU Manifesto [gnu manifesto, 2007] and the formation of the Free Software Foundation (`www.fsf.org`).

In the early days of computing, when computers were not so much a general business tool, but more for research and experimentation, programmers would freely share their software with colleagues at other locations. In the 1960s, however, when software became as much a marketable asset as hardware, more and more software became a proprietary resource, providing its owner with some degree of competitive advantage, and the free sharing of non-trivial code naturally ground to a halt.

With the development of the UNIX operating system by Bell Labs in the early 1970s and the licensing agreement that made it available to educational institutions at very little cost, UNIX quickly became a popular operating system at the university level. Much software for the UNIX platform was freely exchanged, as were modifications and enhancements to UNIX itself. In the early 1980s, however, when the licensing terms of UNIX became less attractive to educational institutions, Stallman longed for a UNIX clone that he could distribute freely ["Free" in this sense comes from *libre*, as in "freedom of speech," as opposed to *gratis*, as in "without cost," or "free beer"]. Thus was born the GNU ("Gnu's not UNIX") project [Wikipedia[*] Unix, 2007], [The Open Group, 2003].

[*] We recognize that the Wikipedia is not a refereed source, and as such it may be considered somewhat less reliable than other such sources. However, some of our background material, which forms more of a historical frame of reference than it does a research foundation, has been taken from en.wikipedia.org. Such references are provided in an attempt to chronicle the history of some events, and separate actual chronology from folklore, to the extent that such anecdotal records are available.
Moreover, the date cited in references to online sources in general is the date such a web page was last updated, if available; if the date of last update is not available, the most recent copyright date shown will be cited; in the absence of either of the above two dates, the date the page was referenced will be cited.

It was not too difficult for the FSF to develop most of its GNU operating system (the development environment, tools, libraries, and documentation), except for the O/S kernel itself. Fortunately for the FSF, a Finnish graduate student named Linus Torvalds was working on an O/S kernel based on the Minix operating system, which had been developed by Andrew Tanenbaum as an instructional tool for courses in Operating System design concepts. Torvalds posted this announcement to the `comp.os.minix` newsgroup:

> "I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since April, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).
>
> I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months [...] Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT portable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-(.
>
> [...] It's mostly in C, but most people wouldn't call what I write C. It uses every conceivable feature of the 386 I could find, as it was also a project to teach me about the 386. As already mentioned, it uses a MMU, for both paging (not to disk yet) and segmentation. It's the segmentation that makes it REALLY 386 dependent (every task has a 64Mb segment for code & data - max 64 tasks in 4Gb. Anybody who needs more than 64Mb/task - tough cookies). [...] Some of my "C"-files (specifically mm.c) are almost as much assembler as C. [...] Unlike minix, I also happen to LIKE interrupts, so interrupts are handled without trying to hide the reason behind them." [Wikipedia Linux Kernel, 2007]

Following this announcement, a number of people contributed to the source code, and version 0.01 of the Linux kernel was released in September 1991. It was not until March 14, 1994, that Linux 1.0.0 was released.

The specific development path for Linux is not unlike that of other F/OSS projects in that such projects are typically initiated by an individual (often with little more than a prototype), released into the F/OSS community, and then either an individual or a small group of individuals manages the incorporation of enhancements contributed from the community into the released versions.

A veritable plethora of studies have been conducted on the F/OSS process itself. Perhaps the first, seminal paper on the subject is "The Cathedral and the Bazaar," by Eric S. Raymond [Raymond, 2000]. Because every project is different, meeting differing needs of differing target users, there is not a one-size-fits-all description of how F/OSS projects work. However, it is fair to say that most F/OSS projects (the Linux kernel included) exist in a state of "perpetual maintenance."

## 2.2 Software Maintenance

Consider the "classical" software development cycle [Schach, 2007]. In the initial phase, *requirements* of what the software is supposed to do are elicited from the users (requirements gathering). The requirements phase then gives way to an *analysis* phase, in which the requirements, which are typically expressed in language appropriate to the user's milieu, are converted into *specifications* expressed in a language more appropriate for the programmers who will have to ultimately implement those requirements. Following this analysis phase, a *design* phase then determines the best way to actually structure the required pieces of code and data structures. The *implementation* phase is when the programming actually takes place to produce working software. Once the software is delivered to the customer (or is released to the user base), it enters the *maintenance* phase, and it stays in the maintenance phase until the software is *retired*, or removed from service.

Maintenance can be broadly broken down into two categories: corrective maintenance and enhancements. Corrective maintenance is typically considered “bug-fixing”; that is, correcting those aspects of the software that do not perform as designed or intended. Enhancements typically encompass two types of activities – (1) perfective maintenance, which is the incorporation of changes to add new functionality to the program, or to improve the existing functionality (such as improved efficiency, capacity, response times, etc.); and (2) adaptive maintenance, which includes changes made to software to support changes in the software’s operating environment. The “environment” could include hardware/software platforms, but could also include less concrete considerations such as ever-changing regulatory requirements, evolving standards, and support for new protocols and other aspects that were either unforeseen at the project’s inception, or were not chosen for incorporation in previous versions.

Given these definitions of maintenance activities, and the way the Linux kernel was developed (Linus released an early version, and then programmers contributed various corrective, adaptive, and perfective changes), it is fair to say that the Linux kernel, ever since version 0.01, has been in a state of perpetual maintenance of one form or another.

The Linux kernel, up through versions 2.5.x, has been through alternating cycles of "development" and "production" kernels. The version numbering system that has been in place uses the first number as the "major" version, and there have only been two, "1" and "2." Within each major version, the "development" kernels have had odd numbers in the second position (1.1, 1.3, 2.1, etc.), and the "production" kernels have had even numbers in the second position (1.0, 1.2, 2.0, 2.2., etc). The third position in the numbering scheme is simply an ordinal number indicating sequential releases of that kernel (1.0.1 was followed by 1.0.2, 1.0.3, etc.).

In an attempt to avoid confusion between the three portions of a kernel version number, we have adopted the convention of using a two-digit (zero-padded) number to indicate the middle portion of a kernel version number, and a three-digit (zero-padded) number for the last portion of the version number. Therefore, we will refer to kernel version numbers like 2.02.000 and 2.04.031, rather than 2.2.0 and 2.4.31. When we refer to an entire series of kernels, we will represent the three-digit last portion using "x" as the wildcard character. For example, a reference to "2.02.x" is synonymous with "2.02.000 – 2.02.026."

The "development" kernels have primarily seen perfective and adaptive maintenance. Once the Linux kernel's project managers (Linus himself, or an individual or two designated as the gatekeeper of the code) have deemed the "development" kernels to be of sufficient quality and possessing an acceptable set of features, the development cycle ends, and the code is re-christened as “production” code, at which point the primary activity performed on the code is corrective maintenance.

In version 2.06.x, however, the “development” and “production” cycles have been merged, and the 2.06.x kernels have seen corrective, adaptive, and perfective maintenance [Wikipedia Kernel Versions, 2007].

Therefore, with “perpetual maintenance” as the Linux kernel’s primary development model, it would seem intuitive that if the source code is difficult to maintain, that the ultimate long-term success of the product would be jeopardized. How does one measure “difficult to maintain”? As we have seen in Chapter 1, there is no objective, quantifiable measure of “maintainability” for source code, because “maintainability,” like “quality,” is a quality factor, and not a metric. The only way to measure maintainability is to measure some *other* objective, quantifiable metric and determine whether that metric correlates strongly to the relative ease or difficulty in maintaining the code, or the quality of the code.

### 2.3 Metrics Related to Quality and Maintainability

Some simple source-code metrics would seem to intuitively be strongly correlated to aspects of quality and maintainability, such as the following:

- Lines of code (LOC) – as a program becomes longer, it may also become more complicated. More complicated code is inherently more difficult to maintain (at least without introducing regression faults). Consequently, LOC could be used as a measure of maintainability. However, it is generally considered a poor measure, because longer programs are not *necessarily* more complex than shorter ones.

- Code with few (or no) comments would be more difficult to maintain. Programmers new to the code base (or those returning to a section of code after a long time) would have trouble knowing exactly what the code does (and how it works) with no (or poor) comments. Consequently, changes made to the code could either be ineffective, or inadvertently cause other problems.
- Maximum (or average) depth of control statements – it seems likely that a program with deeply nested `if-then-else` and/or `switch` statements would incorporate a complicated, convoluted hierarchy of potential cases. Maintaining and testing such code could cause some subcase to be missed, or have some other undesirable side effect. Either way, code with a high level of control statement nesting would seem to be a candidate for low maintainability.
- Code granularity (number of LOC per function) – Long, monolithic blocks of code do not capitalize on the benefits of a more modular design (information hiding, clarity of flow, and delegation of specific supporting activities to other lower-level functions with improved fault isolation). On the other hand, artificially pushing the granularity (making functions shorter just for the sake of lowering the metric) can lead to chaos, with functions calling other functions unnecessarily.
- Excessive use of global variables (GVs). We discuss global variables in more detail in the next subsection.

### 2.3.1 Global Variables and Common Coupling

Unless code is to be written as a monolithic block, there will have to be some separation of functionality into different modules and/or functions. Once there are multiple functions, the next question becomes one of which data in one function needs to be available in some *other* function. There are two ways to make such data available: (1) explicitly pass the value to the other function (and back, if needed); or (2) make the data globally available.

The former approach can cause performance problems:

- In the case of frequently used variables, there can be a significant overhead penalty for pushing the same variables onto the call stack too frequently.
- The use of long parameter lists can lead to faults. Ensuring the use of the correct *number* of parameters can be enforced by the compiler, but getting them in the right *order* cannot be guaranteed programmatically. For example, if one function calls another with a parameter list of `(a, b, c, d, e, f, g)`, all of which happen to be of type `int`, then the compiler will not catch the error if the call is coded with the parameter list `(a, b, c, e, d, f, g)`. The longer the parameter lists are, the more likely such parameter-ordering faults will be.
- In the case of a large number of functions calling each other, pushing large numbers of parameter variables onto the call stack (not necessarily the same variables over and over) can cause not only performance issues, but can unnecessarily waste limited stack space. In some environments, such as an operating system, the stack is a relatively scarce resource, exacerbating this problem.

On the other hand, the latter approach can also lead to problems:

- The over-use of GVs can lead to sloppy programming practices and negate the benefits of modularizing the code.
- More importantly, without control over the visibility and use of a GV, changes made to one function can impact another function's operation by making an unexpected change to the GV; this is a major problem caused by common coupling (CC).
- In a typical application, such as a word processor or a spreadsheet, use of GVs is inherently somewhat safer than it is in an operating system. Most code is written under a programming model that makes it appear to the program that it is the only process running on the computer. Of course, code is constantly being interrupted for everything from simple clock updates to DMA (Direct Memory Access) I/O completion notifications. If a spreadsheet application is executing, and is interrupted by the operating system to service some system-level event, when control is returned to the spreadsheet, it will not realize that it has been interrupted, and the state of the spreadsheet application will be just as it was prior to the interrupt. Now consider the case of operating system code. Suppose code in the memory management subsystem is executing and a timer interrupt occurs (we must further suppose, of course, that interrupts are enabled at this point). The timer interrupt is serviced, and control returns to the memory manager. Just as in the spreadsheet example, the memory manager code does not realize that it has been interrupted, but what happens if some global variable that the memory manager uses was changed by the timer code? In the spreadsheet example, the global variables in the spreadsheet are independent of the global variables in the operating system. Therefore, an operating system interrupt

cannot unexpectedly modify any of the spreadsheet's global variables. However, if the operating system is preemptable, then it can interrupt *itself*, and the value in a global variable potentially becomes much more volatile. This makes the issue of common coupling much more serious in operating system code, because variables may vary unexpectedly.

- From a maintenance standpoint, an even more insidious effect can occur – Clandestine Common Coupling (CCC) [Schach et al., 2003]. Suppose files A, B, and C are all common-coupled (in that they share access to some GV). There are three common-coupling relationships present (A–B, A–C, and B–C). Now suppose a programmer adds code to file D in the system, and that the code in file D also refers to the same global variable. The introduction of this one new file raises the number of instances of common coupling from three to six (A–B, A–C, A–D, B–C, B–D, and C–D). The inclusion of file D, because it might make unexpected changes to the GV it uses, can now cause problems in file(s) A, B, and/or C, even though none of those files has been changed in any way. This effect is known as clandestine common coupling (CCC), and can be disastrous in terms of quality, reliability, stability, and maintainability.

If we represent the files involved in common coupling as nodes in a graph, and the instances of coupling as edges between the nodes, then the number of instances of common coupling becomes the number of edges in the fully connected graph between these nodes as shown in Figure 2.1:

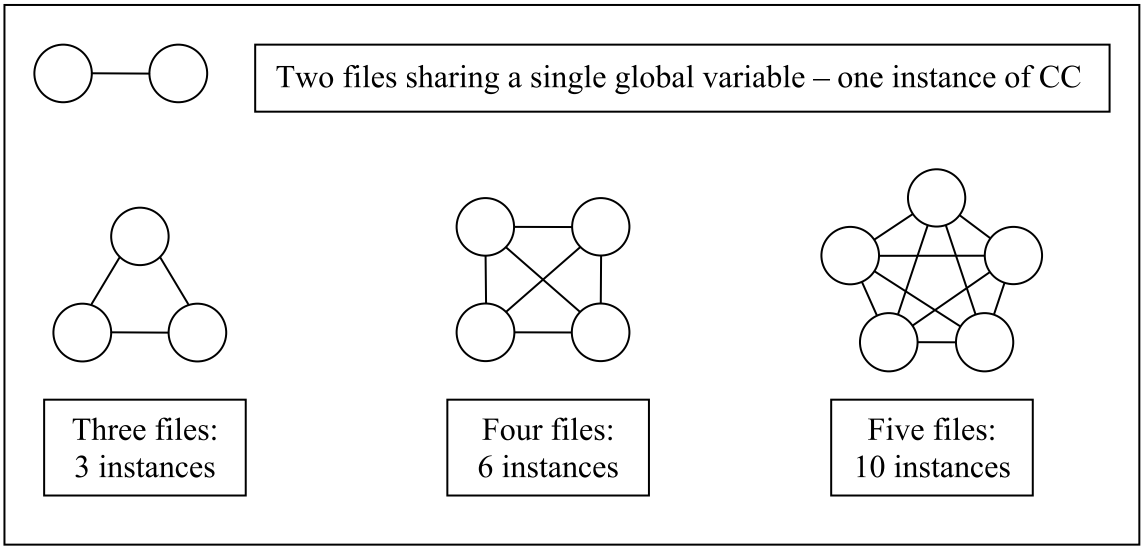


**Figure 2.1 – Instances of Common Coupling for 2, 3, 4, and 5 files**

In general, for *N* files sharing one variable, there will be $\binom{N}{2} = \frac{(N^2 - N)}{2}$ instances of common coupling.

How then, does a programming team strike the appropriate balance between passing too many parameters through the call stack (avoiding all CC issues, but incurring performance penalties) and using too many GVs, thereby introducing regression faults through CCC? There is no simple answer to this question. However, given that common coupling can lead to such problems, if the rate of common coupling is, over successive versions of the same code base, growing faster than the overall size of the code base, then it could be argued that the code is becoming more and more unstable.

Comparing the CC rates of fundamentally different code bases (such as the Linux kernel and the Apache web server) is meaningless, because there are so many other factors that differentiate the two products. However, measuring the same metric (CC rate) over successive versions of the same product can yield useful insights as to the trends over time, and make some predictions possible.

Suppose that the CC rate is growing exponentially (in successive versions of the same product), whereas the LOC is growing only linearly. This would indicate that a source of potential problems is growing unchecked. As such, the product must either be redesigned to reduce the number of instances of CC relationships or it must eventually collapse under the weight of the inevitable regression faults introduced by CC. Because of the rate of growth in common coupling that Schach et al. observed in the Linux kernel, their conclusion was that the Linux kernel would become increasingly difficult to maintain unless it was "completely restructured with a bare minimum of common coupling."

In [Schach et al., 2002], the authors analyzed common coupling within the context of the 19 files comprising the `/kernel/` subdirectory. We remark that this is only one part of the Linux kernel, but much of the core functionality of any operating system is contained in these files (the following is a partial list from 1.00.000):

- `dma.c` – DMA (direct memory access) manager
- `exit.c` – Handles destruction of processes
- `fork.c` – Handles creation of processes
- `ioport.c` – Input/output port manager
- `irq.c` – Interrupt request manager

- `sched.c` – Task scheduler
- `signal.c` – Handles interprocess communication via signals
- `time.c` – Time-based operations

The files examined in that study did *not* include any of the code in other subsystems, including (again, a partial list from 1.00.000):

- `boot/` – files required to load and initialize the kernel itself at boot time
- `drivers/` – various device drivers
- `fs/` – file systems
- `mm/` – memory management
- `net/` – network support

These other subsystems are no less important, but they are either not necessarily required for all builds, or they may be used in vastly different ways. Of course, the boot code is required, and *some* file system must be present, but Linux (particularly in newer versions) supports multiple file systems, even though most users will use only one or perhaps two file systems. Similarly, most (but not all) installations will use some form of network access, but the particular protocols and services to be included in the kernel can vary widely.

Perhaps the most variable part of any kernel's build configuration is the specific list of devices to be supported, and indeed, the `drivers/` subdirectory contains more code than any other. In 1.0.0, because Linus Torvalds was considering only the i386 architecture, and AT-style (IDE) hard drives, the underlying architecture support code was simple, and came basically in one flavor – i386. Linux has since been ported to run on at least a dozen architectures, with a number of chipsets, I/O controllers, and an untold number of video, sound, SCSI, RAID, game controller, and network cards. This makes considering the `drivers/` subdirectory somewhat misleading, as a great percentage of the code it contains will never be used in most kernel builds.

Common coupling is not the only metric that has been examined and correlated with quality issues. We now examine some of these other metrics.

### 2.3.2 McCabe's Cyclomatic Complexity Metrics

McCabe's Cyclomatic Complexity [McCabe, 1976] is an old metric (first published in 1976) that is still often used to describe the degree to which code can be considered "complex." Intuitively, complex code would seem to be more likely to contain faults, be more difficult to test, debug, and maintain. Overall, "high complexity" would seem to be an undesirable attribute.

McCabe presents a concept derived from graph theory as a measure of software complexity – the cyclomatic number of the control graph of the program. This value is the number of distinct potential paths through the code in a program.

McCabe shows that this value is equivalent to the number of predicates (conditional branches) in the program, plus 1. In a program with no conditional statements (`if-then-else`; `for`, `while`, and `do` loops; and `case` statements), the cyclomatic complexity will be one. The inclusion of a single `if` statement (with or without an `else` clause) creates a second potential path through the program. This makes determining the cyclomatic complexity rather simple for the programmer (or a CASE tool), in that the number of predicates is the number of potential conditional branches.

In a production environment with which McCabe was associated, the McCabe Cyclomatic Complexity (MCC) was applied to the code development process, and an arbitrary value of 10 was chosen as the maximum acceptable value for MCC. When code was developed with an MCC value greater than 10, the code had to be either re-written or modularized in order to get the MCC value back below 10.

In anecdotal form, McCabe relates that in a large real-time graphics system, where reliability was considered critical, the project members' ranking of the most troublesome routines correlated strongly with high values of MCC.

McCabe presented the notion of cyclomatic complexity, but did so in two different ways; one was to count the number of predicates (decision-making branches) and add one. The other way adds "phantom predicates" in the case of compound conditions (see disadvantage #6 below). This second method always raises the cyclomatic complexity in the presence of a compound condition.

Myers [Myers, 1977] proposed that, in order to properly consider MCC, it should be evaluated as lying in a range whose *lower* bound is the number of predicates plus 1, and whose *upper* bound is the number of individual conditions plus 1.

There are numerous references in literature to the "Extended Cyclomatic Complexity," which seems to be universally understood to be the number of conditions plus one (Myers' upper bound), but McCabe never coined such a term. According to some, Myers did so with his 1977 paper, but the only "extension" Myers contributed was to the notion that there are shortcomings to using either the lower or upper bound and, as such, the complexity should be expressed as a range bounded by *both* values. Myers never used the specific term "extended cyclomatic complexity."

MCC has the following advantages:

1) It is simple to calculate directly from the source code. This property makes it practical for programmers to check their own code directly.
2) It has a certain degree of intuitive validity – more potential paths through the code (higher MCC values) means more intricate logic, with more potential sub-cases, which implies more test cases are required to test all possible paths through the code.
3) Beyond this intuitive validity, MCC has been validated with respect to the likelihood of the existence of a fault in code [Walsh, 1979].

MCC has the following disadvantages:

1) Its value is determined entirely by the complexity in control logic of the application. There is no inclusion in this metric for the data complexity of an application [Schach, 2007].
2) MCC was developed at the function level, and does not have any provision for evaluating the complexity of the procedural interconnections *between* functional units of code. This was addressed in [McCabe, 1989], but only in the context of a completely different topic, related to integration testing and, as such, lies beyond the scope of this project.
3) Any threshold delineating the break between acceptable and unacceptable values is purely arbitrary and subjective. McCabe uses a single threshold of 10, indicating that "studies have shown that the risk of errors jumps for functions with a cyclomatic complexity over 10, so there's a validated threshold for reliability screening" [McCabe, Watson, 1994], although the SEI (Software Engineering Institute) divides the MCC scale into four ranges [SEI Cyclomatic, 2000], as shown in Table 2.1:

**Table 2.1 – MCC threshold values suggested by the SEI**

| MCC Value | Complexity and Risk |
|---|---|
| 1 – 10 | Simple, low risk |
| 11 – 20 | Moderately complex, moderate risk |
| 21 – 50 | Complex, high risk |
| 50+ | Untestable, very high risk |

In [Watson, McCabe, 1996] the authors indicate that the precise threshold is up to the management team, and that "an organization can pick a complexity limit greater than 10, but only if it is sure it knows what it is doing, and is willing to devote the additional testing effort required by more complex modules."

4) It has been argued that as a program gets longer, it is reasonable to assume that there will be more instances of *all* statement types (assignment, `if`-statements, loops, etc.). To the extent that MCC increases simply because LOC has increased, MCC and LOC may be correlated. The fact that MCC does not take LOC into account at all has led to the creation of a new complexity-related metric, sometimes known as "Decision Density" or "Cyclomatic Density," which is the MCC divided by the LOC. Code with higher decision density has more predicates per LOC than code with a lower decision density and, as such, may be a candidate for increased testing.

5) Another problem with MCC is that it has not "scaled well" in the three decades since its introduction, although it is still widely used. By "scaling," we refer to MCC's inability to adapt to software projects in the new millennium. The old adage that "programs will expand to fill all available memory" is alive and well. "Feature bloat" and "eye candy" have become de rigueur in everything from productivity suites to operating systems. Faster processors, larger memory capacities, the proliferation of personal computers, and increasingly more capable machines have created the demand for, and the viability of, increasingly complex software systems.

McCabe's original paper was written during the time when FORTRAN was the language of choice for most projects, and indeed, the examples in McCabe's paper were in FORTRAN. McCabe [McCabe, 1976] gives the example of "a main program M and two called subroutines A and B having a control structure shown [in Figure 2.2]":

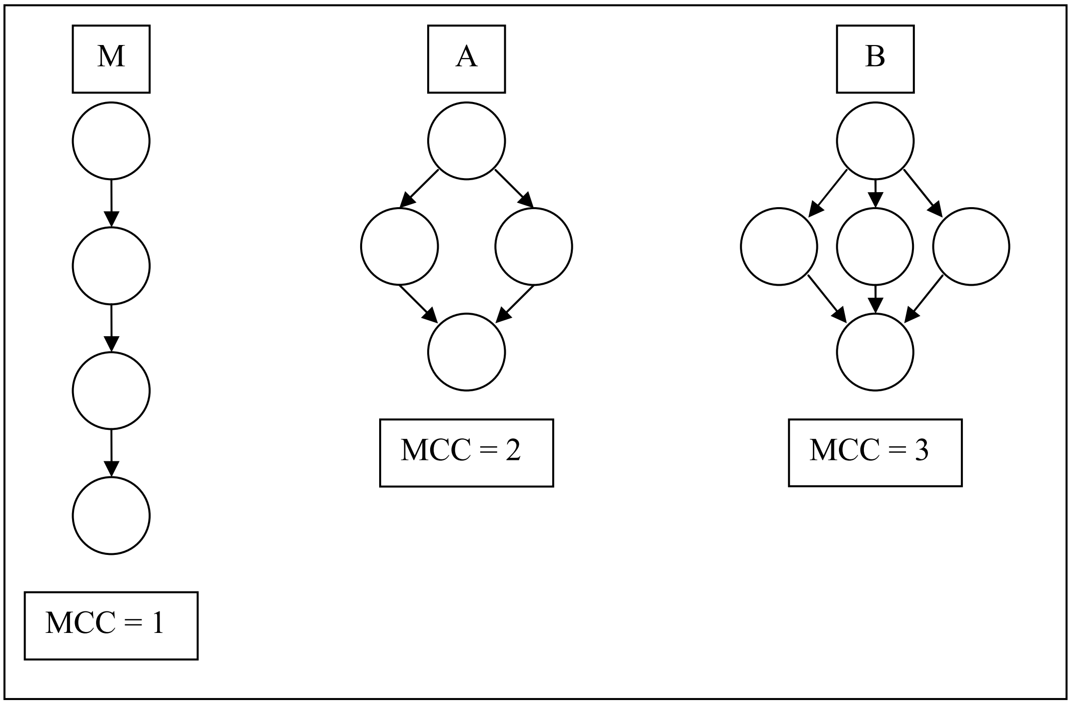


**Figure 2.2 – MCC values of Main program M and two subroutines A and B**

McCabe then demonstrates that the computation of the cyclomatic complexity of M ∪ A ∪ B is the same as the sum of the complexity of each of the components (6, in this example), and states that "this method ... can be used to calculate the complexity of a collection of programs, particularly a hierarchical nest of subroutines as shown above ... In general, the complexity of a collection *C* of control graphs with *k* connected components is equal to the summation of their complexities."

Consider now a project such as the Linux kernel. In version 1.00.000, there were a total of 282 source files (`.c`) files and 205 header (`.h`) files. In version 2.06.000, there were over 6,000 `.c` and 6,200 `.h` files. It must be noted, however, that neither all of the `.c` nor all of the `.h` files are actually used in any given build of the kernel, and some files (particularly the `.h` files) are used more than once. Clearly, it is not simple to determine the control graph from a project so modularized. As such, the closest estimate we can make then, at the project level, is to simply sum MCC from *all* functions in *all* modules. If a given function is called from more than one location (that is, it does not have unique entry and/or exit points), then MCC as described above will under-estimate the complexity of the control graph. As McCabe points out, the statement `if (A and B) then`… is logically equivalent to `if A then if B then`…. Using a logical conjunction in a predicate essentially creates a new predicate. As previously mentioned, McCabe does not give this scenario a name, but it has come to be known as the "*extended cyclomatic complexity.*"

In the case of a production environment, where programmers' performance is evaluated using the extended cyclomatic complexity (with lower values being considered more desirable), it would be simple for a programmer to circumvent this effect and transform the extended cyclomatic complexity back to the simple cyclomatic complexity by introducing a new logical variable outside of the predicate:

```
C = A and B
if (C) then…
```

As such, the conjunction will not be visible in the predicate, and the predicate will only contribute 1 to the complexity total.

Another less-obvious issue with using the extended cyclomatic complexity is that of short-circuiting the evaluation of logical expressions and side effects. Consider the run-time evaluation of the following expression:

```
if (a and b) then …
```

In the interest of performance, the compiler *may*, as McCabe says, treat this statement as:

```
if (a) then
   if (b) then…
```

One implication of this form of evaluation is that, if `a` is false, the program never checks the value of `b;` because once part of a conjunction is known to be false, then the entire expression must be false. Accordingly, there is no sense in wasting the CPU time to evaluate the other parts of the conjunction. A similar argument applies for a *disjunction* (`or` expression) in which one part is already known to be true – the entire disjunction must also be true.

In the case of simple variables, this is not a problem, because the evaluation of a simple variable has no side effects. Consider, however, what happens if part of a conjunction is the invocation of a function that returns a logical value. Suppose we have a function `read_char_from_stdin` which returns `true` if it successfully reads a value from `stdin`. In the case of an expression such as `if (a && read_char_from_stdin() )`, if `a` is false, then we will never invoke `read_char_from_stdin`. This may or may not be what the programmer intended, and the state of the input stream may not be as expected.

The conjunction does not necessarily have to involve a function call for side effects to be an issue. Some operators, particularly in C, have their own side effects. The following line of code is subject to the same effect:

```
 if (a && b++ && --c) /* do something */
```

If the language is short-circuiting the evaluation of the logical expression and `a` is false, then `b` will not be incremented and `c` will not be decremented.

The short-circuiting of logical expression evaluation varies from language to language, and may be selectable at the compiler level (short-circuiting can be considered a performance optimization). In cases where the language does not support short-circuiting the evaluation of logical expressions, then the compiler would treat the following expression:

```
if (a and b) then...
```

as:

```
Evaluate a (which might entail a function invocation) and store its value.
Evaluate b (which might entail a function invocation) and store its value.
Create temporary_variable, and initialize its value to (a and b)
if (temporary_variable) then…
```

This version of the execution order obviates the extra (implied) predicate, and thus the simple and extended cyclomatic complexity values are the same, and no side effects are missed, because the evaluation of `a` and the evaluation of `b` is guaranteed to catch invocation-related side effects.

In yet other cases, the use of short-circuiting can be precisely what the programmer intended. Suppose that we need to do something with a pointer, but only if the pointer is not `NULL`. We can take advantage of the short-circuiting of C and perform the `NULL` test along with the function invocation, knowing that the function will *not* be invoked in the event that the pointer is `NULL`:

```
if (p != NULL && free(p)) ...
```

We do not want to free memory that has not been allocated, so before calling `free(p)`, we must ensure that `p` is not `NULL`. Languages that do not support short-circuiting would check `p` *and also* call `free(p)`, resulting in a run-time error. Taking advantage of such language-dependent features can be quite convenient, but for programmers not intimately familiar with the reason for coding the condition this way, and the potential problems that can arise from misusing such constructs, the result can easily be code faults.

We will be using the MCC value as measured by our CASE tool, which does not treat conjunctive conditions as multiple predicates (i.e., it does not measure the extended cyclomatic complexity, but rather the simple version, obtained by counting the predicates, and ignoring conjunctions within them). This will also underestimate the complexity of the control graph, although we do not know the extent to which this will happen.

6) Case statements (`switch` in the C programming language) with a large number of potential branches could unduly raise the apparent MCC value of a program. In software driven by dispatch tables or control codes, MCC may be a poor measure of complexity, because a single multi-branch case statement could imply highly complex code in what most would argue is quite simple code. In [McCabe, 1976], the author indicates that, when the MCC value was pushed above the arbitrary threshold of 10 due to the inclusion of such a case statement, the requirement that the programmer keep the MCC below the threshold was waived, and the programmer was allowed to submit the code despite the high MCC value.

7) By simply counting *instances* of control statements, we ignore the *context* of each individual decision. Intuitively, an `if/then` construct nested five levels deep would appear to be categorically more complex than five sequential `if` statements, yet either would add five to the cyclomatic complexity [Shepperd, 1988].

8) Yet another aspect that McCabe fails to even mention is that of recursion. A simple example of recursion in modern software systems, particularly in operating system design, lies in file systems. Given a hierarchical tree of directories and subdirectories as the internal nodes, with files as the leaf nodes, recursive traversal of the tree is a fundamental feature of the file system code.

   McCabe claimed that the Cyclomatic Complexity of the whole was *equal to* the sum of the Cyclomatic Complexities of the parts [McCabe, 1976]. In fact, the sum of the Cyclomatic Complexities of the parts is only a *lower bound* on the Cyclomatic Complexity of the whole. Reasons for this include:

   - The conjunctive conditional problem (see #5 above);
   - The under-reported number of connected components arising from multiple *re-use,* rather than *hierarchical use* (see #5 above); and
   - The complete ignoring of recursion as multiple repetitions of a control graph (see #8 above).

   For our purposes, however, we will consider the Cyclomatic Complexity of an entire project to be the sum of the cyclomatic complexity values of the individual files making up the project, with the Cyclomatic Complexity for a file defined as the sum of the Cyclomatic Complexities of each function contained within that file.

Even this, however, can be misleading. Consider another example of a project, this time consisting of a main program with cyclomatic complexity of 1, which calls two subroutines, also of cyclomatic complexity of 1, as depicted in Figure 2.3:

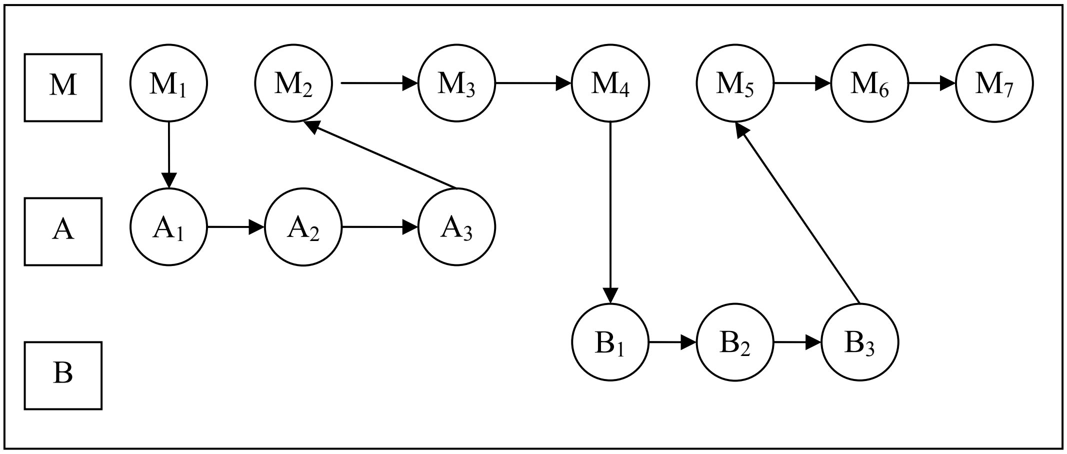


**Figure 2.3 – Main program M and two subroutines A and B**

It should be obvious that M, A, and B all have a cyclomatic complexity of 1, so the entire system would have a cyclomatic complexity of 3, but there is no reason that the code could not have been written in one long routine with a cyclomatic complexity of 1. The fact that it was split into multiple smaller files has artificially raised the apparent cyclomatic complexity. We could have considered the *average* cyclomatic complexity, which in this case would have been 1, and would have solved this problem in this case, but in the case of a program with 100 modules similar to the above, but including one “problem” module with a cyclomatic complexity of 50, the average cyclomatic complexity would be only 1.45. The large number of relatively simple modules would obscure the one “problem” module. Considering the module with the highest cyclomatic complexity value is a worst-case scenario, and completely misses the contribution of the rest of the modules in the project.

9) Despite the validation in [Walsh, 1979] and eighteen other studies cited in [Shepperd, 1988], the validity of MCC has been *challenged* in a number of studies, including [Shepperd, 1988] and [Shepperd, Ince, 1994], on both theoretical and empirical bases. There is a great deal of controversy over the true validity of MCC. Its proponents have claimed numerous rigorous validations against a variety of other variables, such as:

    - LOC
    - Errors, both in absolute number and relative density
    - Programming effort
    - Fault location
    - Program recall
    - Design effort

    [These are all cited individually in Shepperd, 1988]

    On the other hand, McCabe’s detractors seem to think that the cyclomatic complexity, while somehow intuitive in motivation, falls short in one way or another under any of several theoretical and/or empirical arguments. The fact that "complexity" has led to the derivation of a number of variations on the theme suggests that the central theme is perhaps so incomplete that a number of other submetrics have been developed to "shore up" its weak points. The following derivatives of cyclomatic complexity are documented at http://www.mccabe.com/iq_research_metrics.htm:

- Cyclomatic Complexity Metric ($v(G)$) - a measure of the complexity of a module's decision structure. It is the number of linearly independent paths and, therefore, the minimum number of paths that should be tested.
- Actual Complexity Metric (*ac*) - the number of independent paths traversed during testing.
- Module Design Complexity Metric ($iv(G)$) - the complexity of the design-reduced module. It reflects the complexity of the module's calling patterns to its immediate subordinate modules. This metric differentiates between modules which will seriously complicate the design of any program they are part of and modules which simply contain complex computational logic. It is the basis upon which program design and integration complexities (*S0* and *S1*, explained below) are calculated.
- Essential Complexity Metric ($ev(G)$) - a measure of the degree to which a module contains unstructured constructs. This metric measures the degree of structuredness and the quality of the code. It is used to predict the maintenance effort and to help in the modularization process.

- Pathological Complexity Metric (*pv*(*G*)) - a measure of the degree to which a module contains extremely unstructured constructs.
- Design Complexity Metric (*S0*) - measures the amount of interaction between modules in a system.
- Integration Complexity Metric (*S1*) - measures the amount of integration testing necessary to guard against errors.
- Object Integration Complexity Metric (*OS1*) - quantifies the number of tests necessary to fully integrate an object or class into an object-oriented system.
- Global Data Complexity Metric (*gdv*(*G*)) - quantifies the cyclomatic complexity of a module's structure as it relates to global/parameter data. It can be no less than one and no more than the cyclomatic complexity of the original flowgraph.

### 2.3.3 Halstead Volume (Software Science)

In [Halstead, 1977], Halstead proposed a group of five metrics related to computational complexity (and by extension, to maintainability). The metrics are derived from four low-level metrics obtained directly from the source code:

$n_1$ = The number of distinct operators in the program. The determination of what does and does not constitute an operator may be language-dependent. In C, operators include all accesses to simple variables, indexed (array) accesses, accesses to fields, use of unary, binary, and ternary operators, and all function calls. For purposes of this research, all function calls count as a single operator ("function call," without respect to *which* function is being called).

$n_2$ = The number of distinct operands in the program. Operands are considered all identifiers and constants.

$N_1$ = The *total* number of operators in the program.

$N_2$ = The *total* number of operands in the program.

From these four primitives, the following five metrics, forming the basis of Hlasread's "Software Science" can be calculated:

$$\begin{aligned}
\text{Program Length } (N) &= N_1 + N_2 \\
\text{Program Vocabulary } (n) &= n_1 + n_2 \\
\text{Volume } (V) &= N \log_2(n) \\
\text{Difficulty } (D) &= n_1/2 * (N_2/n_2) \\
\text{Effort } (E) &= D * V
\end{aligned}$$

We restrict our analysis to the Halstead Volume ($V$) metric and the four primitives required to compute the volume ($N_1$, $N_2$, $n_1$, and $n_2$).

The Halstead Volume (hereafter, HV) is relatively simple to compute, but it is not practical to do so manually, particularly for large projects. Fortunately, we have used a CASE tool that measures the HV for each module (file) in a project.

The advantages of HV are:

1) It is relatively simple to compute.
2) It addresses some of the shortcomings of McCabe's Cyclomatic Complexity. Namely, it makes an allowance for both the length of the program (at least as measured by the number of operators and operands) and the data complexity of the program (in that it counts the number of operands, which include variables).
3) It has a certain intuitive "correctness" or "viability." Taken at face value, Halstead's rationale behind his metrics seems "reasonable" or at least "plausible" (but see #2 below).
4) There have been numerous studies reporting empirical validation of many of Halstead's metrics. In [Fitzsimmons, Love, 1978], the authors cite 14 major Software Science studies, claiming to show strong correlations ($r^2$ values $\geq 0.90$) in 10 of 22 cases (some studies measured more than one dependent and/or independent variable).

The disadvantages of Halstead's Volume metric are:

1) Because it breaks each statement (where "statement" is roughly equivalent to "LOC") into operands and operators, it is subject to some of the same criticisms as LOC (see Section 2.3.4).
2) Although Halstead's rationale "seems intuitive," science cannot be based on intuition, and all of Halstead's Software Science has been the object of considerable criticism [Shepperd, Ince, 1994].

3) There is widely differing opinion as to the worth of HV. In [Jones, 1994], Jones says that HV is “a more convoluted way [than LOC] of dealing with code…[and] unreliable.” On the other hand, Oman described it “among the strongest measures of maintainability.” (although the question remains of how maintainability could ever be measured in order to compare it to HV).

4) Halstead’s software science metrics were developed during the mid 1970s (and published in 1977). There were no applications in existence at that time to compare with current applications such as the Apache Web Server, the Mozilla browser, or the Linux kernel. The notions of “distinct operators” and “distinct operands” begin to fall apart in the context of programs where the source code is spread across dozens, hundreds, or even thousands of source files. It is not practical to seek out the number of distinct operators and operands.

5) Following on this idea of larger, more modularized programs, how do the Software Science metrics apply three decades later? Halstead validated his metrics against small programs (the first set was the first 14 algorithms published in *Communications of the ACM*, and the largest set used to validate Program Length (N), which is used to calculate V, was “a set of 120 PL/I programs with a combined total of over 100,000 PL/I statements.” Programs averaging 800 lines are simply not representative of today’s large projects, where total size is typically measured in millions of LOC.

6) Much of the work done to validate the Halstead Software Science metrics was done in relation to effort (the relative amount of human effort required to develop, comprehend, or maintain a program). In the context of F/OSS, effort becomes

somewhat irrelevant – there are so many developers potentially available to contribute to a project that effort is rarely a limiting factor.

7) Continuing with the notion of effort, as it relates to comprehension of the code, studies [Aoki et al., 2001], [Dempsey, Weiss, Jones, Greenberg, 2002], [Mockus, Fielding, Herbsleb, 2002] have shown that the bulk of the contributions to F/OSS projects, even on extremely large projects, come from a relatively small number of developers. Given that a small number of developers are likely to have written most of the code, then those same developers will already be familiar with it, and comprehension will start at a high level.

8) It is unclear how to aggregate the total number of operators and operands project-wide on a project such as the Linux kernel. Surely, the "addition" operator will be used in an overwhelming number of files. For programs of a certain size, it is reasonable to expect that the entire set of operators provided by the language may be used.

9) Halstead does not indicate how to handle measuring the number of unique operators and operands relative to the modularization of a program, However, it seems reasonable that, in the case of breaking out a section of code into a new subroutine (procedure, function, or class method), this new subroutine effectively becomes a new operator, because it can be invoked by its name. Thus, in exceedingly large projects, the total number of unique operators will be close to the number of functions, and roughly proportional to the number of lines of code, to the extent that subroutines typically have a somewhat consistent length (in LOC).

### 2.3.4 Lines of Code

Although it would seem to be a simple, clear metric to use, lines of code (or LOC) is considered one of the weakest of all code-based metrics. Jones [Jones, 1994] calls LOC "one of the most imprecise metrics ever used in scientific or engineering writing." There are many aspects of LOC that make it less than straightforward to even capture:

1) Should LOC include blank lines?
2) Should LOC include comment lines?
3) What about lines containing both comments *and* code (in-line comments)?
4) In languages that allow multiple statements per line, should a "line" be a physical line, or an executable statement?
5) Some languages allow a single statement to span multiple physical lines. Should a statement continued on a second (or third) physical line count as one or two (or three)?
6) Some higher-level languages are "code-dense" in that a single statement (one LOC) may equate to a great deal of computational functionality, as opposed to a low-level language (like assembly language) in which a LOC is merely a single machine instruction. Comparing LOC across languages is essentially meaningless.
7) Related to #5, some languages provide macro expansion capabilities, and a single macro line may expand to multiple LOC. In counting LOC, should we count the macro line as one LOC, or should we count LOC only *after* expansion of macros?

Aside from the questions of precisely *how* we measure LOC, it does have a few advantages:

1) Once we determine how we will define and measure LOC, it is extremely easy to measure.
2) Intuitively, longer programs would seem to be more difficult to maintain, more prone to faults (assuming fault density is approximately constant), and more costly to develop and maintain.

LOC also has many disadvantages:

1) It does not reveal anything about the complexity of the code, either within modules or between modules.
2) It does not reveal anything about the volume or complexity of the data and its usage.
3) Unless comments are included in LOC, internal documentation is ignored. Internal documentation can play a strong role in maintenance, testing, and reliability.
4) Different programmers may code the same functionality using significantly differing LOC, so LOC may be strongly influenced by programmer style.

### 2.3.5 Oman's Maintainability Index (MI)

Oman [Coleman, Ash, Lowther, Oman, 1994], [Oman, Hagemeister, 1994], [SEI Oman, 2002] proposed the following polynomial to determine the Maintainability Index (MI) for a program:

$$MI = 171 - 5.2\ln(\overline{HV}) - 0.23\overline{MCC} - 16.2\ln(\overline{LOC}) + 50\sin(\sqrt{2.46\,perCM})$$

*where:*

$\overline{HV}$ is the average Halstead Volume per module for the program,

$\overline{MCC}$ is the average McCabe's Cyclomatic Complexity value,

$\overline{LOC}$ is the average number of lines of code per module for the program, and

$perCM$ is the percentage of lines containing comments in the program.

We will not be examining this latter metric (*perCM*) by itself in this project, simply because it is deemed to be of insignificant value. The volume of comments says nothing about their relevance, correctness, or the degree to which they reflect the current version's code at all (it is quite possible to have current code and obsolete comments). Features are sometimes deprecated by commenting-out entire blocks of code and, as such, this does not even accurately discriminate between what is a comment and what is "old code." Some programmers embellish their comments with borders, extra blank lines, and other cosmetic features that make the comment itself no more valuable. Overall, the number of comments in a program can be relied upon to tell us very little about that program. The CASE tool we are using, however, *does* report the number of lines of comments in a source file, so we can compute MI.

Oman began by issuing a questionnaire on maintainability to software engineers at the Hewlett-Packard Company (HP). The questionnaire was a 25-question, forced-answer instrument with responses on a five-point scale from “very poor” to “very good,” to which Oman assigned the point values of 1 through 5. The resulting responses were summed, yielding a possible score of 25 to 125 for the questionnaire. The resulting scores from the responses of the HP engineers ranged from 61 to 95.

It is important to note that Oman was performing a regression analysis on subjective data, obtained from a questionnaire. Furthermore, the questionnaire was apparently given to only a small number of people. Oman indicates that the questionnaire was given to the Software Engineers at HP who were responsible for the maintenance of eight software suites, totaling about 28,500 LOC at two HP sites. Oman did not indicate the number of engineers to whom the questionnaire was administered, but he did refer to engineers in a plural form, and it seems reasonable to assume that the number of engineers overseeing the maintenance of 28,500 LOC was between 2 and 8 (one per suite). When Oman validated his polynomials, the validation was performed on six suites, totaling about 24,500 LOC, and again, the implication in Oman’s paper is that there were somewhere between 2 and 6 engineers to whom this second subjective questionnaire was administered.

Oman constructed a number of regression models, and identified three that were deemed "potentially useful." One model was based on only one metric, one was based on four metrics, and the last one was based on five. The simpler, one-metric model had a regression correlation of 0.72, while the four- and five-metric models had correlation coefficients of 0.90 and 0.94, respectively.

In all three cases, Oman started with a positive bias constant, and then subtracted from that constant those components that would intuitively seem to detract from maintainability – size (as measured by LOC), complexity (as measured by McCabe's Cyclomatic Complexity), and the difficulty in implementing the solution to a particular problem in software (as expressed in Halstead's Effort metric). In the four- and five-metric models, Oman added a term for the number of comments in the source code, and in the five-metric model, yet another term was added (see below). The three models Oman proposed were:

| Metrics | $R^2$ | Formula |
|---|---|---|
| One | 0.72 | $MI = 125 - 10\log(\overline{HE})$ |
| Four | 0.90 | $MI = 171 - 3.42\ln(\overline{HE}) - 0.23\overline{MCC} - 16.2\ln(\overline{LOC}) + 0.99\overline{CMT}$ |
| Five | 0.95 | $MI = 138 - 2.76\ln(\overline{HE}) - 0.33\overline{MCC} - 12.2\ln(\overline{LOC}) + 0.88\overline{CMT} + 1.04EDOC$ |
| Final | | $MI = 171 - 5.2\ln(\overline{HV}) - 0.23\overline{MCC} - 16.2\ln(\overline{LOC}) + 50\sin(\sqrt{2.46perCM})$ |

Where

$\overline{HE}$ is the average Halstead Effort per module,

$\overline{HV}$ is the average Halstead Volume per module,

$\overline{MCC}$ is the average McCabe Cyclomatic Complexity per module,

$\overline{LOC}$ is the total number of lines of code,

$\overline{CMT}$ is the total number of comment lines in the code, and

*EDOC* is "a 20-point subjective evaluation" (15 points for external documentation and 5 points for the ease of building the system)

*perCM* is the percentage of lines containing comments.

Oman noted that:

1) The Halstead metrics for program effort (*E*) and Volume (*V*) were very strong predictors of maintainability.
2) The Halstead primitive metrics ($n_1$, $n_2$, $N_1$, and $N_2$) added little that was not accounted for in *V* and *E*.
3) The two cyclomatic complexity metrics were equally significant. Either the simple cyclomatic complexity [*V*(*g*)] or the extended cyclomatic complexity [*V*(*g*')] can be used for modeling maintainability; it does not matter which.

In [Coleman, Ash, Lowther, Oman, 1994], the authors chose the four-metric model as most appropriate, and dropped (without explanation) the coefficient of 0.99 from the $\overline{CMT}$ term, essentially changing it from 0.99 to 1. They also noted that, in preliminary testing, the model seemed too sensitive to large numbers of comments, so they replaced $\overline{CMT}$ with *perCM* and a ceiling function to limit the term's maximum contribution to 50 [$50\sin(\sqrt{2.46\,perCM})$]. They also changed the model from Halstead's average-*E* to average-*V* metric to avoid the problem of the nonmonotonicity of average-*E*. Oman had already noted that either average-*E* or average-*V* could be used to explain the majority of the variation observed. The only other change required in switching from average-*E* to average-*V* was the constant coefficient, which was changed from -3.42 to -5.2, resulting in the final formula in the table above.

There are three items that are unclear in the application of this formula:

1) Is the average McCabe Cyclomatic Complexity value to be applied per function, or summed within a file, and then applied per file? Because MI is based on averages, we take the average MCC value over all functions in a file to determine the MI value for the file. We then compute the MI value at the kernel level by taking the average MI value for all files in that version of the kernel.
2) The term $50\sin(\sqrt{2.46 perCM})$ explicitly refers to the *percentage* of lines with comments. This would seem to imply that *perCM* is a value between 0 and 100, in which case 2.46 *perCM* would be a value between 0 and 246, and the square root of that would be a value between 0 and 15.68. Apparently, the intent was that perCM should be the ratio of comments (comment lines / total lines), because the range of $\sqrt{2.46 perCM}$ , as *perCM* ranges from 0 to 1, is 0 to approximately π/2, which certainly makes more sense. We will assume, therefore, that *perCM* is intended as the ratio of comments to the total number of lines,

   If we consider the *perCM* term not as a *percentage*, as the authors indicate, but rather as a *ratio*, ranging between 0 and 1, and treat the sine argument as radians, rather than degrees, we get the plot in Figure 2.4.

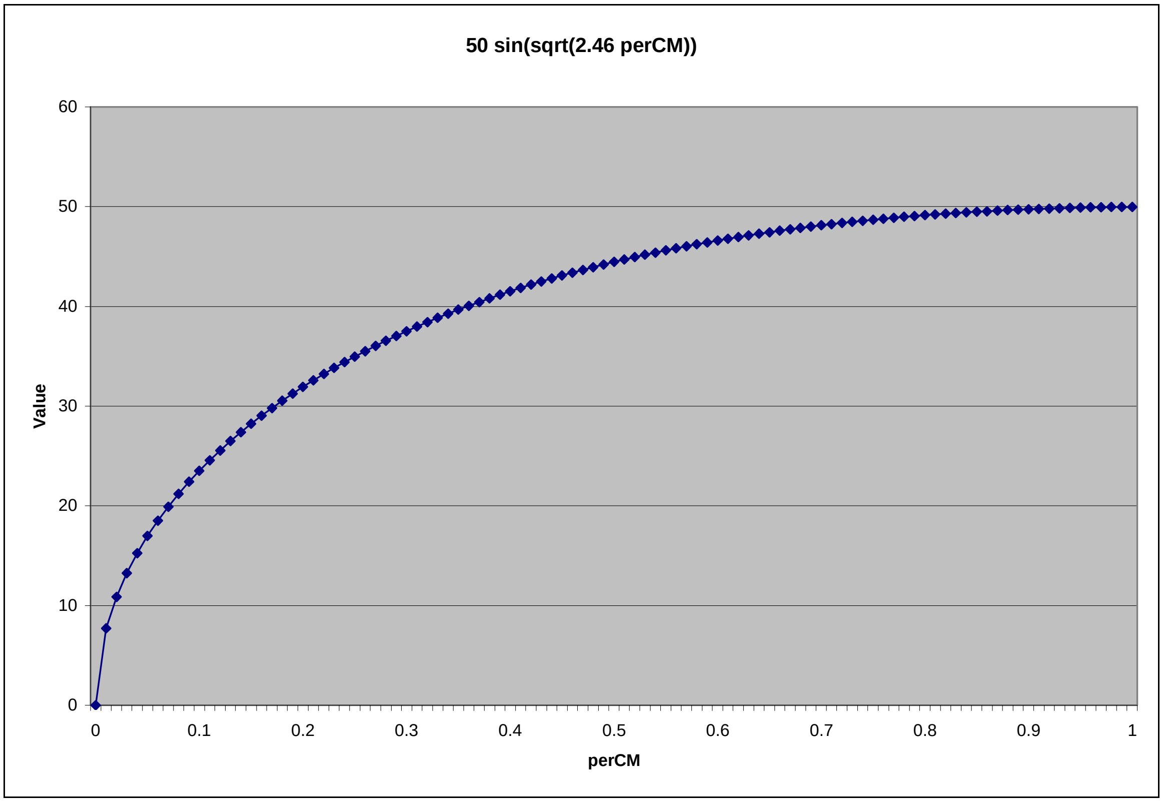


**Figure 2.4 – The correct interpretation of the *perCM* term**

This interpretation has the function monotonically increasing, with emphasized value placed on increases in the number of comment lines when there are very few, and a diminishing return for excessive comments, and utilizing the full 0-50 scale. We will assume that this is the authors' intended interpretation of this term.

Proponents of Oman's Maintainability Index claim that "MI measures maintainability by taking the size, the complexity, and the self-descriptiveness of the code into account…Since MI is based on average values, it is relatively independent of the absolute size… and may be used to compare systems of different size" [Samoladas, Stamelos, Angelis, Oikonomou, 2004]. As we show in Chapter 9, the fact that MI is so *totally* “independent of the absolute size,” it can lead to absurd results (can a million lines of code *actually* be as maintainable as 13 lines?)

Because this metric is based on metrics of arguable validity (Halstead volume, McCabe Cyclomatic Complexity, LOC, and Percentage of Comments), and because it looks at only averages of those four metrics, it would seem to be vulnerable to all of the criticisms of its component metrics. In addition, taking averages across an application with a large number of modules may very well obscure pertinent information.

The coefficients on the respective components in the MI formula were obtained by a regression analysis on eight applications at Hewlett-Packard, four written in C, and four written in Pascal. Precisely what these applications were (application domain) and how well these results generalize to other applications in other languages are not known. If we apply this metric to successive versions of the Linux kernel, however, then we are keeping within a single application, all in the same language, so the examination of a metric of questionable usage across such a homogeneous code base *may* be of value.

3) The final item that is unclear is the interpretation of the term “LOC.” When we consider “percentage of lines with comments,” should we count blank lines as LOC? We will be using non-blank lines, and considering this value as

$$perCM = \frac{Comment}{Comment + NCNB}$$

NCNB (non-comment, non-blank) lines are lines with some non-comment code on them; comment lines are lines that contain only a comment. Lines of code with an inline comment, such as:

```
current++; /* advance to next task in table */
```

will be considered as both an NCNB line, *and* a comment line.

We must remain aware that Oman obtained questionnaire values in the relatively narrow range of 61–95 (from a maximum possible range of 25–125), and then tuned his polynomials to best duplicate those numbers using combinations of the metrics gathered on the code bases. In each case, the polynomial starts with a positive bias constant, and subtracts values from this constant for those metrics which would appear to be detrimental to maintainability (size, complexity, and comprehension effort), and adds terms back in for those metrics (documentation) that should increase maintainability. Because Oman was attempting to fit data in the 25–125 range (the range of his questionnaire), it would appear that MI should lie along a 100-unit scale, from 25 to 125.

### 2.3.6 Klocwork's Risk Metric

Klocwork's CASE tool defines a new metric, whose intent is to indicate the likelihood that the code in a given module contains faults that were introduced during development, but that have been latent (hidden), and escaped unit and system testing. The description of this metric, as listed in Klocwork's documentation [Klocwork Defects 2006] is, "a measure of the file using key complexity characteristics that predict the likelihood of fault insertion during modification or being inherent when created."

The precise formula used to calculate Klocwork's Risk metric is proprietary and confidential; however, somewhat like Oman's Maintainability Index (cf. Section 2.3.5), it is a composite metric based on a weighted sum of multiple lower-level metrics. The relative weights assigned to the respective components of the Risk metric are the result of training the index using actual historic data.

This metric is arbitrarily divided into fault risk ranges of GREEN ($f(x) < 0.1$), YELLOW ($0.1 <= f(x) <=0.2$), and RED ($f(x) > 0.2$).

This index is based "on an historical analysis with C/C++-based systems from a major high-reliability software development project in telecommunications. [This index] was used successfully for predicting software which [sic] contained faults that normally elude unit and system testing (test escapes)." This statement begs the question of how it was possible to conclusively determine when previously-hidden faults were all identified. In other words, the metric claims to have been "used successfully," but no framework or objective criteria for determining that success are presented.

No other validation or theoretical justification is presented for this metric.

We examine this metric in detail in Chapter 9.

## 2.4 Categorization of Instances of Global Variables

Definition-use analysis [Allen, Cocke 1976] divides global variable accesses into “definitions,” in which the value of a global variable is modified (or written), and “uses,” in which the value of the global variable is used (or read), but not modified. Such analysis is typically referred to as “def-use” analysis.

Subsequent research has led to the conclusion that this two-way classification of variable accesses is not always sufficient. In [Yu et al. 2004], the authors present a five-way classification that divides global variable instances into the following five categories:

1. A global variable defined in kernel modules but not used in any kernel modules.
2. A global variable defined in one kernel module and used in one or more kernel modules.
3. A global variable defined in more than one kernel module and used in one or more kernel modules.
4. A global variable defined in one or more nonkernel modules and used in one or more kernel modules.
5. A global variable defined in one or more nonkernel modules and defined an used in one or more kernel modules.

This classification is still basically a def-use analysis, with the added consideration of whether the defs or the uses occur within kernel or nonkernel modules. In [Feitelson at al., 2007], the authors present a different classification of global variables, aimed at classifying each instance of the GV:

1. Simple definitions (e.g., `gv = value;`)
2. Simple use (e.g., `x = gv;`)
3. Combined definition and use (e.g., `gv++;`, which his equivalent to `gv = gv + 1;`)
4. Atomic operations (e.g., atomic test-and-decrement)
5. Passing a GV as a function call parameter by value
6. Passing a GV as a function call parameter by reference
7. Pointer dereference (e.g., `gv->member`)
8. Execution (GV holds a pointer to a function)
9. sizeof

As we shall discuss in depth in Chapter 6, we used a modified version of Feitelson et al.'s classification.

## 2.5 What Constitutes "The Kernel"?

In [Schach et al., 2002] the authors consider "the kernel" to be the files contained in the `/kernel/` subdirectory of the source tree for a given version of the kernel, and they performed their longitudinal analysis accordingly. The authors of [Yu et al., 2004] used this same definition when analyzing the 2.04.020 version of the kernel.

The authors of [Feitelson et al., 2007] proposed three possible alternatives to the approach taken in the two aforementioned studies:

1. Use the `Makefile(s)` to determine which modules are always included in the compilation of a kernel, and analyze those files. This led the authors to a set of 52 files in addition to the 26 in the `/kernel/` subdirectory, but they still made some arbitrary decisions about what to include or exclude, such as networking support. Their study was based on the 2.04.020 version, but the authors did not indicate which files would always be included in the compilation of any other kernel. As such, the set of files included in a longitudinal study could vary from version to version.
2. Include all files required for the compilation of a kernel based on "simplest possible Intel-based i386 platform." This approach led to a total of 342 source files, and 494 header files. The authors note that their "selected configuration was typical of a modern desktop, including Ethernet and USB." It is not clear if their "simplest possible" approach was truly for an i386, or if they used the much newer Pentium 4 as their platform. Support for other platform-specific issues, such as chipsets, was not mentioned. Their decisions regarding what constituted a "modern desktop" were apparently arbitrary. For a longitudinal study, "modern" will vary considerably from year to year (not to mention from decade to decade).
3. Include *all* code except that in the `/arch/` (architecture-specific code) and `/drivers/` (device drivers) subdirectories.

As we discuss in Chapter 5, we developed a completely different approach to determining what the constitutes "the kernel."

## 2.6 Software Evolution

In [Lehman, Ramil, and Sandler, 2001], the authors examined long-term growth trends in software, using sequential release numbers as their independent variable, stating:

> The focus here, as in previous studies … is on size as an indicator of evolution. Why? *Size* has consistently been identified as a significant cost factor… as a determinant of the difficulty, or otherwise, of achieving reliable implementation in a minimal time interval. It has been assumed that it provides a surrogate for the growth of functional power of a system as it is evolved and has been plotted and studied over *release sequence numbers* (rsn), a *pseudo-time* unit… The use of rsn rather than calendar time relates to the fact that only at the moment of release is the software and its documentation, that is the system, fully defined.

Because we were not interested in interpolating between releases, the fact that the software and its documentation are fully synchronized only at the moment of release is immaterial to us. What *is* of concern, however, is the "pseudo" part of "pseudo-time unit."

Other research in software evolution has also focused on sequential release number, rather than release date [Shepperd, 2000]. As we will show in Chapter 8, when the release frequency changes drastically, conclusions regarding trends and rates of growth also change drastically. We question the choice of release sequential number as the independent variable for software evolution.

# CHAPTER III

# LINUX: SOURCE CODE AND TOOLS

## 3.0 Introduction

Any analysis of the Linux kernel's source code begins with the source code itself and a number of tools to be used to manipulate and examine that code. This chapter lists and briefly describes the third-party tools we used in this study; the CASE tools we built are described in Chapter 10.

## 3.1 The Linux Source Code

The first step in building the Linux kernel is to obtain the source code. The various versions are available for downloading on the Internet at `ftp://ftp.kernel.org/pub/linux/kernel`.

When viewing this FTP site in a browser, bear in mind that the versions are sorted numerically, so 2.0.1 is followed by 2.0.10 through 2.0.19 before 2.0.2 appears. It would have been helpful if the version numbers had been padded with leading zeroes, so that 2.0.00 through 2.0.09 would appear in the list before 2.0.10.

At first glance, it appears that Version 2.0.0 is not available for download, but upon closer inspection, it can be found at the same FTP site, but it is listed under "`linux-2.0.tar.bz`," so it appears *after* all of the other 2.0.x versions.

The kernel source archives are all available as either `.tar.bz2` or `.tar.gz` archives. The `.tar.bz2` versions are more highly compressed, and therefore take less time to download and less space to store. The total size of the 2.00.x, 2.02.x, and 2.04.x kernel source archives, in `.tar.bz2` format, is approximately 1.5 GB.

## 3.2 Building Linux Under Linux – the Debian Distribution

The Linux kernel source code archives we downloaded all had to be built (or compiled), which we describe in greater detail in Chapters 5 and 7. The process of building a kernel takes place under a running Linux environment.

Debian (`www.debian.org`) is one of the oldest Linux distributions still in widespread use. In addition to the base distribution (the current version at the time of writing is called `etch`), Debian provides a rich library of pre-compiled *packages* that can be installed to augment the base installation. These are available at `packages.debian.org`.

Debian categorizes its packages as:

- Unstable – the newest, unsupported versions of all packages, awaiting thorough testing;
- Testing – packages currently being tested, but not yet deemed suitably stable; or
- Stable – packages that have been thoroughly tested – the latest official releases.

The "Administrative Tools" section of the stable package library includes `synaptic`, a GUI package manager used to download and install the other required packages.

The desktop environment used in this dissertation was KDE 3.5, available from the "KDE" section of the stable package library. KDE includes a GUI text editor called `KWrite.`

The stable package library contains a section called "development" which contains the following packages:

- `make` (v.3.81-2) The GNU version of the "make" utility.
- `gcc272` (v.2.7.2.3-19) The GNU C compiler, version 2.72.
- `gcc-2.95` (v.2.95.4-27) The GNU C compiler, version 2.95.
- `gcc272-docs` (v.2.7.2.3-19) Documentation for the `gcc` compiler (`gcc272`).
- `gcc-2.95-doc` Documentation for the GNU compilers (`gcc` 2.95)
- `cxref` (1.6a-1.1) Generates LaTeX and HTML documentation for C programs

The stable package library contains a section called "interpreters" which includes the following:

- `cpp-2.95` (2.95.4-27) – The GNU C preprocessor (for `gcc` 2.95).
- `gawk` (3.1.5.dfsg-4) – GNU `awk`, a pattern scanning and processing language.

The stable package library contains a section called "utilities" which includes the following:

- `rpl` (1.5.4) – intelligent recursive search/replace utility.

Installing the above packages also requires the following support packages (although the `synaptic` package manager can handle these dependencies automatically, and download and install them as required):

- `binutils` – The GNU assembler, linker, and binary utilities.
- `libc6` – GNU C Library: Shared Libraries.
- `cxref-doc` – Documentation for CXREF.

## 3.3 CXREF

CXREF (http://www.gedanken.demon.co.uk/cxref/) is a utility designed to document, index, and cross-reference several components of C (C only; not C++) source code (variables, TYPEs, `typedef`s, and functions), but the specific items of interest to us in this dissertation are global variables. CXREF shows both the visibility and usage of global variables in C files. As indicated above, CXREF is available as a package from the Debian stable package library.

## 3.4 Klocwork

Klocwork (www.klocwork.com) produces a CASE tool called the "K7 Development Suite." This tool (version 7.5) was used to extract metrics from the Linux kernel source code, and is discussed in more detail in Chapters 5 and 7.

## 3.5 VMWare Workstation

VMWare Workstation (www.VMWare.com) allows the creation and simultaneous execution of up to four virtual machines (VMs) under the Windows XP Operating system. Using VMWare, it is possible to have up to four distinct, fully-independent instances of Linux running at once, each in its own window. VMWare also supports the use of "Shared Folders," in which a Windows file system directory on the host machine is visible to the virtual machine. This facilitates the transfer of files between the host (Windows) and guest (Linux) machines.

## 3.6 WinRAR

WinRAR (www.rarlabs.com) is an archive program along the same lines as WinZip (www.winzip.com). WinRAR handles `.tar.gzip` and `.tar.bzip2` archives created under Linux, and as such is a natural tool to use under the Windows for manipulating Linux archives.

## 3.7 MySQL

We quickly determined that a spreadsheet, such as Microsoft Excel, would not be a viable option for the storage, management, and analysis of the resulting code metrics because of the vast amount of data that needed to be stored. We clearly would need a relational database. We then considered Microsoft Access, but it is limited to a database file size of 2 GB, and we already knew that we would need a much larger database file than that. Microsoft's SQL Server 2005 is a full-featured, industrial-strength closed-source database server, but its freely-available version is limited to a 4 GB database file size. We ultimately decided to use the Open-Source (GPL) MySQL Community Server (version 5.0.51), available from www.MySQL.com. Our 8.7 GB database is managed using the MyISAM storage engine.

# CHAPTER IV

# WHY REVISIT SCHACH ET AL.?

## 4.1 Introduction - Aspects of [Schach et al. 2002] to Reconsider

A Google Scholar search (http://scholar.google.com) conducted on March 29, 2008, on "Linux Maintainability" located approximately 3,830 articles. First on the list, cited by 57 other articles, is [Schach et al. 2002]. Clearly, this work is considered the authority on maintainability of the Linux kernel. The primary conclusions of this study were that the size of the Linux kernel grows *linearly* with respect to version number, and that the number of incidents of common coupling in the Linux kernel grows *exponentially* with respect to version number.

Given that Schach et al. have already examined the rate of growth of common coupling in the Linux kernel, why should we revisit their work? Beyond extending their work, augmenting their results with the corresponding values from kernels released since their work (evolution), how can we provide any substantial changes to their research (revolution)?

The answers to this question lie in a closer examination of their work, the way in which their results were obtained, and even the kernel versions they examined. More specifically, we reconsidered their work and made fundamental changes with respect to the following key areas:

1. The code base considered (the `/linux/` subdirectory);
2. Kernel versions examined (production vs. development, and disregarding subsequent releases once a new branch begins); and
3. Intervals used to determine rate of growth.

We next consider each of these individually.

We remark in passing that Schach et al. deduced on a *statistical* basis that the growth in instances of coupling was exponential: They obtained a better fit to an exponential growth curve than to a quadratic growth curve. In our opinion, there is no theoretical basis behind their claim. Their claim of exponential growth, though strongly supported by the data that they used, may be an artifact of their data. We return to this point in Chapter 8, where we discuss their claim in the light of possible weaknesses in their selection of data (Section 4.3) or in their choice of independent variable (Section 4.4).

## 4.2 The Code Base to Consider - What Constitutes the Linux Kernel?

As discussed in Section 2.5, previous research on the Linux kernel has taken several different approaches to identifying precisely what constitutes the Linux kernel, each of which has advantages and disadvantages:

1. We could consider the source code in only the `/kernel/` subdirectory. Advantages of this approach include:
   - The number of source code files to examine in the `/kernel/` subdirectory is reasonably small (14 in 2.00.000, up to 26 in 2.04.035).

- The functionality provided by these files constitutes a large part of the core functionality of a final kernel. Some of the features implemented in files contained in this subdirectory include:
    - DMA (direct memory access) accelerates transfers between main memory and hardware devices. The code in file `dma.c` provides DMA transfer support.
    - Exiting: File `exit.c` handles the destruction of processes at their completion.
    - Forking: File `fork.c` handles the creation of new processes.
    - Scheduling. File `sched.c` handles task execution scheduling.
    - Interprocess communication: File `signal.c` handles the sending and receiving of signaling between tasks. This is a high-level form of interprocess communication.
    - Timekeeping. File `time.c` provides a number of timekeeping functions.
- Almost all of these files are used in virtually all kernel builds.

Some disadvantages of this approach include:

- Not all of the files in the `/kernel/` subdirectory are used in all kernel builds. For example, the file `acct.c` implements the optional process-level accounting.
- These files do not implement *all* of the required subsystems an operational kernel must provide. Any functioning kernel must also include files from several *other* subdirectories:

- `/mm/` – (memory management),
- `/arch/` – the architecture-specific hardware-level interfaces and implementation of low-level routines (largely in assembly language, for performance).
- `/fs/` – Any kernel must have at least one file system. Systems with only a hard drive, CD-ROM, and floppy support (a rather minimal configuration) will require support for a minimum of least three file systems, one for each these device types.
- `/init/` – The kernel has initialization code that must also be compiled. If we are to consider the kernel in its entirety, then the code required to load and initialize it must also be considered.
- `/net/` – Although not essential, virtually all modern kernels will include some form of networking.
- `/ipc/` – The `signal` mechanism mentioned above is a high-level construct. The lower-level inter-process communication mechanisms, including message passing, semaphores, and shared memory spaces, are managed by the source files in the `/ipc/` subdirectory.
- `/drivers/` – Each particular hardware device requires some form of driver in order to interface the hardware with the kernel. This can range from very low-level hardware support (such as motherboard chipsets, including interrupt controllers, real-time clocks, etc.) to high-level devices such as video, networking, and sound cards, and either SCSI or IDE disk controllers.

- Each of these `/kernel/` source files may `#include` many header files. This leads to the question of whether or not to also include these header files in the analysis. Further complicating this situation is the fact that some of the `#include` statements for the headers appear within conditional compilation statements (`#ifdef`, `#idndef`, `#else`, `#endif`). Determining precisely which header files may or may not ultimately be `#included` is not straightforward.
- Further complicating the `#include` file issue is the fact that some header files are `#included` more than once. Code that would ultimately appear in several files should be given special attention, particularly from a maintenance standpoint, because changes to such code in header files could introduce regression faults in a number of otherwise unmodified source files.

Clearly, considering the files from only the `/kernel/` subdirectory is not an optimal solution.

2. We could consider the *entire* body of source code in a given kernel release. This approach certainly solves the problem of not including significant portions of the kernel (`/mm/`, `/fs/`, `/ipc/`, etc.), but gives too much weight to drivers for hardware we either do not have, or for unlikely or even impossible hardware configurations. Such impossible configurations would include multiple sound cards, multiple network cards, multiple SCSI cards, and so on. Although it is possible to make the argument for a system with two network cards, operating as a network

bridge, or for a file server with multiple SCSI disk controllers, or even a graphics-oriented system with two video cards, such systems are not representative of a typical configuration. Moreover, systems with multiple video cards, multiple network cards, multiple sound cards, and multiple SCSI cards are simply impossible to build, because there would be more drivers than there are physical slots in any system.

The 2.00.000 kernel has 289 `.c` source files in (and below) the `/drivers/` subdirectory, while the 2.04.035 kernel has 2,140 such files. Clearly, including the entire source code base does not provide a representative view of a "typical" kernel.

Another problem with considering the entire code base for the kernel is that it includes architecture-specific code for a number of platforms. Linus Torvalds originally wrote the kernel for only the i386 platform, but the kernel now supports over a dozen different architectures, ranging from hand-held devices to large mainframe computers. Just as no physical system will ever have a dozen different sound cards, no kernel will ever simultaneously run on multiple architectures.

However, if one were considering only maintenance aspects of the kernel, it might make sense to consider the entire code base, because the entire code base has to be maintained.

3. For purposes of this dissertation, however, we took a third approach. First, we attempted to restrict our analysis to a viable, practical configuration on the i386 hardware platform. However, the adjectives "viable" and "practical" are subjective and open to interpretation, whereas we needed a purely objective means of

determining the appropriate hardware configuration to select. This approach is described in Chapter 5.

Schach et al. used the first approach, considering the code in only the `/kernel/` subdirectory. As we have shown, this is an extremely narrow view of the code, and is therefore not necessarily representative of the entire kernel.

### 4.3 Kernel Versions Examined (Production vs. Development)

In [Schach et al., 2002], the authors examined *all* available (at that time) versions of the kernel, except for those versions in an older branch that were released later than those in a newer branch (see below). The Linux kernel, up through versions 2.05.x, has seen two broad types of development cycles – production cycles and development cycles.

During production cycles (those for which the second part of the version number is even – 1.00.x, 1.02.x, 2.00.x, etc.), the code is considered to be essentially stable, and the changes made are relatively infrequent and corrective in nature [Schach, 2007, p. 9]. The changes made during the development cycles (those for which the second part of the version number is odd – 1.01.x, 1.03.x, 2.01.x, etc.) are characterized primarily by enhancements and exceedingly frequent releases. The interim code of the development cycles is considered too unstable for general use. In contrast, the production versions are considered stable releases, with modifications made primarily for security and fault fixing, although some new features *have* debuted in production versions.

Because the development releases are interim versions for the benefit of the development community, and because the production releases are intended for the (much) broader user community, we examined only the production versions.

For all practical purposes, the Linux kernel code forks every time a new development cycle is initiated.  Once a new development cycle began, Schach et al. stopped considering subsequent versions from the old code (production) branch.  To put specific dates and version numbers to this idea, consider Figure 4.1.

The 2.02.x code branch, shown down the left side of Figure 4.1, ran from 2.02.000, released on 1/26/1999, through 2.02.026, released on 2/25/2004.  On 5/19/1999, the same day the 2.02.008 kernel was released, the 2.03.000 development cycle began.  We have drawn the 2.03.x releases as overlapped, to emphasize the fact that the development versions are released much more frequently than production versions.  In this case, the 53 versions of the 2.03.x development cycle were released in the same time frame as the 8 production releases 2.02.008 through 2.02.015.  The last development release of the 2.03.x branch occurred on March 13, 2000, but 2.04.000 was not released until January 4, 2001, between the releases of 2.02.018 and 2.02.019.

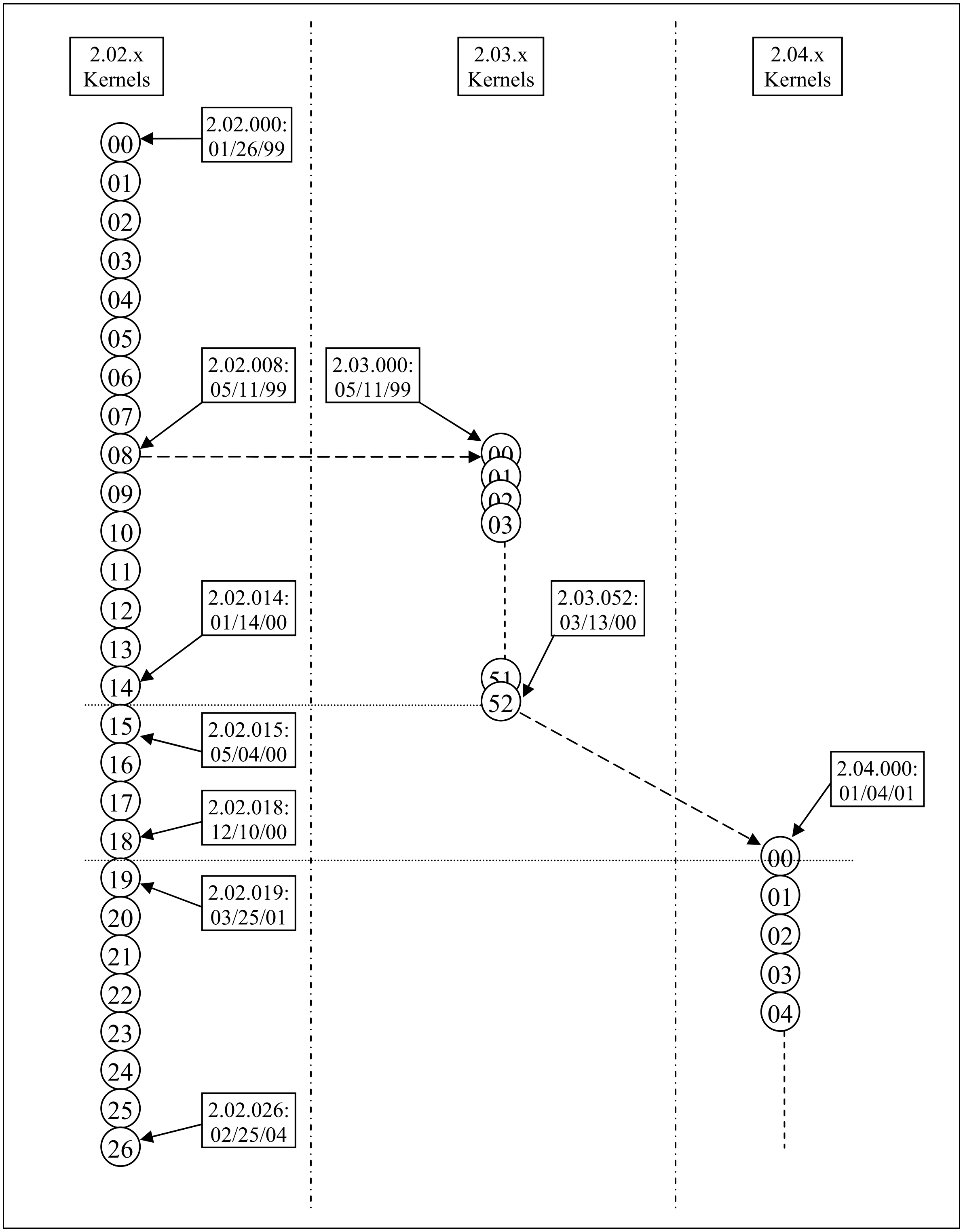


**Figure 4.1 – Forking of Linux kernel versions 2.02.x, 2.03.x, and 2.04.x**

The development-cycle versions do not fork. When the new production branch began (2.04.000 in this case), it began because the development cycle had concluded. Therefore, the development branch does not continue. Some of the production kernels have exceedingly long life spans. As shown in Figure 4.1, there were 8 more releases of the 2.02.x kernels in the next three years after the 2.04.x branch began.

Schach et al. stopped looking at one branch once a new branch had forked off from the old one. In this example, once 2.02.08 was released, and the 2.03.x development branch began, they stopped considering the 2.02.x kernels. Unfortunately, this led to examination of only the first 9 of the 27 production versions of the 2.02 branch (2.02.000 – 2.02.008), though all of the 53 versions of the development branch were included. Including 9 production kernels and 53 development kernels certainly skews the results towards the development code, in which we are not interested.

The same thing happened between production kernel series 2.00.x and 2.02.x. The 2.01.x development branch was initiated between production kernels 2.00.021 and 2.00.022. As such, Schach et al. did not examine kernel versions 2.00.022 through 2.00.040; rather, they considered the 22 versions from 2.00.000 – 2.00.021 and then jumped to 2.01.000 – 2.01.132, resulting in the inclusion of 22 production kernels and 133 development kernels. Again, this heavily skewed the versions examined towards the development kernels.

The plot in Figure 4.2 shows all releases of the 2.00.x, 2.01.x, 2.02.x, 2.03.x, and 2.04.x kernels. The X-axis shows the release date, and the Y-axis is the "x" part of 2.00.x, 2.01.x, et cetera; that is, the Y-axis is the version number within a particular development or production cycle. The tallest series plotted on this graph shows that the 2.01.x development cycle ran from late 1996 until just before January 1999 (X-axis), with the 133 versions running from 2.01.000 through 2.01.132 along the Y-axis. The larger the X-gap between two data points, the more time elapsed between releases. Another way to view this is that the steeper the line, the higher the release frequency.

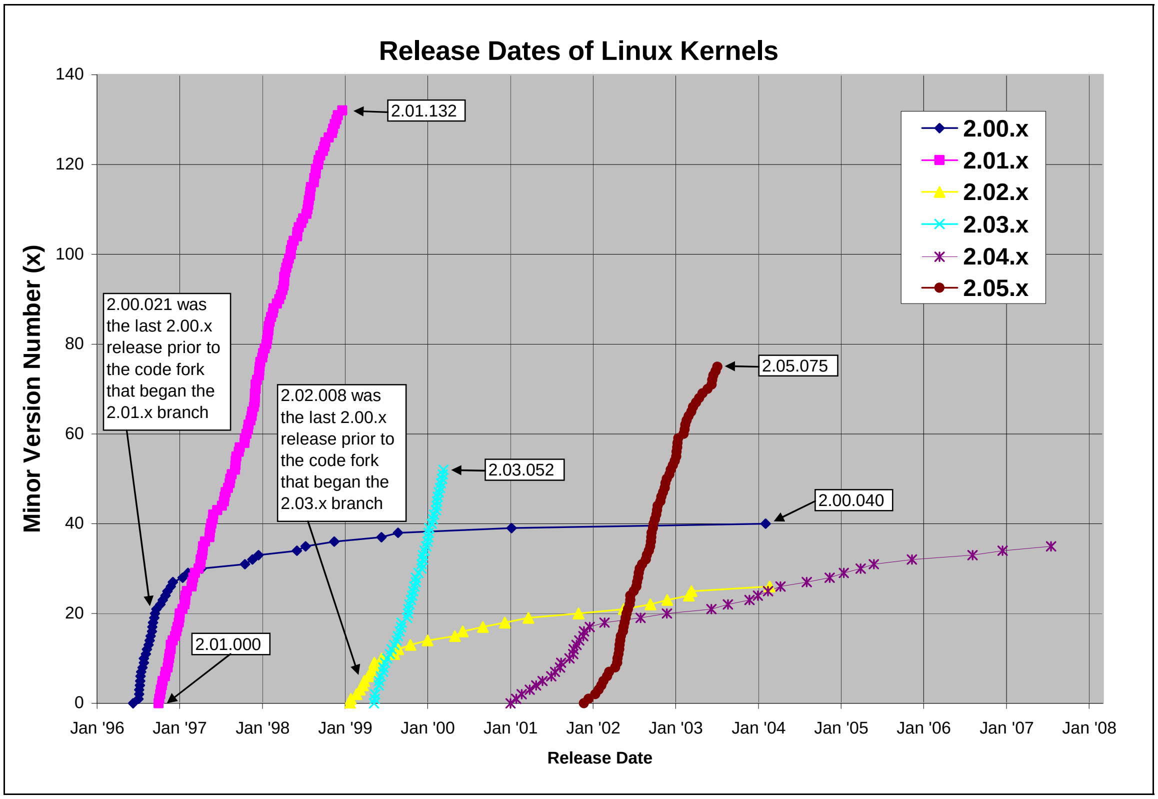


**Figure 4.2 – Kernel Releases 2.00.x, 2.01.x, 2.02.x, 2.03.x, 2.04.x, and 2.05.x.**

Had we used the same approach as Schach et al., we would have stopped considering releases in older versions following a fork, and, as such, all points in the 2.00.x series after 2.00.021 would not have been considered (because 2.01.000 was released between 2.00.021 and 2.00.022). Similarly, as discussed above, we would not have considered points in the 2.02.x series after 2.02.008, because 2.03.000 was released the same day as 2.02.008. This would have left us with the kernel versions shown in Figure 4.3, heavily skewed towards the development versions, and discarding half of the 2.00.x data, and two-thirds of the 2.02.x data.

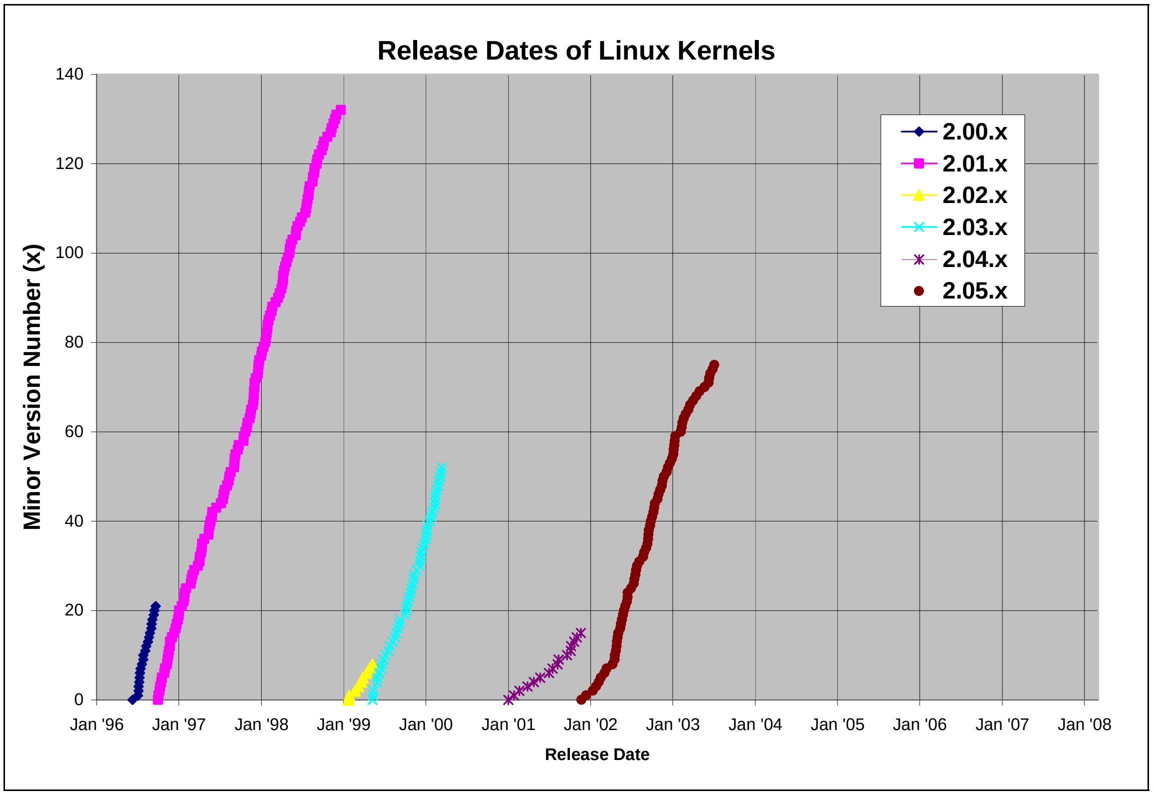


**Figure 4.3 – Kernel Releases 2.00.x, 2.01.x, 2.02.x, 2.03.x, 2.04.x, and 2.05.x.**

As explained at the beginning of this section, we considered *all* production kernels from 2.00.000 through 2.04.035, and *no* development kernels, resulting in the versions shown in Figure 4.4.

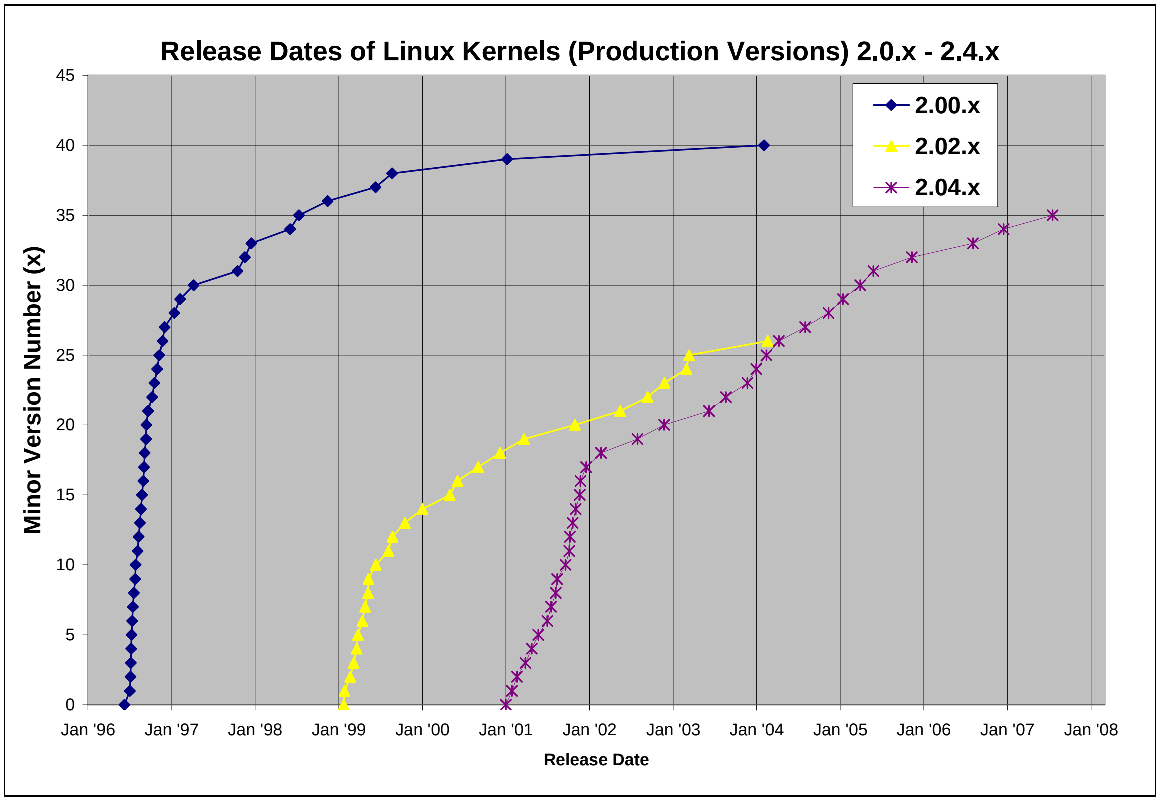


**Figure 4.4 – Kernel Releases 2.00.x, 2.02.x, and 2.04.x.**

## 4.4 Intervals used to determine rate of growth

Schach et al. reached their conclusions regarding the rate of growth in the kernel (as measured in lines of code), as well as the rate of growth in the number of instances of common coupling in the kernel, using serial release numbers as their independent variable (i.e., the X-axis of their growth plots was the kernel version number). The problem with this is that the kernels were not released on a regular schedule. Therefore, to say the kernel is growing linearly or that common coupling is growing exponentially, when the data points are kernel version numbers, rather than release dates, does not accurately depict the rate of growth with respect to time. This difference may have changed the apparent rate of growth, because the time per unit across the x-axis was not constant.

This problem is exacerbated with the difference in the frequency of releases between the production and development kernels. The 2.00.x, 2.02.x, and 2.04.x kernels had an average release frequency of about 61 – 74 days between releases. In contrast, the 2.01.x and 2.03.x development branches averaged one release every 6 days (approximately). Even within a branch, the release frequency can vary significantly. For example, the first 14 releases of the 2.02.x branch occurred at an average rate of approximately one version every 20.5 days, but the next 13 releases occurred at an average rate of approximately one version only every 126 days.

If all kernel releases had occurred at fixed, regular intervals, then the three series plotted in Figure 4.4 would have all been parallel straight lines. We used the release date as our independent variable for all rate-of-growth questions.

In this chapter, we have described the ways in which our approach differs from that of [Schach et al. 2002]. The results of our research using our approach are presented in Chapter 8.

# CHAPTER V

# SELECTING A CONFIGURATION FOR EMPIRICAL RESEARCH ON THE LINUX KERNEL

## Contributions in this Chapter

2. Previous studies have concentrated on examination of the source code in only the `/linux/` subdirectory. That approach captures only a tiny portion of the source code base. We have examined the entire source code base and constructed complete, fully configured kernels, actually capable of running on real hardware.

3. Because the Linux kernel is so highly configurable, the set of configuration options has a profound impact on the code base selected for compilation. We have developed a framework for selecting a consistent configuration across multiple versions of the Linux kernel, which makes longitudinal studies more readily comparable. This method can be extended to other operating systems, and to software product lines that are hardware-dependent.

## 5.1 Chapter Introduction

In Chapter 4, we explained why we revisited the work of Schach et al. There were several aspects of their work that we wish to reconsider, and one aspect was that of the code base they analyzed. Schach et al. examined the source code in the `/linux/`

subdirectory, but, as we discussed in Chapter 4, we elected to examine the entire code base used to build a *complete* kernel, capable of being loaded into memory and executed, rather than the small set of files they examined. Part of the process of building a complete kernel is the specification of the configuration for which support needs to be included. But to compare multiple kernels, a "consistent" configuration is required from version to version. In this chapter, we present our procedure for determining the consistent configuration for multiple versions of the Linux kernel.

There are four main steps required to compile the Linux kernel, each an invocation of `make`:

```
1. make mrproper
2. make config
3. make dep
4. make
```

The first step, `make mrproper,` deletes any potential remaining vestiges of previous builds, leaving `make` a pristine environment in which to perform the current build (which starts in the second step, `make config`). The third step, `make dep`, sets up the file dependency information for the build, and the last step, `make`, actually invokes the tools to build the kernel.

In this chapter, we develop a method for creating a "consistent" kernel configuration. Once such a method is created, we can then determine our responses to the configuration prompts in the second step above (`make config`), and proceed with actually compiling the kernel. We require a "consistent" configuration to use across all versions we analyze;

we define “consistent” more precisely in Section 5.3. Our responses to the configuration prompts in this step collectively define our particular configuration so that the appropriate hardware-specific and feature-specific kernel code can be included. Some versions of the kernel support menu-driven configuration utilities (`make menuconfig`) or even GUI-based configuration tools (`make xconfig`, which uses the X11 window system). However, *all* versions of the kernel support the text-based `make config` approach, in which the configuration script prompts the user for “yes/no” configuration selections. We use this latter approach for all kernels.

The responses to the configuration questions determine not only high-level subsystem issues like which file systems to include and whether or not to include support for networking, they also determine how those subsystems will be configured (for example, if networking is selected, then which protocol(s) are to be supported, which specific network card should be selected, and so forth).

## 5.2 Kernel Version Series Chosen for Inclusion in This Study

In addition to its pre-release 0.x versions, the Linux kernel has been released in two primary sets of versions – the 1.x versions, and the 2.x versions.

We elected to exclude the version 1.x kernels from our study for three main reasons:

1) The version 1.x kernels were fundamentally different from the 2.x kernels in several ways. Perhaps the most significant difference was that the 1.00.x kernels supported only the i386 architecture. The code was not structured in a way that allowed any

other architecture until some time during the 1.01.x development cycle. In its 1.00.x and 1.02.x incarnations, the Linux kernel was simply an immature product.

2) The 1.x series did not have a sufficient number of versions released to be statistically significant. Only ten versions of the 1.00.x kernel (1.00.000 – 1.00.009) were released, all within a 34-day period (13-Mar-1994 – 16-Apr-1994), and only 14 versions of the 1.02.x kernel (1.02.000 – 1.02.013) were released, over the 104-day period of 07-Mar-1995 – 02-Aug-1995.

3) As Wikipedia indicates, the changing of the first part of the Linux kernel version number (from "1.x" to "2.x") occurs "only when major changes in the code and the concept of the kernel occur." This further reinforced our opinion that the 1.x and 2.x kernels were so fundamentally different that comparisons between the two would be unreliable at best, and meaningless at worst.

Once it was decided that we would consider only the 2.x kernels, we had to choose which of the 2.x kernels we would consider. The development model of the 2.x kernels, up through the 2.05.x series, was the same – alternating production (stable) and development cycles. The production (stable) versions are the ones running in the "real world" on servers, workstations, and embedded controllers, whereas the development versions are primarily of interest to only testers, experimenters, and the developers themselves. As such, we considered the latter groups to be a small minority of the Linux base, and not representative of Linux as a whole. Beginning with the 2.06.x series, the development and production series merged, so there is no way to consider only the production versions in the 2.06.x series. Accordingly, the versions of the 2.06.x series were excluded from this research.

At the time this project began, there were 41 versions of the 2.00.x series, 27 versions of the 2.02.x series, and 33 versions of the 2.04.x series, just enough for reliable statistical analysis. The 34th, 35th, and 36th versions of the 2.04.x series (versions 2.04.033, 2.04.034, and 2.04.035, respectively) were subsequently released (and which we included in this research); a 37th version (2.04.036) was also released, although it was released too late to include in this study. Based on the small number of lines of code added, changed, and/or deleted in this 37th version of the 2.04.x kernel series, and the fact that the 2.06.x series was already over four years old at the time of its release (suggesting that most 2.04.x users had migrated to the 2.06.x series), it is unlikely that any further significant development will occur in a 2.04.x series kernel. Therefore, omitting the 37th version would have had negligible impact on our results; the 37th version was virtually identical to the 36th, and both were of questionable relevance, as we discuss in Chapter 8.

For all of the reasons stated above, we chose to limit our analysis to the 2.00.x, 2.02.x, and 2.04.x kernels. Due to its release date, we did not include 2.04.036; otherwise, we examined every version from 2.00.000 through 2.04.035, inclusive. In all, there were 104 versions of the kernel in our study, as shown in Table 5.1

**Table 5.1 – Linux kernel versions included in this study**

| Series | Versions | Total Count |
|---|---|---|
| 2.00.x | 2.00.000 – 2.00.040 | 41 |
| 2.02.x | 2.02.000 – 2.00.026 | 27 |
| 2.04.x | 2.04.000 – 2.04.035 | 36 |
|  | Total: | 104 |

As we explain in Chapter 8, the last few versions of each series were ultimately dropped from consideration, resulting in a total of 83 versions actually being analyzed.

### 5.3 Selecting a Configuration for Empirical Research on the Linux Kernel

Given that we wished to select a configuration for empirical research on the Linux kernel, there were four broad approaches we could use, each with advantages and disadvantages (although the disadvantages were sometimes unacceptable):

1) Use `make config` as a tool, and simply choose the *default* configuration for all versions. This has the advantage of being exceedingly simple, and avoids the possibility of being challenged as to why particular selections were made. The problem is that this approach does not yield an apples-to-apples analysis. For example, in the case of the SCSI subsystem, the default response to the question about including SCSI support for the 2.00.x kernels is "N" ("No, do not include SCSI support"), whereas the default response for the same question in the 2.02.x and 2.04.x kernels is "Y." Adopting a strict blindly-go-with-the-default approach, would result in completely missing the SCSI subsystem in 2.00.x.

Comparing the code from a kernel with the SCSI subsystem to a kernel without the SCSI subsystem reduces the degree to which the comparison is direct. Some of this is inevitable; features come and go, but it is vital to minimize the degree to which this impacts comparisons between kernels.

Moreover, the sound subsystem also defaults to “N” in 2.00.x and to “Y” in 2.02.x and 2.04.x, so at least two major subsystems would be missed if the approach of simply accepting all defaults were adopted.

(Although it is out-of-scope for this dissertation, we remark that the 2.06.x kernel takes yet a different approach, and defaults nearly everything to “Y.” This solves the problem of missing key subsystems, but it also bloats the code base with every conceivable option, no matter how obscure. Furthermore, in the case of SCSI, Network, and Sound cards, support for EVERY respective card defaults to “Y.” Clearly, when “pick one” is the appropriate approach, accepting “Y” as the default to *every* one is not at all acceptable.)

2) *Arbitrarily select* a configuration for each branch of the kernel source. This approach, if correctly carried out, could result in the appropriate subsystems being included, but it also exposes every configuration choice to the challenge – “Why include (or exclude) *this* option or *that* one?” Furthermore, what exactly constitutes an “appropriate subsystem”? Some options are much more likely to apply to servers (such as SCSI and high-speed networking), whereas other options would be more applicable to laptops (such as power management and wireless networking). Clearly, any arbitrarily chosen configuration would make the analysis valid for only that

particular case, rather than broadly applicable, no matter how carefully the set of configurations were chosen, based on expert knowledge of real-world systems.

3) Choose the *same* configuration for all versions of Linux. For the most part, each successive version of Linux is a superset of the preceding one. Functionality and features are constantly being added in response to user demand, technological advances, and other factors. Although it is mostly possible to select the same functionality across all versions, thereby guaranteeing an apples-to-apples comparison, the only way this can be achieved is to *disable* all new functionality in newer versions. This removes the aspect of growth, and results in a new kernel that behaves essentially like an old one. There is not much sense in stripping out all the new features of the new kernels just to make them look like the old ones. If the resulting new kernels implement only the functionality of the old ones, then the code required to implement the same functionality probably looks quite similar, and in fact, should yield rather consistent results from version-to-version. This seems, at best, to be an academic exercise with little practical applicability.

4) Select a *consistent* configuration, using a *default-based* method to merge the defaults with a few knowledge-based changes to handle the exceptions that the method's rules do not directly handle. This is the approach we have chosen for this dissertation.

As mentioned in Section 5.2, the newest version of the kernel available when this dissertation began was 2.04.034. Version 2.04.035 was released in time to be included in this study, but version 2.04.036 was not. The differences in the configuration options between versions 2.04.034 and 2.04.035 were inconsequential. We developed the configuration method below using 2.04.034. However, had we used 2.04.035, the resulting set of configurations would have been identical.

Selecting a consistent configuration requires a "both ends against the middle" approach:

- We cannot just start with the 2.00.000 kernel and go forward. A prime example of how doing so would cause problems can be seen in the SCSI system. In versions 2.00.x, support for the SCSI system defaulted to "N," though it *was* available. Had we started at 2.00.000, selecting the defaults and working forward, we would have accepted the "N" default for 2.00.000, and run into an inconsistency when the 2.02.000 default became "Y." Similarly, Plug-and-Play support, mouse support, and sound card support all first appeared with "N" as the default.
- Similarly, we cannot just start with the 2.04.034 kernel and go backward for the same reason – it leads to an inconsistent configuration, but not in quite the same way. Consider the SCSI subsystem. SCSI support defaults to "Y" in version 2.04.034, and had we started with 2.04.034 and gone backwards, we could have continued to select "Y" all the way back to 2.00.000, but SCSI support requires the selection of a SCSI controller *card*. The default SCSI card for 2.04.034 was the SYM53C8XX series. Unfortunately, support for this card did not exist in version 2.00.000 (support for this card first appeared in 2.02.006). Therefore, if we had started with version 2.04.034 and gone backwards, we would have selected support for the SCSI subsystem (which we *could* have continued to select

all the way back to 2.00.000), but not for the default card (which we *could not* have continued to select all the way back to 2.00.000) – inevitably leading to an inconsistency. Rather, we had to identify a card that existed in 2.00.000 that was still supported in 2.04.034. For some configuration options, support was deprecated along the way, so identifying a card for which support existed in 2.00.000 was necessary but not sufficient for obtaining a consistent configuration.

Accordingly, we had to start with the 2.04.034 end of the spectrum and work our way backward, but, as we did so, each step had to be based on keeping the configuration as consistent as possible from 2.00.000 through 2.04.034.

To determine the appropriate response to most of the configuration options, we used the following method:

```
For each configuration option in 2.04.034
   If the default configuration selection is “Y,” then
      If that option was available in some previous version (whether
                or not it was the default in that previous version),
                then
          Select this item in all builds to be analyzed.
        Else
          De-select this item in all builds to be analyzed
        End if
     Else
       De-select this item in all builds to be analyzed
   End if
Next configuration option
```

This method yields consistent configurations in all but a few areas, which were handled individually, as discussed below. These areas predominately coincide with technological advancements as they move from “leading-edge” to “mainstream,” such as new CPUs, bus architectures, power management, device detection, and so on. As the hardware has matured, the software has naturally had to mature to keep up. A few adjustments had to be made to accommodate these changes.

We ran through the 2.04.034 configuration, first responding to each prompt (there are approximately 425) with "?," forcing the display of the extended help information for all configuration options, and then responding again with our desired choice. In the case of configuration items that had default values of "Y" in 2.04.034, and which existed in some previous version (whether or not they were also the default choice in that previous version), or those that had default values of "N" in 2.04.034, there was no decision to make – we selected the 2.04.034 default.

The remaining issues were:

A) Symmetric Multiprocessing (SMP – supporting multiple CPUs) defaults to "Y" in all 2.02.x and 2.04.x kernels. SMP support *is* available in the 2.00.x kernels, but it is not selected via a `make config` option; rather this option is selected in the `Makefile` by un-commenting the line "`# SMP = 1`." This defines the environment variable `SMP`, and the 2.00.000 code is replete with references to "`__SMP__`." Clearly, SMP support *is* available in 2.00.x; it simply is not selected via the `make config` process. We un-commented this line in the 2.00.x `Makefiles` to enable the feature.

B) The default CPU for 2.04.024 is the Pentium 3, but the Pentium 3 did not exist at the time of the 2.00.000 kernel. The 2.00.x kernels default to the Pentium. The most advanced CPU we could select for the 2.00.x kernels was the Pentium Pro. The Pentium Pro was still supported in the 2.04.x kernels, so we elected to use this CPU for all kernels. In the configuration script, the Pentium Pro is selected by `PPro` in the 2.00.x kernels, by `PPro/6x86MX` in the 2.02.x kernels, and `Pentium-Pro` in the 2.04.x kernels, but all three of these choices set the same configuration variable, `CONFIG_M686`.

C) ISA Bus Support defaults to `"Y"` in 2.04.034, but there is no such option in 2.00.000. The reason for this is that 2.00.000 dates back to 1996, when the ISA bus was the most-widely-used bus in all PCs. The EISA (Extended ISA) bus had been in use for a little over a year (primarily in servers), but it never established itself as a mainstream bus, and was eclipsed by the PCI bus about that time. Support for the ISA bus is considered to be implicit in the 2.00.x kernels. Therefore, when the `make config` dialog for a version of the kernel prompted us for ISA support, we responded with `"Y,"` even though there is no such option in 2.00.000.

D) The 2.04.034 `make config` dialog prompts for PCMCIA support. This feature applies primarily to laptop systems, but in an attempt to follow the method's rules, we investigated the PCMCIA support in 2.00.000. Running `grep` over the 2.00.000 source tree revealed that there *was* a configuration option called `CONFIG_BLK_DEV_IDE_PCMCIA` at that time. Around this time, the only PCMCIA devices available were hard drives, but other special purpose devices quickly followed (primarily modems and networking cards). Given that there clearly is some degree of PCMCIA support in 2.00.000, we responded `“Y”` to prompts about the inclusion of PCMCIA support in all versions to be analyzed, though we did not enable any special PCMCIA features. Similarly, there is an option in 2.04.034 about CardBus support (which was a bus-mastering enhancement of “plain” PCMCIA), and there *are* three places in the code tree containing `CARDBUS` or `CardBus`, but the support does not appear to be complete enough to include this sub-feature of PCMCIA. As such, CardBus support, when prompted for, was NOT included.

E) The 2.04.034 `make config` dialog prompts for the desired `kernel core format`, and the available options are `a.out` and `ELF`, with the latter being the default. In the 2.00.000 configuration dialog, there is a prompt for `Compile kernel as ELF`, which defaults to `“Y.”` `ELF` and `a.out` are two formats for executable programs under Linux. This option would change how the linker might handle compiled object code, but would not have much impact on the source code going into the kernel (perhaps a minor difference in the loading/initializing code). We assume that these two options, even though they have different names, refer to the same concept, and therefore used `ELF` as the preferred method of building all kernels.

F) The 2.04.034 `make config` dialog includes a prompt for `MISC` executable files (as opposed to `a.out` and `ELF`). The documentation for this feature explicitly mentions Java, Python, Emacs-LISP, and any other compiled language that needs an interpreter at runtime. The 2.00.000 configuration includes a prompt for `CONFIG_BINFMT_JAVA`, but it is not clear whether this is the same feature or not. As such, this option was not included in the analysis.

G) The 2.04.034 `make config` dialog contains a prompt for enabling power management. There are two ways of handling power management – according to either the (older) APM (Advanced Power Management) specification or the (newer) ACPI (Advanced Configuration and Power Interface) specification. In the 2.04.034 dialog, both of these default to `"N"` (these are sub-prompts – once `power management` is selected, the next two prompts are for *how* to implement it). Also the 2.04.034 documentation indicates that, when SMP (see item "A" above) is enabled, APM is *disabled*. The 2.00.000 configuration offers only the APM approach, which defaults to `"N."` Given all of these factors, power management was completely disabled in all kernels analyzed, even though the 2.04.034 default is `"Y."`

H) The 2.04.034 `make config` dialog contains two prompts regarding `Plug and Play` (PnP) support (one to enable PnP altogether, and one specifically for PnP on ISA-bus cards). "Plug and Play" really came into its own with Microsoft's Windows 95 operating system. Linux 2.00.000 was released in June of 1996. Because there are numerous instances of `PnP` in the source code tree, and because we have already established that we have implicit support for the ISA bus in

2.00.000, we assumed that ISA PnP support was implicit in 2.00.000 and we therefore enabled PnP (and ISA PnP) whenever prompted.

I) The 2.04.034 make config dialog contains a prompt related to "Packet Socket" support (CONFIG_PACKET). There *is* a constant called SOCK_PACKET in 2.00.000, but the CONFIG_PACKET option appears to be an intra-kernel communication channel, rather than the usual "network-centric" interpretation of packets in 2.00.000. As such, we did not enable this option in any kernel we analyzed.

J) There is an option in the 2.04.034 configuration called CONFIG_UNIX that selects Unix domain sockets. In the entire 2.00.000 code base, the only instances of CONFIG_UNIX are located in net/protocols.c, where it says:

```
#define CONFIG_UNIX   /* always present...    */
```

Farther down in that same file are two lines that say "#ifdef CONFIG_UNIX," so we assume that this was one of those hard-wired-into-2.00.x items that can selectively be *disabled* in 2.04.x, but, because it was "always present" in 2.00.x, we selected "Y" for CONFIG_UNIX in any version that prompts for it.

K) The next configuration item that defaults to "Y" in 2.04.034, but which did not exist in 2.00.000 is CONFIG_IDE. There was no CONFIG_IDE option in 2.00.000; however, CONFIG_BLK_DEV_IDE (configure block devices on IDE) not only existed in 2.00.000, its default value was "Y." Because IDE support clearly existed in 2.00.000, we assumed that this was another of those items that was implicitly built in the 2.00.000 code base, but which was possible to omit in the later versions. We responded "Y" to all instances of CONFIG_IDE in the configuration prompts.

L) In the configuration section on SCSI Disk Support in 2.04.034 is the option Maximum number of SCSI disks that can be loaded a modules, with a default value of 40. The configuration variable for this option is CONFIG_SD_EXTRA_DRVS. In the 2.00.000 code base, the variable SD_EXTRA_DRVS is defined in code (rather than in the configuration script) with a value of 2, so we used 2 as the value for this item in any version that prompts for it.

M) In 2.04.034, there is an option called CONFIG_SCSI_DEBUG_QUEUES, whose description was Enable extra checks in new queueing code. This option defaults to "Y," but the same option does not seem to exist in 2.00.000. There *was* a constant called SCSI_DEBUG_MAILBOXES, with a hard-wired value of 8, but because the variable names are different, and because the description of this variable in 2.04.034 explicitly indicates that this is "new code," we assumed that this is a different feature, and disabled the option in any version that prompts for it.

N) There was is option called `CONFIG_UNIX98_PTYS` that enables support for pseudo consoles (pseudo-ttys, or PTYs), which has a default value of `"Y"` in 2.04.034, but there is no such configuration variable in 2.00.000. It is not clear how *anything* with `UNIX98` could have been included in the 1996 2.00.000 code, but there was code mentioning `pty` in `drivers/char/pty.c` and `drivers/char/tty_io.c`, so this functionality was apparently included long before it became a `UNIX98` standard feature. As such, we responded `"Y"` to enabling `PTY` support, but we did not enable any particular sub-features. Related to this item is the number of `PTY`s to support. The default value of 256, as presented in the 2.04.034 configuration script, agrees with the value defined as a constant in 2.00.000, so whenever prompted for this value, we used 256.

O) The three remaining issues for which the method needs a clarifying decision to be made all involve selecting the particular hardware card with which a particular subsystem should be implemented. In the case of the SCSI, Ethernet Networking, and Sound subsystems, the default cards in 2.04.034 simply did not exist in 2.00.000. There were several cards supporting each of these respective subsystems in 2.00.000, but the method does not provide any way of determining which of the several cards we should select. We elected to make these three decisions:

   a. For the SCSI card, we selected the Adaptec AHA-1542 card. At the time of 2.00.000's release, Adaptec was a market leader in SCSI interface cards, and the AHA-1542 ISA-to-fast-SCSI card was a popular option. At the time of this writing, the card is still being supported by Adaptec, even though it is no longer being manufactured. Support for this card still exists in the 2.04.034 kernel.

b. For the Network card, we selected the Intel EtherExpress Pro. The default card in 2.04.034 is the Intel EtherExpress Pro/100. The EtherExpress Pro (which was available in 2.00.000) was a 10 Mbps ISA bus card, whereas the EtherExpress Pro/100 was a 100 Mbps PCI card. Even though these two cards are clearly different, because they came from the same product family by the same manufacturer, and one was the 2.04.034 default, we elected to use the EtherExpress Pro for systems for which the EtherExpress Pro/100 is not an available option.

c. Similarly, the default sound card in the 2.04.034 kernel is the Creative Ensoniq AudioPCI 97. Unfortunately, this card did not exist in 1996 when 2.00.000 was released (the AC97 [Audio Codec 1997] specification had not been released) at that time. The industry standard sound card, for which emulation has always been the least common denominator, has been the original Creative Labs SoundBlaster SB 16. Support for this card exists in 2.00.000 and 2.04.034, and as such, was our sound card of choice.

In the worst case, if we had selected a card (SCSI, Ethernet, or Sound) that was somehow sub-optimal or non-representative of the rest of the class, then the net effect on the code base should be negligible. For example, the body of code required to support the particular card used to handle networking surely pales in comparison to the body of code that implements the higher-level aspects of Ethernet – protocols, routing, packet construction / decoding, sockets, and so on.

## 5.4 The Configuration Script – `make config`

When the user enters the `make config` command, the first configuration question to appear is:

```
*
* Code maturity level options
*
Prompt  for  development  and/or  incomplete  code/drivers  (CONFIG_EXPERIMENTAL)
[N/y/?] n
```

This prompt illustrates the dialog mechanism. Each prompt has several components:

- Lines beginning with an asterisk (`*`) are treated by the configuration script processor as comments. Such lines are echoed to the user, but do not constitute prompts per se. Most comment lines are used to delineate sections of the configuration script.
- The text of the prompt (`"Prompt for development and/or incomplete code/drivers"`) gives a brief description of this configuration option.
- The value in parentheses (`"CONFIG_EXPERIMENTAL"`) is the configuration variable being set by this question. Within the source code, conditional compilation statements (such as `#ifdef CONFIG_EXPERIMENTAL`) will

include or exclude various portions of source code. These sections range in size from only part of a statement all the way to entire hierarchies of nested `#include` files.

- The values in square brackets show the allowable responses to this prompt. In this case, `“N,” “y,”` and `“?”` appear. The value in upper-case is the default response (which is selected by pressing `ENTER)`. In this case, pressing `ENTER` has the same effect as responding with `“N,”` and the remainder of the configuration script will not prompt the user for the inclusion of any experimental or incomplete code. The vast majority of the configuration selections involve yes/no prompts, but a few are of the select-from-a-list type, as in the processor selection (to which we have responded `“PPro”`):

```
Processor type (386, 486, Pentium, PPro) [Pentium] PPro
```

- Responding to the prompt with “?” displays an extended help message for each configuration option. The full text of the extended help messages is located in the file `/Documentation/Configure.help`. These extended help messages tend to be quite complete. For example, the extended help message for the `code maturity level options` item is:

```
“Some of the various things that Linux supports
(such as network drivers, filesystems, network
protocols, etc.) can be in a state of development
where the functionality, stability, or the level
of testing is not yet high enough for general
use. This is usually known as the "alpha-test"
phase amongst developers. If a feature is
```

> currently in alpha-test, then the developers usually discourage uninformed widespread use of this feature by the general public to avoid "Why doesn't this work?" type mail messages. However, active testing and use of these systems is welcomed. Just be aware that it may not meet the normal level of reliability or it may fail to work in some special cases. Detailed bug reports from people familiar with the kernel internals are usually welcomed by the developers. Unless you intend to help test and develop a feature or driver that falls into this category, or you have a situation that requires using these features you should probably say N here, which will cause this configure script to present you with fewer choices. If you say Y here, you will be offered the choice of using features or drivers that are currently considered to be in the alpha-test phase.”

- The final item on this line is our response, “n.” Although the default response was “N,” we have explicitly responded to every configuration prompt, even when our responses coincided with the default.

The first several prompts in the dialog for 2.00.0 kernel are shown in Figure 5.1, along with our responses.

```
*
* Code maturity level options
*
Prompt for development and/or incomplete code/drivers
(CONFIG_EXPERIMENTAL) [N/y/?] n
*
* Loadable module support
*
Enable loadable module support (CONFIG_MODULES) [Y/n/?] n
*
* General setup
*
Kernel math emulation (CONFIG_MATH_EMULATION) [N/y/?] n
Networking support (CONFIG_NET) [Y/n/?] y
Limit memory to low 16MB (CONFIG_MAX_16M) [N/y/?] n
PCI bios support (CONFIG_PCI) [Y/n/?] y
System V IPC (CONFIG_SYSVIPC) [Y/n/?] y
Kernel support for a.out binaries (CONFIG_BINFMT_AOUT) [Y/n/?] y
Kernel support for ELF binaries (CONFIG_BINFMT_ELF) [Y/n/?] y
Compile kernel as ELF - if your GCC is ELF-GCC (CONFIG_KERNEL_ELF)
[Y/n/?] y
Processor type (386, 486, Pentium, PPro) [Pentium] PPro
  defined CONFIG_M686
*
* Floppy, IDE, and other block devices
*
Normal floppy disk support (CONFIG_BLK_DEV_FD) [Y/n/?] y
Enhanced IDE/MFM/RLL disk/cdrom/tape support (CONFIG_BLK_DEV_IDE)
[Y/n/?] y
```

**Figure 5.1 – Excerpt from the beginning of the configuration script for the 2.00.000 kernel.**

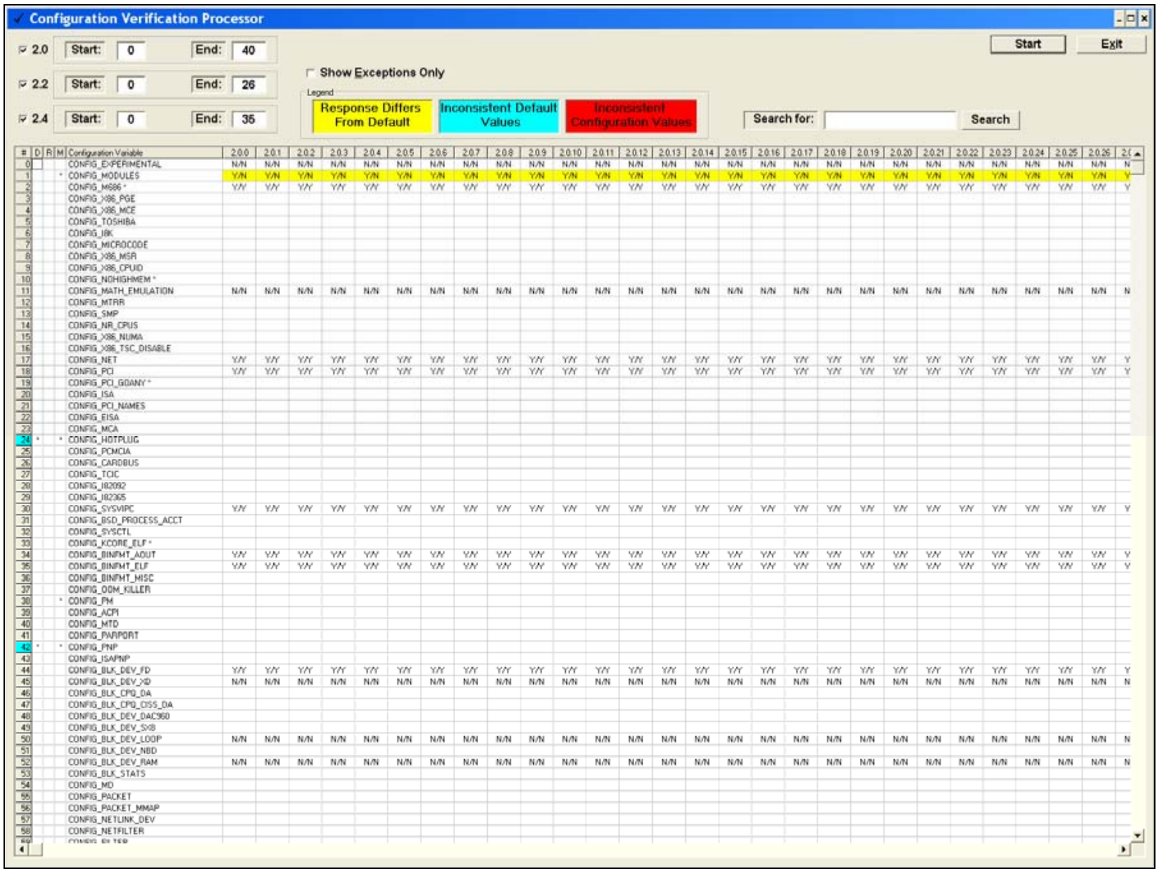
There are approximately 150 configuration items in our 2.00.000 configuration script, and approximately 425 in 2.04.035. There are potentially more configuration options in the entire configuration script, but certain responses prune the list of prompts presented to the user. For example, if the user responds "N" to the prompt for Networking Support, then the subsequent prompts for network related options (such as protocols, services, and network interface cards) are not presented.

## 5.5 The One Major Exception to Our Method

The only major configuration item, which defaults to “Y” in all versions from 2.00.000 through 2.04.034, and for which we did not select the default, was for Loadable Modules. Beginning with the 2.00.000 kernel (introduced in the 1.03.x development cycle), Linux supported kernel-loadable modules, or KLMs. Kernel-loadable modules allow Linux to dynamically load and unload modules as needed. This allows the kernel to have a smaller memory footprint, and can be particularly advantageous in memory-limited systems. Because we were not concerned with memory usage, and because we did not want to examine multiple instances of the module loading/unloading code, we elected to not enable support for KLMs.

## 5.6 Verifying Our Configuration as Selected

After running through the configuration script for each version, we collected the screen output and saved it in a text file. We created a tool, called `Configuration Checker`, to read these scripts and compare our responses from version to version in order to make sure we had responded to all of the configuration options as we had intended. With hundreds of configuration options in each version, it is easy to get one incorrect. There are 588 distinct configuration options between all of the kernel versions. The `Configuration Checker` flags instances where our responses were inconsistent from version to version, where the default value changes from version to version, and where our response differs from the default. A screen shot from the Configuration Checker is shown in Figure 5.2.



Configuration Verification Processor

☑ 2.0 Start: 0 End: 40

☑ 2.2 Start: 0 End: 26

☑ 2.4 Start: 0 End: 35

□ Show Exceptions Only

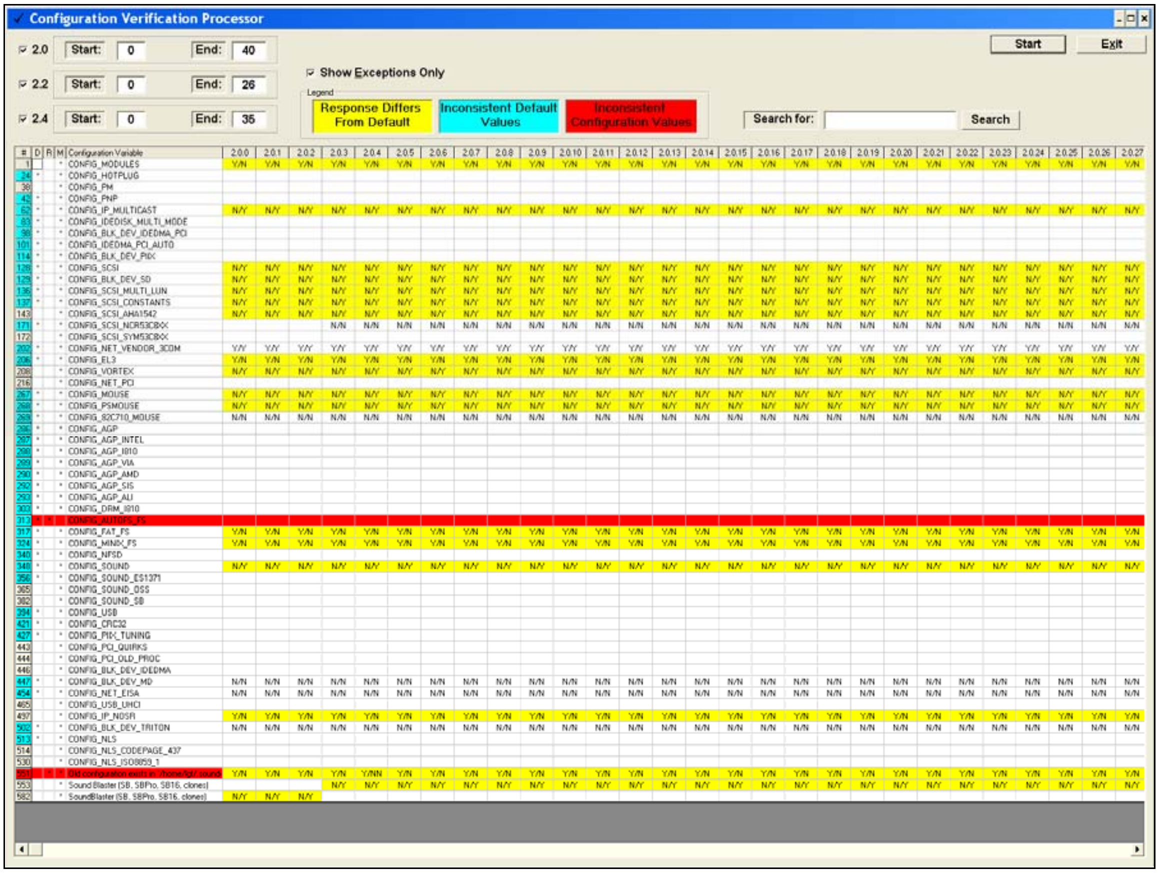
Legend: 🟨 Response Differs From Default | 🟦 Inconsistent Default Values | 🟥 Inconsistent Configuration Values

Search for: [ ] Search

| # | D | R | M | Configuration Variable | 2.0.0 | 2.0.1 | 2.0.2 | 2.0.3 | 2.0.4 | 2.0.5 | 2.0.6 | 2.0.7 | 2.0.8 | 2.0.9 | 2.0.10 | 2.0.11 | 2.0.12 | 2.0.13 | 2.0.14 | 2.0.15 | 2.0.16 | 2.0.17 | 2.0.18 | 2.0.19 | 2.0.20 | 2.0.21 | 2.0.22 | 2.0.23 | 2.0.24 | 2.0.25 | 2.0.26 | 2.( |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | | | CONFIG_EXPERIMENTAL | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N |
| 1 | | | * | CONFIG_MODULES | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y |
| 2 | | | | CONFIG_M686 * | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y |
| 3 | | | | CONFIG_X86_PGE | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 4 | | | | CONFIG_X86_MCE | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 5 | | | | CONFIG_TOSHIBA | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 6 | | | | CONFIG_I8K | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 7 | | | | CONFIG_MICROCODE | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 8 | | | | CONFIG_X86_MSR | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 9 | | | | CONFIG_X86_CPUID | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 10 | | | | CONFIG_NOHIGHMEM * | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 11 | | | | CONFIG_MATH_EMULATION | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N |
| 12 | | | | CONFIG_MTRR | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 13 | | | | CONFIG_SMP | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 14 | | | | CONFIG_NR_CPUS | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 15 | | | | CONFIG_X86_NUMA | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 16 | | | | CONFIG_X86_TSC_DISABLE | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 17 | | | | CONFIG_NET | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y |
| 18 | | | | CONFIG_PCI | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y |
| 19 | | | | CONFIG_PCI_GOANY * | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 20 | | | | CONFIG_ISA | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 21 | | | | CONFIG_PCI_NAMES | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 22 | | | | CONFIG_EISA | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 23 | | | | CONFIG_MCA | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 24 | * | | * | CONFIG_HOTPLUG | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 25 | | | | CONFIG_PCMCIA | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 26 | | | | CONFIG_CARDBUS | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 27 | | | | CONFIG_TCIC | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 28 | | | | CONFIG_I82092 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 29 | | | | CONFIG_I82365 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 30 | | | | CONFIG_SYSVIPC | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y |
| 31 | | | | CONFIG_BSD_PROCESS_ACCT | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 32 | | | | CONFIG_SYSCTL | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 33 | | | | CONFIG_KCORE_ELF * | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 34 | | | | CONFIG_BINFMT_AOUT | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y |
| 35 | | | | CONFIG_BINFMT_ELF | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y |
| 36 | | | | CONFIG_BINFMT_MISC | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 37 | | | | CONFIG_OOM_KILLER | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 38 | | | * | CONFIG_PM | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 39 | | | | CONFIG_ACPI | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 40 | | | | CONFIG_MTD | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 41 | | | | CONFIG_PARPORT | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 42 | * | | * | CONFIG_PNP | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 43 | | | | CONFIG_ISAPNP | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 44 | | | | CONFIG_BLK_DEV_FD | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y |
| 45 | | | | CONFIG_BLK_DEV_XD | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N |
| 46 | | | | CONFIG_BLK_CPQ_DA | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 47 | | | | CONFIG_BLK_CPQ_CISS_DA | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 48 | | | | CONFIG_BLK_DEV_DAC960 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 49 | | | | CONFIG_BLK_DEV_SX8 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 50 | | | | CONFIG_BLK_DEV_LOOP | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N |
| 51 | | | | CONFIG_BLK_DEV_NBD | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 52 | | | | CONFIG_BLK_DEV_RAM | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N |
| 53 | | | | CONFIG_BLK_STATS | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 54 | | | | CONFIG_MD | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 55 | | | | CONFIG_PACKET | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 56 | | | | CONFIG_PACKET_MMAP | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 57 | | | | CONFIG_NETLINK_DEV | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 58 | | | | CONFIG_NETFILTER | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 59 | | | | CONFIG_FILTER | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

**Figure 5.2 – Screenshot of `Configuration Checker`.**

The `Configuration Checker` can also filter the response sequences, and display only the exceptions from a range of kernel versions. A screen shot of all of the exceptions from the 2.00.000 through 2.00.027 kernels is shown in Figure 5.3. Scrolling to the right in `Configuration Checker` would reveal the subsequent versions, through 2.04.035.



✓ Configuration Verification Processor
- □ ×
Start
Exit
Search for:
Search
Legend
🟨 Response Differs From Default
🟦 Inconsistent Default Values
🟥 Inconsistent Configuration Values

☑ 2.0 Start: 0 End: 40

☑ 2.2 Start: 0 End: 26

☑ 2.4 Start: 0 End: 35

☑ Show Exceptions Only

| # | D | R | M | Configuration Variable | 2.0.0 | 2.0.1 | 2.0.2 | 2.0.3 | 2.0.4 | 2.0.5 | 2.0.6 | 2.0.7 | 2.0.8 | 2.0.9 | 2.0.10 | 2.0.11 | 2.0.12 | 2.0.13 | 2.0.14 | 2.0.15 | 2.0.16 | 2.0.17 | 2.0.18 | 2.0.19 | 2.0.20 | 2.0.21 | 2.0.22 | 2.0.23 | 2.0.24 | 2.0.25 | 2.0.26 | 2.0.27 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | □ |  | * | CONFIG_MODULES | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N |
| 24 | * |  | * | CONFIG_HOTPLUG |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 38 |  |  | * | CONFIG_PM |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 42 | * |  | * | CONFIG_PNP |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 62 | * |  | * | CONFIG_IP_MULTICAST | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y |
| 83 | * |  | * | CONFIG_IDEDISK_MULTI_MODE |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 98 | * |  | * | CONFIG_BLK_DEV_IDEDMA_PCI |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 101 | * |  | * | CONFIG_IDEDMA_PCI_AUTO |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 114 | * |  | * | CONFIG_BLK_DEV_PIIX |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 128 | * |  | * | CONFIG_SCSI | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y |
| 129 | * |  | * | CONFIG_BLK_DEV_SD | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y |
| 136 | * |  | * | CONFIG_SCSI_MULTI_LUN | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y |
| 137 | * |  | * | CONFIG_SCSI_CONSTANTS | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y |
| 143 |  |  | * | CONFIG_SCSI_AHA1542 | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y |
| 171 | * |  | * | CONFIG_SCSI_NCR53C8XX |  |  |  | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N |
| 172 |  |  | * | CONFIG_SCSI_SYM53C8XX |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 202 | * |  | * | CONFIG_NET_VENDOR_3COM | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y | Y/Y |
| 206 | * |  | * | CONFIG_EL3 | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N |
| 208 |  |  | * | CONFIG_VORTEX | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y |
| 216 |  |  | * | CONFIG_NET_PCI |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 267 | * |  | * | CONFIG_MOUSE | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y |
| 268 | * |  | * | CONFIG_PSMOUSE | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y |
| 269 | * |  | * | CONFIG_82C710_MOUSE | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N |
| 286 | * |  | * | CONFIG_AGP |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 287 | * |  | * | CONFIG_AGP_INTEL |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 288 | * |  | * | CONFIG_AGP_I810 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 289 | * |  | * | CONFIG_AGP_VIA |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 290 | * |  | * | CONFIG_AGP_AMD |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 292 | * |  | * | CONFIG_AGP_SIS |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 293 | * |  | * | CONFIG_AGP_ALI |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 303 | * |  | * | CONFIG_DRM_I810 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 313 | * | * |  | CONFIG_AUTOFS_FS |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 317 | * |  | * | CONFIG_FAT_FS | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N |
| 324 | * |  | * | CONFIG_MINIX_FS | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N |
| 340 | * |  | * | CONFIG_NFSD |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 348 | * |  | * | CONFIG_SOUND | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y |
| 356 | * |  | * | CONFIG_SOUND_ES1371 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 365 |  |  | * | CONFIG_SOUND_OSS |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 382 |  |  | * | CONFIG_SOUND_SB |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 394 | * |  | * | CONFIG_USB |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 421 | * |  | * | CONFIG_CRC32 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 427 | * |  | * | CONFIG_PIIX_TUNING |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 443 |  |  | * | CONFIG_PCI_QUIRKS |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 444 |  |  | * | CONFIG_PCI_OLD_PROC |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 446 |  |  | * | CONFIG_BLK_DEV_IDEDMA |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 447 | * |  | * | CONFIG_BLK_DEV_MD | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N |
| 454 | * |  | * | CONFIG_NET_EISA | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N |
| 465 |  |  | * | CONFIG_USB_UHCI |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 497 |  |  | * | CONFIG_IP_NOSR | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N |
| 502 | * |  | * | CONFIG_BLK_DEV_TRITON | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N | N/N |
| 513 | * |  | * | CONFIG_NLS |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 514 |  |  | * | CONFIG_NLS_CODEPAGE_437 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 530 |  |  | * | CONFIG_NLS_ISO8859_1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 551 |  | * | * | Old configuration exists in '/home/lgt/.sound | Y/N | Y/N | Y/N | Y/N | Y/NN | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N | Y/N |
| 553 |  |  | * | Sound Blaster (SB, SBPro, SB16, clones) |  |  |  | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y | N/Y |
| 582 |  |  | * | SoundBlaster (SB, SBPro, SB16, clones) | N/Y | N/Y | N/Y |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

**Figure 5.3 – Screenshot of `Configuration Checker`, showing only exceptions**

There are two items for which the `Configuration Checker` indicates we have made inconsistent configuration selections (those highlighted in red):

1) One item is related to saving the old sound card configurations before applying the new ones. Because we *always* specified new sound card configuration settings, it did not matter if we saved the old settings or not. Consequently, we sometimes responded with “Y” and sometimes with “N,” which `Configuration Checker` flagged as inconsistent configuration settings. In this instance, however, the inconsistency had absolutely no impact on our code base.

2) The other item is related to a feature that we had to select in order to keep the configuration as consistent as possible. Versions 2.04.x include (by default) “`Kernel automounter version 4 support (also supports v3)`.” Because this was the default in 2.04.034, and because kernel automounter support was also available as far back as version 2.02.000, we selected it. But the Version 4 support began in 2.04.000. In 2.02.x, the only automounter support available was Version 3. Even though the Version 3 support was still available in 2.04.x, the method’s rules called for us to select the Version 4 support in versions that provided it, and Version 3 in the versions that did not. The `Configuration Checker` sees this as an inconsistency, whereas it actually makes the configurations more consistent.

Once the kernel configuration has been established, the kernel can then be built (compiled and linked). The details of the process of building the Linux kernels and extracting the source code metrics are presented in Appendix A.

# CHAPTER VI

# GLOBAL VARIABLES: A NEW CATEGORIZATION AND ITS CASE TOOL

## 6.1 How “Global” is “Global”?

The term “global variable” is somewhat incomplete in and of itself. For CXREF to identify a variable as being “global,” the variable must be defined at a global level; that is, outside of any function. However, in the absence of a declaration of that variable with the “extern” attribute in another file, such a variable is visible only within the module (file) in which it is defined. One might consider such a variable to be “global within the file in which it is declared,” but not truly a “global variable” (one visible in more than one file). The CXREF output excerpt in Figure 6.1 shows a FILE line and a VARIABLE block. This particular variable, dma_chan_busy, *is* defined in dma.c at the global level (i.e., outside of any function); however, there is no extern attribute elsewhere in this version of Linux, so this variable is local to the module in which it is defined, as evidenced by the “[Local]” attribute on the “VARIABLE” line (as well as the fact that this variable is not listed as being visible or used in any other file).

```
FILE : 'kernel/dma.c'

VARIABLE : dma_chan_busy [Local]
Defined: kernel/dma.c:611
Type: volatile unsigned int dmahan_busy[8]
Used in free_dma : kernel/dma.c
Used in request_dma : kernel/dma.c
```

**Figure 6.1 – Excerpt from CXREF output.**

A distinction should also be made between the *declaration* of a variable and the *definition* of a variable. A variable is *defined* when the compiler allocates storage for it, as in `int x`; conversely, a variable is *declared* when the compiler is made aware of a variable's existence, as in `extern int x`. This is a fine but important distinction. A variable defined in one file (at the global level), and never declared from another file, is *local*, for the purposes of this dissertation. A variable defined at the global level in two files (e.g., `int x;` in each of two files) actually allocates two distinct integer variables called `x`. The instance of `x` in one file is completely independent of the instance of `x` in the other file. Although this might be considered poor programming practice, this does not create a *truly* global variable `x` (and there would be no coupling in this case, either).

## 6.2 Classifying the GV Instances – Instance Kinds

Our original intent was to perform a def/use analysis on the GVs, but on examining the code, we quickly realized that the usage patterns did not cleanly fall into Used and Defs. As discussed in Section 2.4, the authors of [Feitelson et al., 2007] used a nine-way classification scheme for GV accesses. We made the following modifications to their classifications for our analysis:

1. We did not treat combined definitions and uses as two operations. For example, `gv++` was considered by Feitelson et al. as one occurrence of a GV, but also as both a def *and* a use. We believed that every occurrence of a GV should count as exactly one kind of occurrence. We therefore treated `gv++` as a (single) assignment occurrence.

2. We did not differentiate between passing GVs by value and passing them by reference. There is no way to determine whether the function to which the GV is passed will actually modify the GV, even if it is passed by reference. Moreover, some GVs are *already* pointers. We therefore elected to merge these two occurrence kinds, and simply count the number of instances in which a GV was passed as a parameter, without regard as to *how* it was passed.

3. In keeping with the desire to have a single occurrence of a GV count as exactly one kind of occurrence, we simplified the authors' pointer dereferencing definition. Instances of the form `gv->member1->member2` count as two instances of the "pointer dereference" kind according to Feitelson et al, but we counted such occurrences as one "read" of the GV, because the value of the GV is read in order to perform the pointer dereference. Because pointer dereferencing was considered to be a simple read of the GV, we did not keep a separate count for this kind of occurrence.

4. We expanded Feitelson's "sizeof" kind to include *all* compile-time references to a GV. In addition to `sizeof(gv),` the kernel code also contains `typeof(gv)` references. Perhaps the most frequently occurring compile-time reference to GVs occurs in the `__builtin_constant_p()` pseudo-function (see Kind 1, below).

5. Feitelson et al. did not mention if and how they count instances of GVs used as parameters to assembly-language calls. As we will see in Section 6.4, there are very few such instances in the 2.04.020 version of the kernel that Feitelson et al. examined, but in the 2.00.x series, and even more so in the 2.02.x series, GVs are passed to assembly language routines quite frequently. Analyzing each assembly-language fragment to determine whether the GVs were read or written was beyond our scope. We therefore simply counted the number of instances in which a GV was used as a parameter to an assembly language call.

Given these departures from Feitelson et al.'s classifications, we classified all GV instances into nine categories, which we called "Kinds":

Kind 0: Not actually an instance of the GV. This kind includes occasions where a GV appears in a quoted string, and other such instances where the sequence of characters that happens to coincide with the *name* of a GV is not actually an *instance* of the GV in code. All Kind-0 instances were discarded and not even counted.

Kind 1: Compile-time reference to the GV. These references refer to some attribute of the GV besides its address or its value, and are evaluated at compile time, and therefore do not impact coupling. Such instances might be `typeof(gv)` and `sizeof(gv)`. One of the more frequently occurring examples of Kind-1 instances is the GNU construct `__builtin_constant_p()`. This special "function" is a `gcc` extension that is evaluated at compile-time, and returns `true` if its argument is a constant. It is used extensively for hardware I/O where the port number is known in advance (i.e., it is a constant), as opposed to being contained in a variable. The code is rife with instances of the form:

```
__builtin_constant_p(expression)? handle one way: handle another;
```

By handling the operation two different ways, the performance can be optimized. This is a way to manually optimize the code at the C level, so that the compiler may do an even better job of doing so at the level of the generated code.

Kind 2: Assignment. These are instances in which the contents of the GV are definitely modified. Kind-2 examples include using the GV on the left side of a single equals sign, operating on the GV with pre- and post- increment and decrement operators (`++`/`--`), and the compound operators such as `+=`, `-=`, `*=`, `/=`, `&=`, `^=`, and `%=`. These instances are what we call destructive references to the GV, as they categorically destroy the existing value of the GV. In some cases, it is not possible to categorically determine whether an instance in question reads the value of the GV (is a "use") or modifies the value of the GV (is a "def"). However, Kind-2 instances are "definitely defs."

Kind 3: Atomic operation. A key component of any operating system's code is interprocess synchronization, and Linux is no exception. There are several routines provided to implement semaphores and other uninterruptible constructs. As such, these operations are more strictly controlled, and, by their very nature, less prone to cause problems when associated with global variables. There are a number of functions that are passed global variables to perform atomic operations upon them. These include `atomic_add`, `atomic_inc`, `atomic_dec`, `atomic_test_and_dec`, and so on. Whenever a global variable is passed to any of these functions (or, more precisely, the address of the GV is passed), we classify such instances as Kind 3.

Kind 4: Assembler code. Linux implements its low-level hardware interface with assembly-language calls using the `__asm__ gcc` extension. This allows platform-specific interface to the hardware, but it also provides exceedingly efficient implementations of some low-level routines, thanks to hand-optimization of the assembler code. The input and output parameters of the assembler code are often GVs. It is, in general, not possible to determine from static analysis whether the value of a GV passed to an assembly-language call will be modified or not. In an attempt to at least quantify the relative frequency with which GVs are used as parameters to assembly language calls, we count such instances as Kind 4.

Kind 5: Address-of operation. Often, the code refers to the address of a GV and then does something with that address, rather than with the value of the GV. A follow-up analysis of what subsequently happens is beyond the scope of this dissertation.

Kind 6: Passed as parameter. It seems somewhat counterintuitive to pass a GV as a parameter to a function. After all, if a variable is global, why should we pass it at all? Surely the function to which we are passing the GV can *already* see the GV? The answer lies in how the function processes the GV and how we have interpreted GVs.

First, a function may operate on constants, private variables, or global variables. Just because we happen to pass a GV to a function in one instance does not mean that *all* calls to that function (perhaps in other files and other contexts) would have passed the same GV.

Second, and perhaps more importantly, we have not performed a fine-grained analysis of the GV usage. If a GV happens to be an array, a function call may pass only one element of the array (and that element may be determined only at run-time, thereby requiring a dynamic analysis of the function).

Moreover, passing a GV to a function is normally handled by value. If we wish to allow the called function to modify the GV, we could pass it by reference, using `&gv`. Even if we pass a GV by reference, there is, in general, no way to determine via static analysis whether or not the function actually *will* modify the GV. As such, we cannot count a GV passed by reference as a "definite def."

Because the atomic operations discussed above in Kind 3 have special controls in place to prevent race conditions, passing a GV to one of the functions that implements an atomic operation is treated differently, even though both are instances of passing a GV to a function. Because the atomic functions are "special" functions, we count those separately.

Kind 7: Execution. Sometimes a GV is used as a pointer to a function, or as a pointer to a table of functions, sometimes called a dispatch vector or a jump table. In such cases, the GV is "executed" when we use its contents to point to executable code, rather than to data. This sort of access is different enough from other forms of GV access for us to count such an instance separately to keep the instances segregated.

Kind 8: Other. The Kind-8 references include all other references to a GV, including using the GV in an expression, or on the right side of a single equals sign. Just as Kind-2 instances are "definitely defs," Kind-8 instances are "definitely uses." These instances are definitely reads of the GV, and hence non-destructive references to the GV's value.

## 6.3 Performing the GV Instance Classification

Given these rather straightforward Kinds of GV instances, we then needed a mechanism to scour the source code and make the categorizations, keeping the Kind counts on a per-function basis. It was then straightforward to roll up the per-function instances into file-level and kernel-level instances counts.

Because we were clearly outside the bounds of the functionality available from any known CASE tool, we had to develop our own, and thus was born the `CodeSweeper` application. This program scans through the entire source code base for a given version of the kernel, seeking to automatically identify and classify as many GV instances as possible, and then relying on manual intervention to resolve the remaining instances it could not parse automatically.

The automatic parser looks for particular code constructs involving GVs. These constructs are called "classes." Because the program was designed to identify and *classify* GV instances, we adopted the perhaps unfortunate name of "class" to describe patterns in the code ("a particular class of access" as opposed to "class" in the object-oriented sense). We defined a total of 36 distinct classes for which we could parse, and `CodeSweeper` searches for them in this order:

| Class # | Description |
|---|---|
| 1 | `__builtin_constant_p(gv)`. If a GV occurs between the parentheses delimiting the parameter list for `__builtin_constant_p()`, it is considered a Class-1 instance. All Class-1 GV instances are of Kind 1 (Compile-time). |
| 2 | `--gv` or `++gv`. The preincrement and predecrement operators definitely modify the value of the GV, and as such, all Class-2 instances are of Kind 2 (Assignment). |
| 3 | `gv--` or `gv++`. The postincrement and postdecrement operators definitely modify the value of the GV, and as such, all Class-3 instances are of Kind 2 (Assignment). |
| 4 | `--(gv)` and `++(gv)`. In many places, expressions are evaluated as part of a macro expansion, and in order to prevent ambiguities or unintended side-effects, such expansions are placed inside parentheses. The preincrement and predecrement operators, operating on a GV immediately enclosed in parentheses, are Class 4 instances. All Class-4 instances are of Kind 2 (Assignment). |
| 5 | `(gv)--` and `(gv)++`. Same as Class 4. All Class-5 instances are of Kind 2. |
| 6 | Not currently used. |

| | |
|---|---|
| 7 | `gv=` at the start of a statement. This class encompasses occurrences where the GV is the first thing on a line, or if not the first thing on the line, immediately following a semicolon. All Class-7 instances are of Kind 2 (Assignment). |
| 8 | `gv<operator>` at the start of a statement, where `<operator>` is one of `+=`, `-=`, `*=`, `/=`, `%=`, `^=`, `&=`, or `\|=`. All Class-8 instances are of Kind 2 (Assignment). |
| 9 | `gv<operator>` at the start of a statement, where `<operator>` is one of `<<=` or `>>=`. All Class-9 instances are of Kind 2 (Assignment). |
| 10 | `gv->` When a GV is used as a pointer to a structure to refer to a member of the structure, the value of the GV is being read for the sole purpose of locating the member, even if the remainder of this statement modifies the member. Therefore, all Class-10 instances are of Kind 8 (Other). |
| 11 – 13 | Not currently used. |
| 14 | GV names appearing inside quoted strings. All Class-14 instances are of Kind 0 (Not actually a GV instance). |
| 15 | `(0+current_set[smp_processor_id()])` This is the macro expansion for "`current`," which returns the currently executing task. Because `current_set` is a GV, we search explicitly for this often-used construct. Because `current_set` is part of an arithmetic expression, all Class-15 instances are of Kind 8 (Other). |
| 16 | `sizeof(gv)` or `sizeof(struct gv)`. These are references to an attribute of the GV other than its address or contents, and as such, all Class-16 instances are of Kind 1 (Compile-time). |
| 17 | `(gv<2-character operator>< identifier>`, where the operator is one of `==`, `!=`, `&&`, `<=`, `>=`, or `\|\|`. In order to determine whether an identifier follows the operator, we look for an open-parenthesis, and then the GV, immediately followed by one of the two-character operators, immediately followed by any of the characters that are valid in the name or value of an identifier – a letter, digit, or underscore. All such occurrences of this pattern are Class-17 instances, and are of Kind 8 (Other). |
| 18 | `(gv<1-character operator>< identifier>`, where the operator is one of `+`, `-`, `*`, `/`, `<`, `>`, `&`, `\|`, or `%`. The characters signifying the presence of an `<identifier>` are as in Class 17. All Class-18 instances are of Kind 8 (Other). |
| 19 | `<identifier><2-character operator>GV)`. This is the symmetric version of Class 17, where the GV and the closing parenthesis end the expression. Classes 17 and 19 search for the same list of operators. All Class-19 instances are of Kind 8 (Other). |
| 20 | `<identifier><1-character operator>gv)`. This is the symmetric version of Class 18, where the GV and the closing parenthesis end the expression. Classes 18 and 20 search for the same list of operators. All Class-20 instances are of Kind 8 (Other). |
| 21 | `[gv]`. When a GV is used as an array subscript, the value of the GV is read, making all Class-21 instances of Kind 8. |

| | |
|---|---|
| 22 | `gv(`. When a GV immediately precedes an open parenthesis, the GV is either the name of a function, or holds a pointer to a function. All Class-22 instances are of Kind 7 (Execute). |
| 23 | `if (GV)`, `if (!GV)`, `while(GV)`, or `while(!GV)`. These conditional expressions all read the value of the GV, and, as such, all Class-23 instances are of Kind 8 (Other). |
| 24 | `<identifier><2-character operator>gv;` This case is similar to Class 19, except that rather than coming at the end of an expression that ends with a closing parenthesis, the expression in Class 24 ends with a semicolon (most likely, the end of a statement, although it could be the end of a clause in a `for` statement). As with Class 19, all Class-24 instances are of Kind 8 (Other). |
| 25 | `<identifier><1-character operator>gv;` This case is similar to Class 20, except that rather than coming at the end of an expression that ends with a closing parenthesis, the expression in Class 25 ends with a semicolon (most likely, the end of a statement, although it could be the end of a clause in a `for` statement). As with Class 20, all Class-25 instances are of Kind 8 (Other). |
| 26 | `;gv<2-character operator><identifier>` This case is similar to Class 17, except that the expression follows a semicolon. This catches instances of GVs in `for` statements, such as `for (i=1;gv>=i+1;i++)`. All Class-26 instances are of Kind 8 (Other). |
| 27 | `;gv<1-character operator><identifier>` This case is similar to Class 18, except that the expression follows a semicolon. This catches instances of GVs in `for` statements, such as `for (i=1;gv>i+1;i++)`. All Class-27 instances are of Kind 8 (Other). |
| 28 | Class 28 represents GVs passed to any of several "atomic" functions as mentioned in the discussion on Kind 3 in Section 7.4.13.1. The function names we search for, in order of decreasing function name length, are:<br>`__down_interruptable`<br>`atomic_sub_and_test`<br>`atomic_dec_and_test`<br>`atomic_inc_and_test`<br>`atomic_add_negative`<br>`down_interruptable`<br>`init_MUTEX_LOCKED`<br>`__down_trylock`<br>`write_trylock`<br>`down_trylock`<br>`spin_trylock`<br>`spin_unlock`<br>`init_MUTEX`<br>`init_rwsem`<br>`down_write`<br>`__up_write`<br>`atomic_add`<br>`atomic_inc`<br>`atomic_dec`<br>`atomic_sub`<br>`write_lock`<br>`down_read`<br>`sema_init`<br>`__up_read`<br>`read_lock`<br>`spin_lock`<br>`up_write`<br>`up_read`<br>`__down`<br>`__up`<br>`down`<br>`up`<br>All Class-28 instances are of Kind 3 (Atomic) |
| 29 | `=gv` and `=(gv)`. Because we have already checked for various other combinations that can have a GV immediately following an equals sign, such occurrences found at this point must be a GV on the right side of an assignment statement. As such, all Class-29 instances are of Kind 8 (Other). |

| | |
|---|---|
| 30 | =(&GV); or =&(GV); Assigning the address of a GV to something clearly uses the *address* of the GV, rather than its *contents*. As such, all Class-30 instances are of Kind 5 (Address-of). |
| 31 | The GV __strtok is used in assembly language calls to split strings into tokens. It appears as both an input and output parameter. We look specifically for either of the strings "=S"(__strtok) or "Q"(__strtok). Either of these is obviously of Kind 4 (Assembler). |
| 32 | Return statements. Any of:<br>return gv;<br>return(gv);<br>return proc_register(gv<br>return(proc_register(gv<br>return(proc_unregister(gv<br>return proc_unregister(gv<br>Any instances of the first two types are of Kind 8 (Other).<br>Any instances of the last four types are of Kind 6 (Passed as parameter). |
| 33 | GV passed to one of the bit manipulation functions change_bit, clear_bit, test_bit, or set_bit. These are not atomic operations (see Class 28), and as such, they are simply of Kind 6 (Passed as parameter). |
| 34 | Not used at this time. |
| 35 | &&(gv)< This pattern appears with relative frequency and is part of a compound conditional expression. The value of the GV is being compared against something else, and as such, all Class-35 instances are of Kind 8 (Other). |
| 36 | (gv)-> and (&gv)->. These both use the contents of the GV to look up a member of a structure. As such, all Class-36 instances are of Kind 8 (Other). |

We must also point out that when we refer to "GV," we mean not only "simple" variables such as x86_mask, but also "extended" variable names. Variables can be the names of structures, arrays, structures of arrays, arrays of structures, and so on. There is a GV called x86_mask, but the following would also be "extended" versions of the same variable name:

```
x86_mask[1]
x86_mask[1].member
x86_mask.member
x86_mask.member[1][3].member2[function(x)]
```

Put another way, the base GV name can be followed by zero or more occurrences of [subscript] and/or .member. When CodeSweeper locates a GV, it searches forward, identifying the fully-extended GV name. Class 15 in Table 6.1 is an example of using the extended GV name.

Because it is possible for other GV names to appear within the subscripts of other GV names, the CodeSweeper resumes searching for GV names starting with the first character after the base GV name. Therefore, in the following source line, the program would correctly identify that GV starts in column 1, with a length of 28, and that GV2 starts in column 17, with a length of 3:

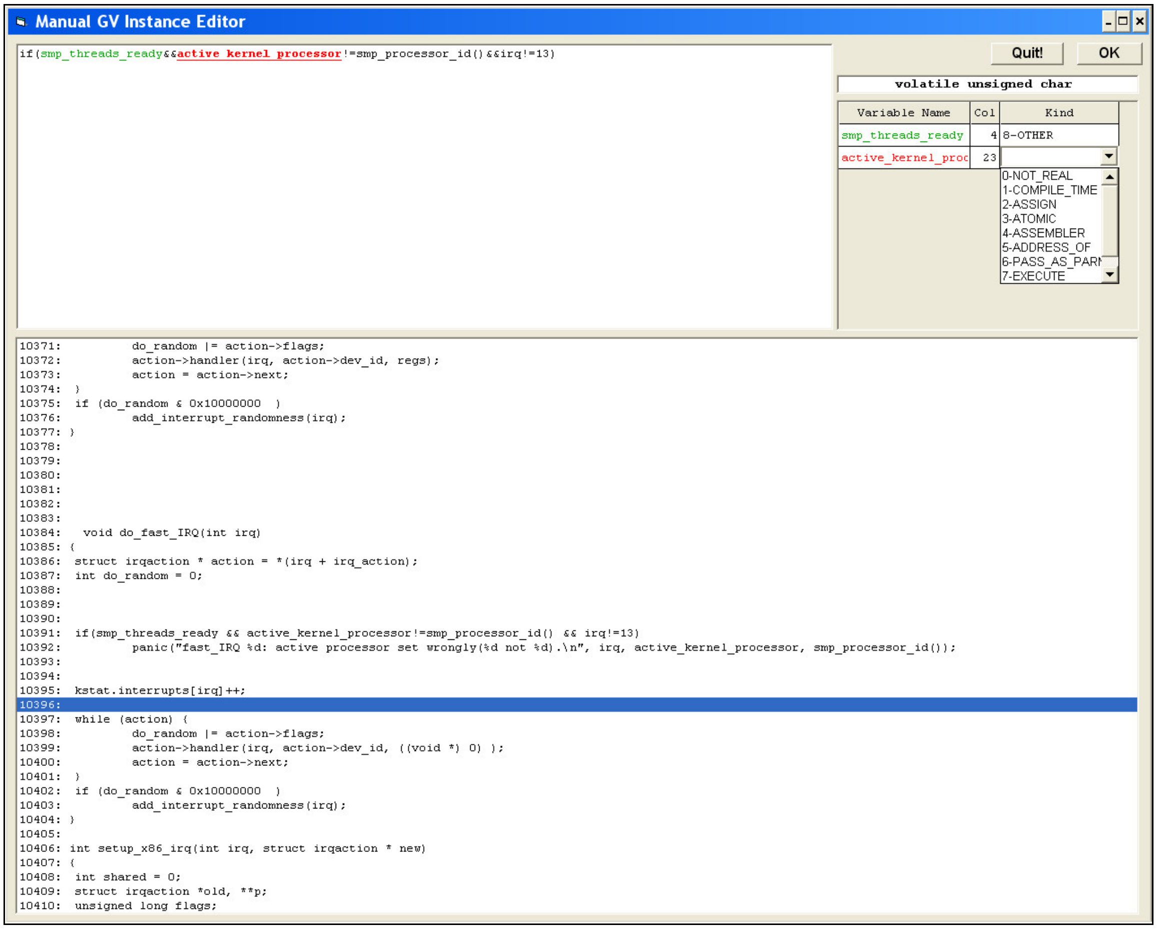
```
         111111111122222222223333
123456789012345678901234567890123
GV[1][q].member[GV2].member2 = 0;
```

In this case, GV is of Kind 2 (Assignment), and GV2 is of Kind 8 (other).

The operation of the automatic parser was independently checked by a second individual, who confirmed that it accurately classified all instances it reported.

There are many instances of GVs that the automatic parser cannot detect and/or classify. One of the more frequently occurring types of GV instances involves code spread across multiple lines. In the case of some long conditional statements, or some long function calls, single statements may be spread across multiple physical lines. The C programming language is white-space-blind, so this is not a problem for the compiler, but our parser works on only individual lines. It would have helped a bit if we had consolidated broken lines (lines ending in an operator, a comma, or an open parenthesis clearly are continued on the next physical line), but doing so would have altered the line count. We could have chosen to replace a statement spread across three lines as one long line and two blank ones, but doing so would have changed the blank line count, even though it would have preserved the total line count.

To address the need to manually categorize the GV instances the automatic parser was unable to handle, `CodeSweeper` checks every GV-bearing line for one of the 36 classes above. If any GV instance on the line could not be automatically classified, the source line was presented to the user, who could then examine the line in question, in the context of the lines immediately before and after, and manually identify the proper Kind for the GV instance. Figure 6.2 shows the Manual Instance Editor from `CodeSweeper`.



```
if(smp_threads_ready&&active kernel processor!=smp_processor_id()&&irq!=13)
```

volatile unsigned char

| Variable Name | Col | Kind |
|---|---|---|
| smp_threads_ready | 4 | 8-OTHER |
| active_kernel_proc | 23 | |

```
                do_random |= action->flags;
                action->handler(irq, action->dev_id, regs);
                action = action->next;
        }
        if (do_random & 0x10000000  )
                add_interrupt_randomness(irq);
       }






         void do_fast_IRQ(int irq)
       {
        struct irqaction * action = *(irq + irq_action);
        int do_random = 0;



        if(smp_threads_ready && active_kernel_processor!=smp_processor_id() && irq!=13)
                panic("fast_IRQ %d: active processor set wrongly(%d not %d).\n", irq, active_kernel_processor, smp_processor_id());


        kstat.interrupts[irq]++;

        while (action) {
                do_random |= action->flags;
                action->handler(irq, action->dev_id, ((void *) 0) );
                action = action->next;
        }
        if (do_random & 0x10000000  )
                add_interrupt_randomness(irq);
       }

       int setup_x86_irq(int irq, struct irqaction * new)
       {
        int shared = 0;
        struct irqaction *old, **p;
        unsigned long flags;
```

**Figure 6.2 – The Manual Instance Editor in `CodeSweeper`**

The upper-left window shows the current line of code, with all GVs highlighted. The GV instances the automatic parser was able to handle are shown in green, and whatever the parser was *not* able to classify is shown in red. The grid in the upper-right corner shows the GV names, starting columns, and Kinds. In this case, the parser was able to classify `smp_threads_ready`, starting in column 4, as Kind 8 (other), and is prompting the user to classify the second GV on the line, `active_kernel_processor`, which begins in column 23 of the line. The large box at the bottom the screen shows the current line in the context of the lines immediately before and after the line being classified. As soon as the user selects the appropriate Kind from the drop-down box in the upper-right corner, the program will go on to the next still-unclassified GV on the line (in this example there is not another one). Once all of the GVs on this line have been assigned a Kind, whether from the automatic parser (those in green) or manually (those in red), the user can click “OK,” and the program starts over, automatically parsing the next line with a GV.

In order to prevent the user from having to manually classify the same lines over and over from file to file (or kernel version to kernel version), once the user manually classifies a line, the program saves the source line, along with a “fingerprint.” The fingerprint consists of the number of GVs on the line, their names, and their starting columns. Subsequently, when the program is unable to automatically parse an entire line, it searches its repository of already-classified instances. If a matching source line is found, and the fingerprints match, the program assumes that this is the same line, appearing in a subsequent file (or kernel), and uses the Kind information, previously manually entered, that the parser has stored.

The automatic parser was able to classify approximately 90 percent of the instances in the 2.0.x kernels, 65 percent of the instances in the 2.2.x kernels, and 75 percent of the instances in the 2.4.x kernels. However, once a line has been manually classified, even if that line occurs again and now needs no manual intervention, it still counts as being manually classified every time it appears.

In order to confirm the correct manual classification of the remaining GVs, a second individual was recruited to independently perform the manual classification process from start to finish. Upon completion, the results were compared, and the situations in which the two individuals responded differently to the manual classifications were resolved by a third individual.

Although checking every line of every file seems like a huge task, `CodeSweeper` processed the entire 2.00.000 kernel (49 MB of preprocessed code in 225 files) in 71 seconds (once all of the manual instances have been classified), and the 2.04.035 kernel (248 MB in 407 files) in 261 seconds.

The output of the `CodeSweeper`, aside from the repository of manually-classified GV-bearing source lines, is a file called `GV_Data.txt`, one file per kernel version. The program also provides the count of GV instances by automatically-classified Class (with a single-line total for the manually-classified instances), and a count by Kind for the whole kernel.

The `GV_Data.txt` file contains one six-field, tab-delimited record per GV instance. The fields are:

1) The file name (with path, relative to the directory immediately below /`linux/`):
2) The function name:
3) The line within the source file of this instance;
4) The GV name;

5) The column in which the GV instance begins; and

6) The Kind of this instance (regardless of whether it was classified manually or automatically).

The first few records from this file for kernel version 2.0.0 are shown in Figure 6.3.

```
   F I L E   N A M E           Function    Line    GV Name        Col
Kind
--------------------------  ----------  ----    -----------   ---  ---
-
arch\i386\kernel\ioport_C.c  apic_write   1437   apic_reg       29    8
arch\i386\kernel\ioport_C.c  apic_read    1442   apic_reg       35    8
arch\i386\kernel\ioport_C.c  strtok       6667   ___strtok      19    4
arch\i386\kernel\ioport_C.c  strtok       6668   ___strtok       6    4
arch\i386\kernel\ioport_C.c  suser        8797   current_set     7    8
arch\i386\kernel\ioport_C.c  suser        8798   current_set     4    8
arch\i386\kernel\ioport_C.c  file_from_fd 8828   current_set    10    8
arch\i386\kernel\ioport_C.c  select_wait  8888   current_set    21    8
arch\i386\kernel\ioport_C.c  sys_ioperm   9004   current_set    31    8
arch\i386\kernel\irq_C.c     apic_write   1879   apic_reg       29    8
arch\i386\kernel\irq_C.c     apic_read    1884   apic_reg       35    8
arch\i386\kernel\irq_C.c     strtok       6966   ___strtok      19    4
arch\i386\kernel\irq_C.c     strtok       6967   ___strtok       6    4
arch\i386\kernel\irq_C.c     suser        8939   current_set     7    8
arch\i386\kernel\irq_C.c     suser        8940   current_set     4    8
arch\i386\kernel\irq_C.c     file_from_fd 8970   current_set    10    8
arch\i386\kernel\irq_C.c     select_wait  9030   current_set    21    8
arch\i386\kernel\irq_C.c     init_bh      9141   bh_base         1    2
arch\i386\kernel\irq_C.c     init_bh      9142   bh_mask_count   1    2
arch\i386\kernel\irq_C.c     init_bh      9143   bh_mask         1    2
arch\i386\kernel\irq_C.c     mark_bh      9148   bh_active      13    6
<10,121 lines omitted>
```

**Figure 6.3 – Contents of `GV_Data.txt` file.**

The final task in processing the GV information is to read the `GV_Data.txt` file and create one record per file/function pair, consolidating the Kinds into counts per kind. This file is called `GV_Data_Processed.dat`, and is created by the program we wrote called GV_Data_Processor_&_DB_Loader. This program also loads the resulting file into the database for subsequent analysis by Grapher (see Chapter 7).

A sample of the contents of GV_Data_Processed.dat is shown in Figure 6.4.

```
                                                 <--- Counts by Kind ---->
    F I L E   N A M E          Function          0  1  2  3  4  5  6  7  8
  --------------------------   ------------      -  -  -  -  -  -  -  -  -
  arch\i386\kernel\ioport_C.c  apic_write        0  0  0  0  0  0  0  0  1
  arch\i386\kernel\ioport_C.c  apic_read         0  0  0  0  0  0  0  0  1
  arch\i386\kernel\ioport_C.c  strtok            0  0  0  0  2  0  0  0  0
  arch\i386\kernel\ioport_C.c  suser             0  0  0  0  0  0  0  0  2
  arch\i386\kernel\ioport_C.c  file_from_fd      0  0  0  0  0  0  0  0  1
  arch\i386\kernel\ioport_C.c  select_wait       0  0  0  0  0  0  0  0  1
  arch\i386\kernel\ioport_C.c  sys_ioperm        0  0  0  0  0  0  0  0  1
  arch\i386\kernel\irq_C.c     apic_write        0  0  0  0  0  0  0  0  1
  arch\i386\kernel\irq_C.c     apic_read         0  0  0  0  0  0  0  0  1
  arch\i386\kernel\irq_C.c     strtok            0  0  0  0  2  0  0  0  0
  arch\i386\kernel\irq_C.c     suser             0  0  0  0  0  0  0  0  2
  arch\i386\kernel\irq_C.c     file_from_fd      0  0  0  0  0  0  0  0  1
  arch\i386\kernel\irq_C.c     select_wait       0  0  0  0  0  0  0  0  1
  arch\i386\kernel\irq_C.c     init_bh           0  0  3  0  0  0  0  0  0
  arch\i386\kernel\irq_C.c     mark_bh           0  0  0  0  0  0  1  0  0
```

**Figure 6.4 – Sample contents of GV_Data_Processed.dat.**

After loading this information into the database, we analyzed the breakdown of the GV instances by Kind and by kernel series.

## 6.4 Analysis of the Instance Kinds

Figure 6.5 shows the total number of instances of GVs in all versions of the kernel we examined. The criteria we used to select kernels for examination are discussed in Chapter 8.

We emphasize that the number of instances of GVs in the source code is not the same as the common coupling count. In computing common coupling, we consider the number of files that access the same global variables. Accordingly, we only consider one instance of a global variable within a file.

In this chapter, we are interested in only the number of times the GVs appear, within the entire corpus of executable code within a kernel, and what Kind of GV access each instance is.

We begin with a high-level view of the number of GV instances across the 2.00.x, 2.02.x, and 2.04.x kernel series. Figure 6.5 shows the total number of instances of GVs in the executable code of each version of the kernel.

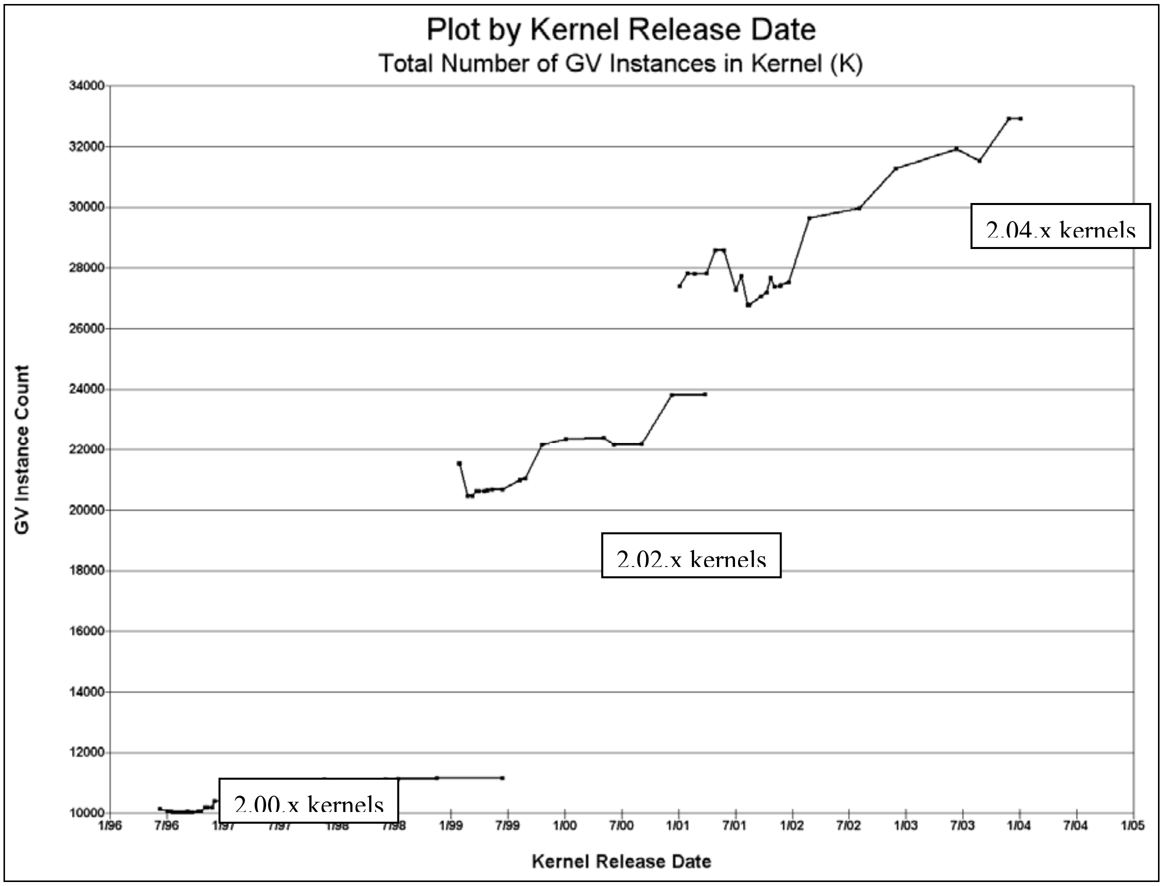


**Figure 6.5 – Total number of instances of GVs in code, for all examined kernels.**

Figure 6.5 shows that the number of times that GVs appear in the Linux kernel code rises from series to series, and that, generally, within a series, the number of GV instances increases.

We next examine the breakdown of the GV occurrences by Kind.

Figure 6.6 shows the percentage of Kind-1 (Compile-time) instances in the three kernel series.

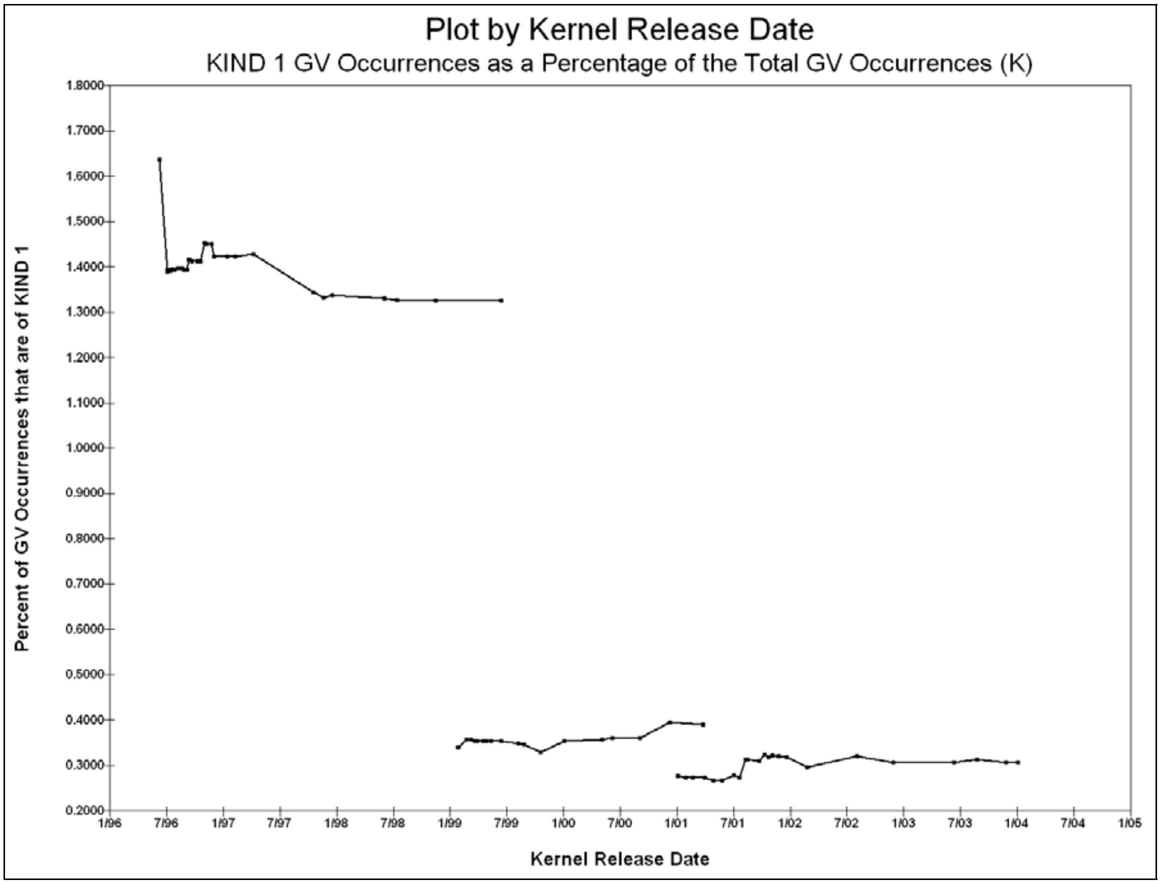


**Figure 6.6 – Percentage of Kind-1 (Compile-Time) GV Occurrences**

Figure 6.6 shows that the Kind-1 occurrences account for an exceedingly small minority of the total GV instances in the code for all three kernel series. In the 2.00.x series, approximately 1.4 percent of the times a GV appears in code, it is of Kind 1 (Compile-time). As the code matured from the 2.00.x to 2.02.x series, the percentage dropped to approximately 0.36 percent, and fell again slightly to 0.30 percent in the 2.04.x series. Because these constitute such a small minority of the GV occurrences in all three series, we consider them to be negligible.

Figure 6.7 shows the percentage of Kind-3 (Atomic operation) GV occurrences.

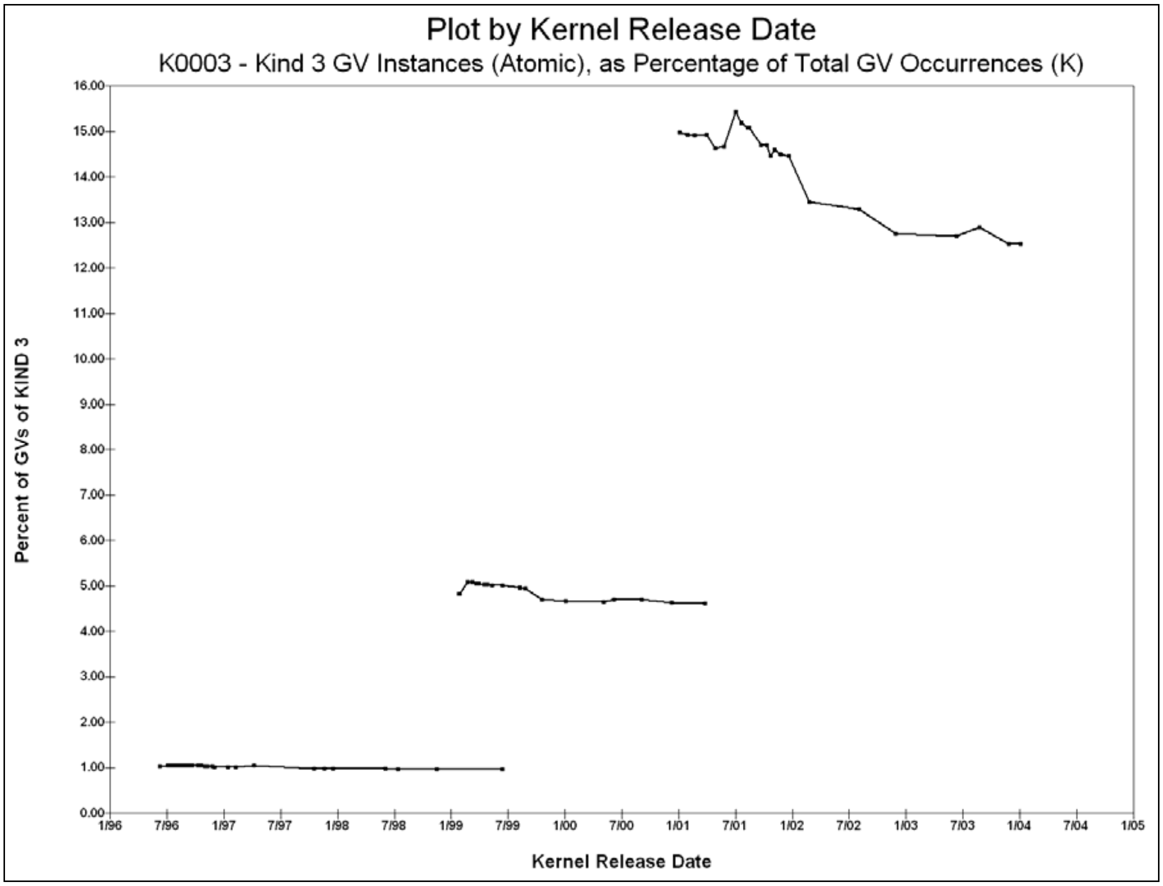


**Figure 6.7 – Percentage of Kind-3 (Atomic Operation) GV occurrences**

The cause for the strong increase in the percentage of Kind-3 GV occurrences in the 2.04.x series is not clear. We believe, however, that, as the kernel continued to mature, and as more subsystems were included, synchronization between them became increasingly important, and, therefore, the number of Kind-3 instances naturally rose.

Figure 6.8 shows the percentage of Kind-4 (Assembler) GV occurrences. It is not clear why the number of GV instances in assembler calls rose so sharply between the 2.00.x and 2.02.x series, and then fell to near zero in the 2.04.x series, but we believe that the drop in the 2.04.x series is related to the `extern inline` issue discussed in Chapter 9.

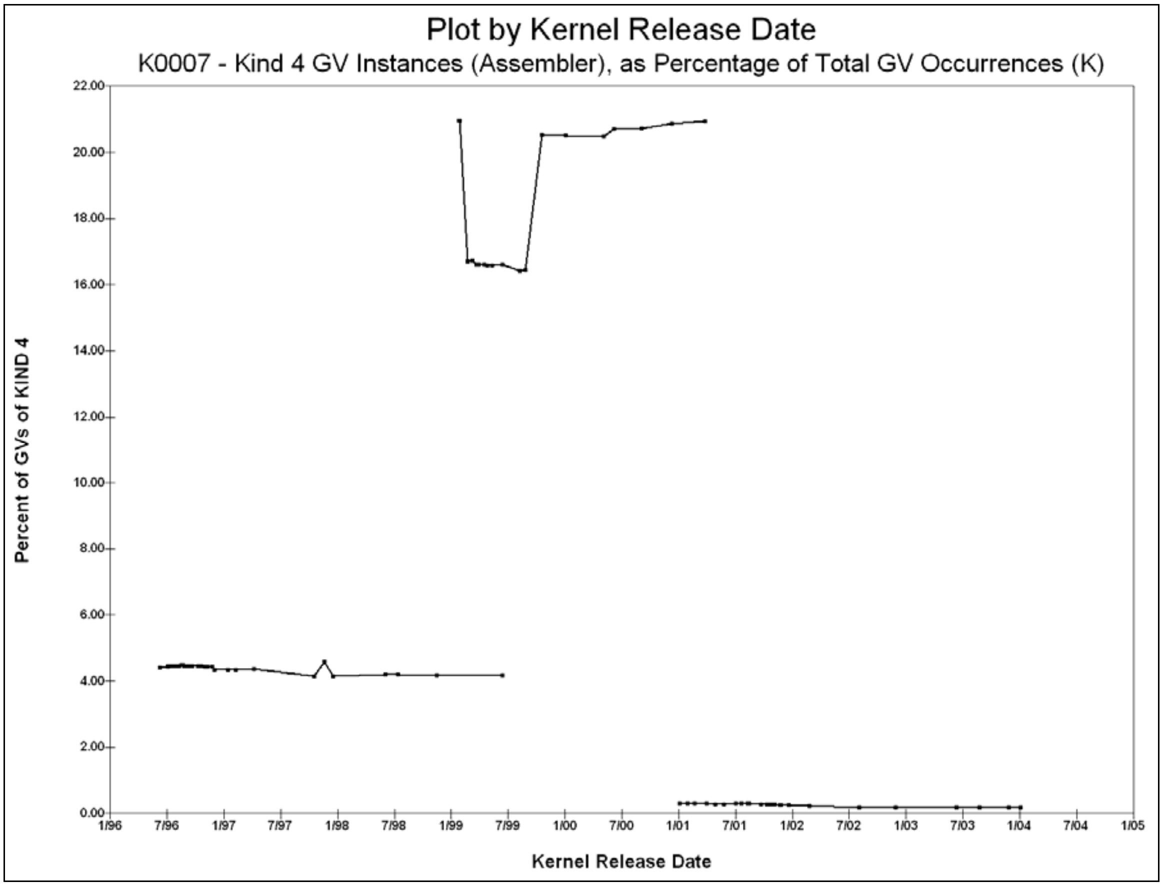


**Figure 6.8 – Percentage of Kind-4 (Assembler) GV Occurrences**

In the interest of brevity, we cite the percentages of the Kind-5, -6, and -7 GV instances, but skip the graphical display.

GV instances of Kind-5 (Address-of) constitute 2.38 percent, 5.29 percent, and 5.04 percent of the 2.00.x, 2.02.x, and 2.04.x series, respectively. Because we cannot determine whether the Kind-5 instances lead to the modification of a GV or not, we cannot delve any deeper into the potential significance of these values, other than to note them.

GV instances of Kind-6 (Passed as parameter) constitute 8.65 percent, 10.09 percent, and 9.81 percent of the 2.00.x, 2.02.x, and 2.04.x series, respectively. Because we do not differentiate between passing parameters by value and passing them by reference, we cannot determine whether the Kind-6 instances lead to the modification of a GV or not, and, again, we cannot delve any more deeply into the potential significance of these values, other than to note them and to point out that they remain relatively consistent from series to series.

The percentages of the Kind-7 (execution) occurrences were also so small as to be irrelevant, at 0.31 percent, 0.15 percent, and 0.06 percent in the 2.00.x, 2.02.x, and 2.04.x kernels, respectively.

We are now left with Kinds 2 (Assignment, or def) and 8 (Other, or use). These are the only two Kinds for which we can tell conclusively whether or not the value of the GV is being altered. The modification of a GV is more dangerous than the use of a GV's value. As we stated in Section 2.3.1, common coupling is a problem when one module alters a GV and another module that uses that GV either does not anticipate or does not check for the change in value. One might argue that, as long as the GV accesses are "write once, read many" (i.e., the GVs are modified in only an extremely small number of locations relative to the number of times they are read), a high number of GV occurrences is more benign than if the GVs are being modified more frequently relative to the number of times they are only read.

Figure 6.9 shows the percentage of Kind-2 (Assignment) GV occurrences. These are the "definite def" or "destructive reference" instances in which the value of the GV unquestionably is modified.

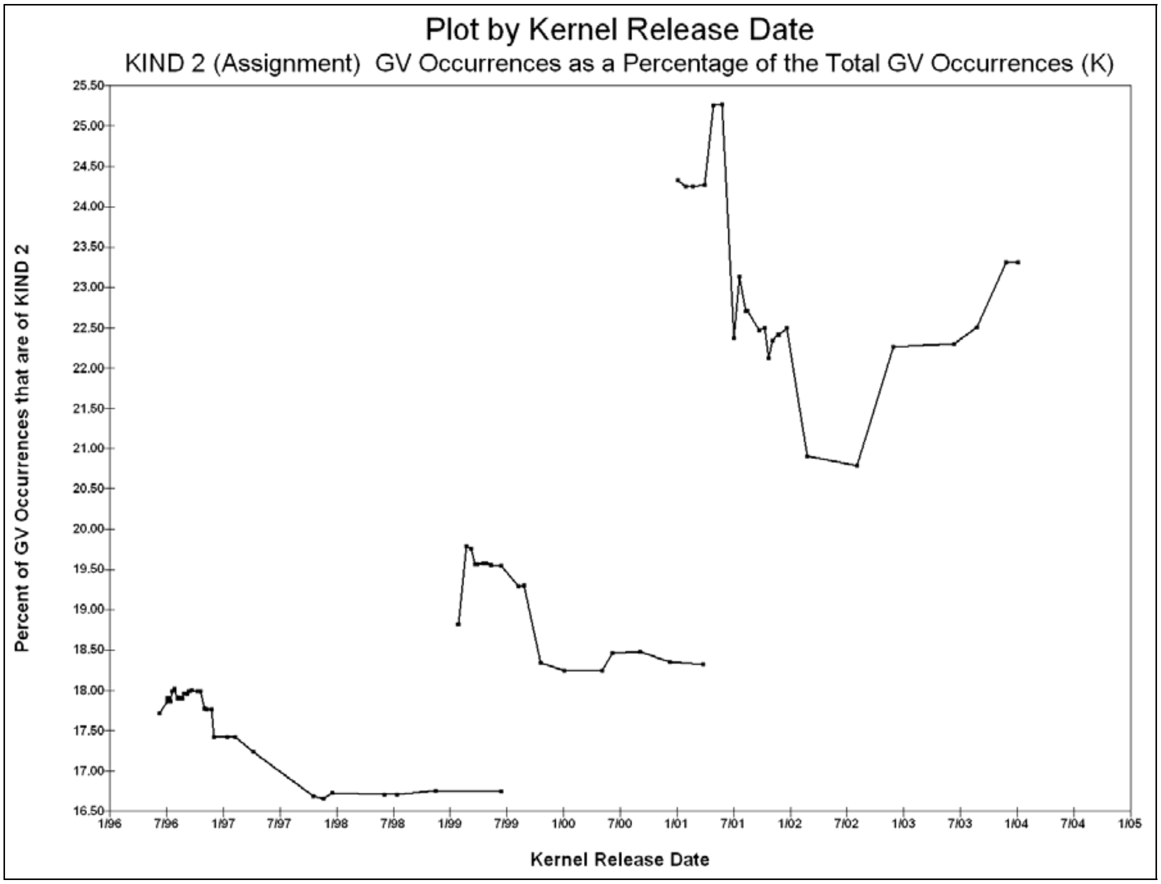


**Figure 6.9 – Percentage of Kind-2 (Assignment, or def) GV Occurrences**

Figure 6.9 shows that the percentage of the Kind-2 GV instances, those in which the value of the GV is definitely being modified, rose from an average of 17.6 percent in the 2.00.x series, to 19.1 percent in the 2.02.x series, and 22.9 percent in the 2.04.x series.

Figure 6.10 shows the percentage of Kind-8 (Other, or use) GV Occurrences.

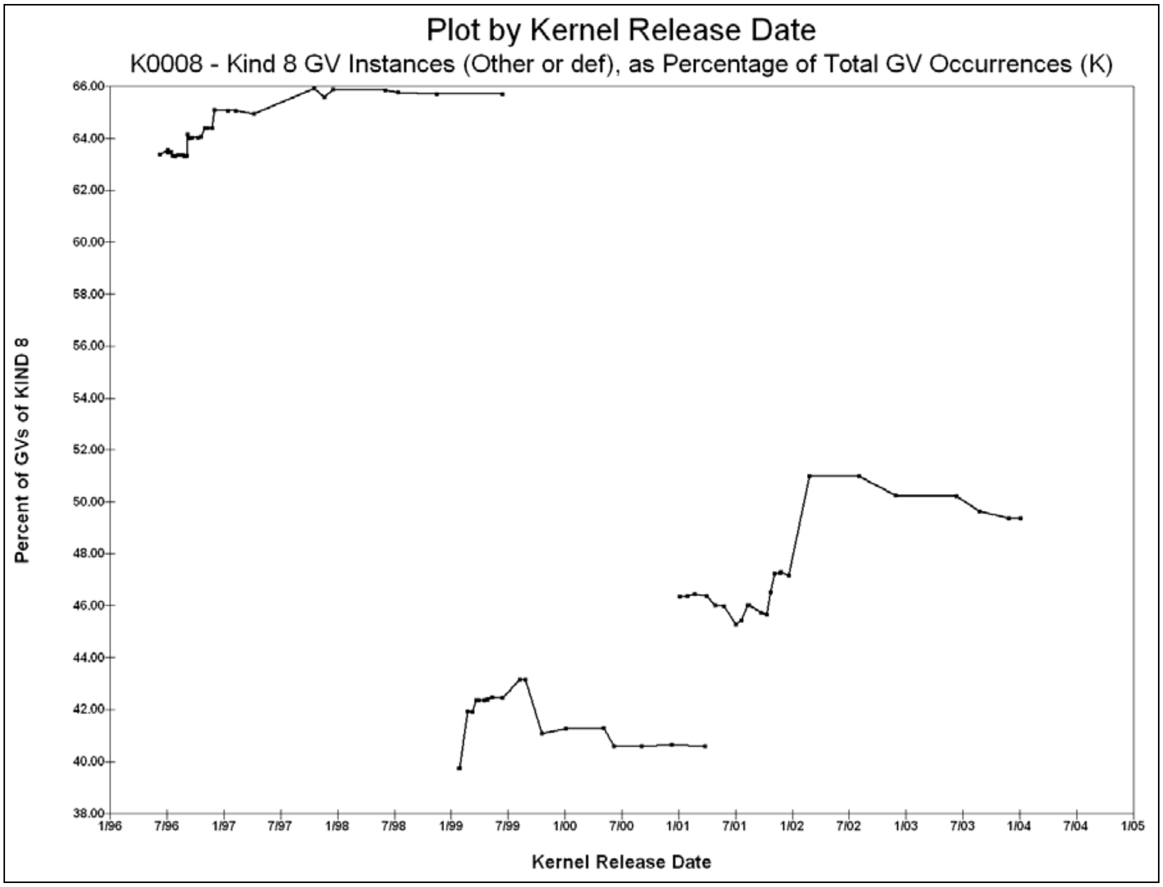


**Figure 6.10– Percentage of Kind-8 (Other, or use) GV Occurrences**

Figure 6.10 shows that, on average, nearly two-thirds (64.2 percent) of the GV occurrences in the 2.00.x series were of Kind 8, although the percentage dropped to (on average) 41.6 percent in the 2.02.x series, and rose slightly to 47.4 percent in the 2.04 series.

We compare the average percentages, by kernel series, from Figures 6.9 and 6.10 in Table 6.1.

**Table 6.1 Kind-2 and Kind-8 Percentages (as Percent of Total GV Occurrences)**

| Kernel Series | Kind-2 Percentage | Kind-8 Percentage | Percentage of Total GV Instances Accounted For | Ratio of Kind-2 Percentage to Kind-8 Percentage |
|---|---|---|---|---|
| 2.00.x | 17.63% | 64.19% | 81.83% | 0.2747 |
| 2.02.x | 19.06% | 41.63% | 60.69% | 0.4579 |
| 2.04.x | 22.93% | 47.35% | 70.28% | 0.4843 |

Table 6.1 shows that, in the 2.00.x series, there was (approximately) one GV write for every 3.6 GV reads, and in the 2.02.x and 2.04.x, it was nearly one-to-two.

We recognize that the Kind-2 and Kind-8 instances collectively account for only 60 to 80 percent of the total number of GV instances. To provide verification that we have not reached an unreasonable conclusion in Table 6.1, we repeat the above with the absolute counts of the GV instances, rather than the percentage of the totals. This is shown in Table 6.2.

**Table 6.2 Kind-2 and Kind 8-Percentages (Absolute Counts of GV Occurrences)**

| Kernel Series | Kind-2 Occurrences | Kind-8 Occurrences | Ratio of Kind-2 Occurrences to Kind-8 Occurrences |
|---|---|---|---|
| 2.00.x | 1,813 | 6,609 | 0.2743 |
| 2.02.x | 4,089 | 8,934 | 0.4576 |
| 2.04.x | 6,562 | 13,582 | 0.4832 |

The findings in Table 6.2 confirm that it does not matter whether or not we consider the other six Kinds in determining the relative number of defs per use.

This certainly negates the argument that the GV usage pattern might be described as "write once, read many." Although there certainly are more GV reads than there are writes, the ratio in the 2.02.x and 2.04.x series is approximately only 2:1.

## 6.5 Analysis of the Instance Kinds and Common Coupling

We now consider the relationship between GV instances in code and common coupling. In Section 2.3.1, we stated that, if there are occurrences of a global variable in *N* files, there will be $\binom{N}{2} = \frac{\left(N^2 - N\right)}{2}$ instances of common coupling. The common coupling count is determined by considering groups of *N* files accessing common GVs, whereas the Kind count simply considers the entire body of source code as a single entity (*not* divided into files). Furthermore, the common coupling is $O(N^2)$. Accordingly, we would not expect to find a linear relationship between the "simple" GV instance count and the somewhat more complicated common coupling count.

Figure 6.11 shows the correlation between the common coupling count and the kernel-wide number of occurrences in the 2.00.x series. Contrary to our expectations, there is a very highly significant correlation ($P < 0.001$) between the total number of GV occurrences in the entire source tree and the kernel-wide common coupling count in the 2.00.x series of kernels. The 2.02.x and 2.04.x series also exhibit similar very highly significant correlations.

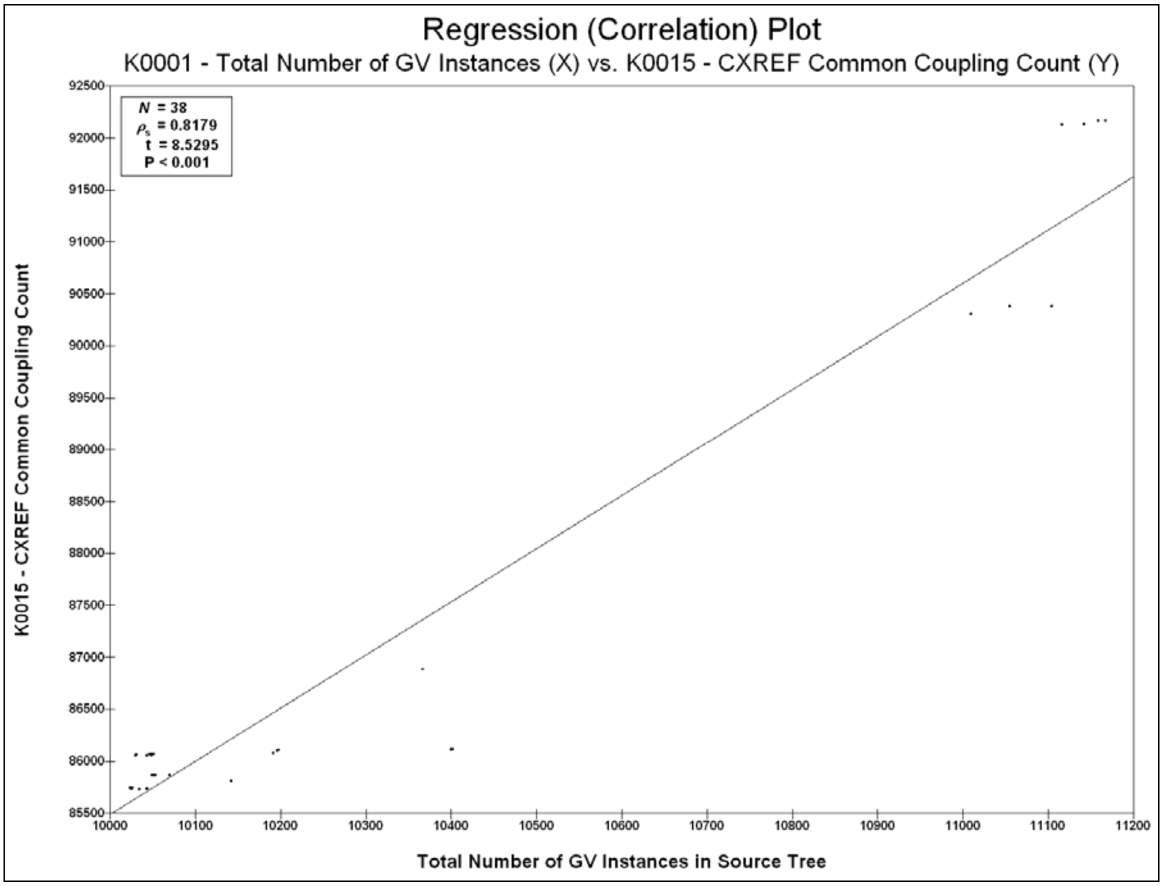


**Figure 6.11– Correlation of Total GV Occurrences and Common Coupling Count in the 2.00.x kernel series.**

We then repeated the correlation analysis, this time using only the Kind-2 instances and again using only the Kind-8 instances.

In the interest of brevity, we omit the graphs, but summarize the results in Table 6.3. All correlations were very highly significant (P < 0.001), and we so indicate throughout the remainder of this dissertation by using bold italics and a heavy cell border line for those cells corresponding to a very highly significant correlation.

**Table 6.3 – Correlations between Number of GV Occurrences (Kind-2, Kind-8, and Total) and Common Coupling, by Kernel Series.**

| | 2.00.x (N = 38) | 2.02.x (N = 20) | 2.04.x (N = 25) |
|---|---|---|---|
| Kind-2 Occurrences vs. Common Coupling | $\rho_s = 0.7488$<br>$t = 6.7788$<br>$P < 0.001$ | $\rho_s = 0.8861$<br>$t = 8.1102$<br>$P < 0.001$ | $\rho_s = 0.9136$<br>$t = 10.773$<br>$P < 0.001$ |
| Kind-8 Occurrences vs. Common Coupling | $\rho_s = 0.9234$<br>$t = 14.429$<br>$P < 0.001$ | $\rho_s = 0.6969$<br>$t = 4.1229$<br>$P < 0.001$ | $\rho_s = 0.6944$<br>$t = 4.6277$<br>$P < 0.001$ |
| Total GV Occurrences vs. Common Coupling | $\rho_s = 0.8179$<br>$t = 8.5295$<br>$P < 0.001$ | $\rho_s = 0.8998$<br>$t = 8.7523$<br>$P < 0.001$ | $\rho_s = 0.8736$<br>$t = 8.6084$<br>$P < 0.001$ |
| Kind-2 vs. Kind-8 Occurrences | $\rho_s = 0.7702$<br>$t = 7.2449$<br>$P < 0.001$ | $\rho_s = 0.6951$<br>$t = 4.1017$<br>$P < 0.001$ | $\rho_s = 0.7969$<br>$t = 6.3268$<br>$P < 0.001$ |

The fact that all of these correlations are very highly significant means that the following are all validated against common coupling in the 2.00.x, 2.02.x, and 2.04.x series of the Linux kernel:

- Total number of GV instances in preprocessed source code;
- Total number of Kind-2 (def) instances in preprocessed source code; and
- Total number of Kind-8 (use) instances in preprocessed source code.

Furthermore, the correlation between Kind-2 and Kind-8 instances is also very highly significant.

# CHAPTER VII

# SCHACH ET AL. 2002, REDUX

## Contribution of this Chapter

1. The authors of [Schach et al., 2002] used the Linux kernel version number as their independent variable, and found that that the common coupling in the Linux kernel was growing exponentially. When we re-evaluated Schach et al.'s data with respect to release date, rather than version number, we obtained a linear growth rate. Repeating this study with our standardized configuration, when applied to the preprocessed source code for three separate kernel series, we again found that the growth of common coupling in the Linux kernel is linear with respect to time.

## 7.0 Introduction

In Chapter 4 we discussed several motivations for reconsidering the work of the authors of [Schach et al 2002]. In this chapter, we revisit and extend their work using our analysis criteria.

## 7.1 Which Versions of the Linux Kernel *Should* We Consider?

It would seem obvious that *all* versions of the kernel should be analyzed, but that knee-jerk reaction can lead to some incorrect conclusions, depending on what we analyze, and how we do so. Schach et al. analyzed (nearly) all available versions of the kernels except when the code forked (i.e., when a new development series began), after which they dropped the old production series from consideration. This had the advantage of giving them a large number of kernels to consider, with no overlapping between the series (i.e., the release dates of the kernels they analyzed were monotonically increasing). Their analysis was performed on a version-by-version basis, without regard for the release date (or the elapsed time between releases) of any versions.

One problem with this selection criterion, as discussed in Section 4.3, is that this approach skews the results heavily towards the development versions, which are not typical of the Linux code running on the vast majority of systems worldwide. We wish to consider only the versions operating in production environments in the "real world," rather than on the machines of the developers or those "bleeding edge" experimenters brave enough to operate in a perpetual test environment.

The terms "development" and "production" might seem to imply a rather hard division, wherein:

- The production series of kernels are largely static releases, the object of purely corrective maintenance (primarily for stability and security); and
- The development series kernels are the only versions on which perfective maintenance is performed (primarily to incorporate new features).

Unfortunately, this division does not adequately represent the relationship between the two series. According to [http://en.wikipedia.org/wiki/Linux_kernel#Version_numbering], with respect to the "B" part of version numbers in the format "A.B.C" (for example, the "04" part of "2.04.017"):

> "Prior to the Linux 2.[0]6.x series, even numbers indicate a stable release, i.e. one that is deemed fit for production use, such as 1.[0]2, 2.[0]4 or 2.[0]6. Odd numbers have historically been development releases, such as 1.[0]1 or 2.[0]5. They were for testing new features and drivers until they became sufficiently stable to be included in a stable release."

The key phrase here is "until they became sufficiently stable to be included in a stable release." The implied hard division above would be more of a gestational model, with the new features not appearing in a production series kernel until the "birth" of a new production cycle (the X.Y.000 version of the next production series). In reality, the model was more accurately described as one in which new features were introduced in the development versions, where they were extensively tested and, if necessary, modified (corrective maintenance) before being integrated into the production series. In other words, both perfective and corrective maintenance were performed in the development versions, whereas the primary activity in the development versions was largely corrective maintenance. For this reason, we hereafter refer to the *production* versions as *stable* versions.

The next obvious question becomes one of *which* stable versions to consider, and by extension, which ones to exclude, and how we should determine the cutoff points.

## 7.2 Development vs. Stable Series and Their Life Cycles

The three stable series kernels – 2.00.x, 2.02.x, and 2.04.x – were similar in their relationships with their corresponding development series:

- They began as the result of the end of a development series. Ostensibly, the development series each reached a point at which their new features could no longer be incorporated into the existing stable series; a new, fundamentally different stable series eventually had to be initiated. These were the 2.00.000, 2.02.000, and 2.04.000 versions.
- There was a flurry of development activity following each ".000" version, characterized by relatively frequent releases. This was the "settling in" period for the new stable series.
- Once the new stable series reached a certain point, a new development series began. During the period of the development series, the stable series releases occurred less often, but additional new features were incorporated into the stable series kernels. These were the periods of most significant growth of the stable series, yet Schach et al. dropped these kernels from their analysis.
- When a development series concluded, with the launch of a new stable series, the previous stable series continued on for some time, with two distinct phases:
  1. When the new stable series was first released, there was already one more version of the old stable series under development ("one still in the pipeline"). This version would eventually be released, and was the last version in the old stable series to be the subject of significant development effort. In the case of the 2.04.x kernels, the release "in the pipeline" was minor, and the *second* release

following the release of 2.06.000 was the last release produced as a result of significant changes to the code.

2. As soon as a new stable series was released, users would have begun to migrate to it. Experimenters and other early-adopters would have been first, followed by desktop users, and servers would probably have been the last to upgrade to the new series. Some time around this point, the old stable series entered "lame duck" status, although there was still some minimal (and ever-decreasing) development in the old series.

Servers are typically deployed for longer-term, hands-off operation, and, as such, servers tend to undergo less frequent updating of software. IT (Information Technology) managers, unless there is a compelling reason, such as a fix to a security vulnerability or to correct some other fault in the software, rarely update servers simply to add a new feature. If a server is running at an acceptable performance level, there is little motivation to rock the boat with an upgrade just to obtain some non-essential feature. IT managers, interested in stability, are more likely to "hang on" to the old kernel, avoiding the risk of disruption in exchange for the newest gee-whiz feature that does nothing to actually improve server performance. Servers do not run forever, however, and will most likely be decommissioned or have their functionality migrated to newer, more capable hardware. In the process of moving to new hardware, it makes sense at that time to also jettison the old software.

> Because servers are typically the last holdouts continuing to run the older stable releases, there was still some (very limited) demand for development of the old stable releases, even though in our opinion they had been abandoned by the vast majority of the user base. This final phase in the life cycle of a stable series was marked by exceptionally long release intervals and minimal development. Eventually, all development efforts ceased, and the final version was frozen.
>
> Because the majority of the world's web pages reside on servers running Linux/Apache, we believe that the Linux server market is the primary reason for the extended length of the lame duck portion of the kernel's life cycle model.

This life-cycle model was employed through all series up to 2.06.x, at which time the development and stable releases were merged. There are no plans for a 2.07.x development series; nor is there a planned 2.08.x stable series. At the time of this writing, the 2.06.x series is approximately 4.25 years old.

## 7.3 Kernel Versions Examined by Schach et al.

### 7.3.1 Schach et al. and the 2.00.x kernels

Schach et al. recognized the problems that the parallel development and stable series posed in terms of performing a longitudinal analysis. To circumvent these problems, they "decided to discard all versions with a date stamp later than a version with a higher version number."

Schach et al. “discovered that versions 2.[0]0.[0]30 through 2.[0]0.[0]39 had date stamps later than the date stamp on version 2.[0]1.[00]0.” Accordingly, versions 2.00.030 through 2.00.039 were discarded and excluded from their analysis. Version 2.00.040 had not yet been released, but under their reasoning, it would have been excluded as well.

A review of the dates on the various releases at [`www.linuxhq.com/kernel`] shows that version 2.01.000 was released on Sept 30, 1996, and that versions 2.00.021 and 2.00.022 were released on September 20 and October 8, 1996, respectively, placing the actual beginning of the 2.01.x development cycle several versions earlier than Schach et al. indicated.

The reason for this discrepancy in the dates (and a similar discrepancy in the 2.02.x series) is most likely due to the fact that the Schach et al. study used the file date stamps on the files they examined in the `/linux/` subdirectory of the source tree. It is possible that these particular files were not modified the same date as some other file(s) in the release. We used the entire kernel’s release date, rather than the individual file stamp date of the files in the `/linux/` subdirectory, for all date-based issues.

### 7.3.2 Schach et al. and the 2.02.x kernels

Schach et al. determined that “versions 2.[0]2.[00]2 through 2.[0]2.[0]17 had date stamps later than the date stamp on version 2.[0]3, so [they] ignored those 16 versions.” Clearly, Schach et al. did not include enough of the 2.02.x kernels to be able to draw any particular conclusion about their growth rate.

A careful examination of the release dates from [www.linuxhq.com/kernel] shows that version 2.03.000 was released on May 11, 1999, the same date that version 2.02.008 was released, placing the actual beginning of the 2.03.x development cycle several versions later.

### 7.3.3 Schach et al. and the 2.04.x kernels

The release of 2.04.000 took place after the Schach et al. study had been completed, and therefore there is no 2.04.x data from that study on which we can comment. In the interest of augmenting their research, however, we examined 2.04.x in a manner consistent with our treatment of 2.00.x and 2.02.x.

## 7.4 Development/Stable Code Forking History

Table 7.1 shows the key dates and versions in the code forking history.

**Table 7.1 – Linux Kernel Code Branching with Dates.**

| 2.00.x | 2.01.x | 2.02.x | 2.03.x | 2.04.x | 2.05.x | 2.06.x |
|---|---|---|---|---|---|---|
| 000 09-Jun-96 | | | | | | |
| … | | | | | | |
| 021 20-Sep-96 | | | | | | |
| | 000-30-Sep-96 | | | | | |
| 022 08-Oct-96 | … | | | | | |
| … | … | | | | | |
| 036 16-Nov-98 | … | | | | | |
| | 132 22-Dec-98 | | | | | |
| | | 000 26-Jan-99 | | | | |
| | | … | | | | |
| | | 008 11-May-99 | 000 11-May-99 | | | |
| 037 13-Jun-99 | | … | … | | | |
| 038 25-Aug-99 | | … | … | | | |
| | | 014 04-Jan-00 | … | | | |
| | | | 052 13-Mar-00 | | | |
| | | 015 04-May-00 | | | | |
| | | … | | | | |
| | | 018 10-Dec-00 | | | | |
| | | | | 000 04-Jan-01 | | |
| 039 09-Jan-01 | | | | | | |
| | | | | 001 30-Jan-01 | | |
| | | | | 002 22-Feb-01 | | |
| | | 019 25-Mar-01 | | | | |
| | | | | 003 30-Mar-01 | | |
| | | | | … | | |
| | | | | 013 24-Oct-01 | | |
| | | 020 02-Nov-01 | | | | |
| | | | | 014 05-Nov-01 | | |
| | | | | 015 23-Nov-01 | | |
| | | | | | 000 25-Nov-01 | |
| | | | | 016 26-Nov-01 | … | |
| | | | | … | … | |
| | | | | 018 25-Feb-02 | … | |
| | | 021 20-May-02 | | | … | |
| | | | | 019 03-Aug-02 | … | |
| | | 022 16-Sep-02 | | | … | |
| | | | | 020 28-Nov-02 | … | |
| | | 023 29-Nov-02 | | | … | |
| | | 024 05-Mar-03 | | | … | |
| | | 025 17-Mar-03 | | | … | |
| | | | | 021 13-Jun-03 | … | |
| | | | | | 075 10-Jul-03 | |
| | | | | 022 25-Aug-03 | | |
| | | | | 023 28 Nov-03 | | |
| | | | | | | 000 18-Dec-03 |
| | | | | 024 05-Jan-04 | | … |
| 040 08-Feb-04 | | | | | | … |
| | | | | 025 18-Feb-04 | | … |
| | | 026 25-Feb-04 | | | | … |
| | | | | 026 14-Apr-04 | | … |
| | | | | … | | … |
| | | | | 036 01-Jan-08 | | … |

Table 7.1 shows how the various code series interrelate in terms of timing. The table is set up with every row in ascending chronological order (though obviously not to scale). Each cell begins with a three digit number, corresponding to the "x" in the version series shown at the top of the column. An ellipsis (…) shows a range of consecutive versions, a vertical arrow shows a gap in the column between a single pair of consecutive versions, and a horizontal dashed line shows where one version was released between two other versions in a previous series.

Some of the pertinent information we can glean from this table includes:

- The 2.00.x series began (with 2.00.000) on 09-Jun-1996.
- The 2.01.x series began on 30-Sep-1996, between 2.00.021 and 2.00.022.
- The 2.01.x series ended with 2.01.132 on 22-Dec-1998, between 2.00.036 and 2.00.037.
- The 2.02.x series began on 26-Jan-1999, also between 2.00.036 and 2.00.037.
- By the time 2.00.037 was released, the 2.02.x series was up to version 2.00.008.
- The same day that 2.00.008 was released (11-May-1999), the 2.03.x series began.
- The 2.03.x series ended on 13-Mar-2000, between 2.02.014 and 2.02.015.
- Although the 2.03.x series ended on 13-Mar-2000, the 2.04.x series did not begin until approximately 10 months later, on 04-Jan-2001. In this 10-month interval, versions 2.02.015 through 2.02.018 had been released.

Schach et al. dropped all subsequent versions once *any* new branch began. There is a discrepancy in the date stamps, as discussed in Section 7.3.1, but according to the description of their approach, they should have dropped the entire 2.00.x series following the 30-Sep-1996 release of 2.01.000 (2.00.022 – 2.00.040).

Schach et al. dropped all 2.02.x kernels following the release of 2.03.000 (although, as discussed in Section 7.3.2, there is another date discrepancy, and our reckoning of dates under their reasoning would have dropped versions 2.02.009 – 2.00.026).

Similarly, although Schach et al. did not have the 2.04.x kernels to consider, their approach would have dropped all 2.04.x versions following the release of 2.05.000.

As explained in Sections 7.1 and 7.2, once a new stable series begins, the previous stable series goes into a "lame duck" phase. Three questions therefore now arise:

1) Do we treat the "lame duck" portion differently to the rest of the series; and
2) If so, how; and
3) How do we decide precisely when the "lame duck" phase begins?

The answer to the first question is that we *must* treat it differently. Once a given stable series has been superseded by a new stable series, the older series is the object of ever-decreasing development for an ever-shrinking user base, as previous users migrate to the newer branch, and new users begin their relationship with Linux under the new branch. Because the release frequency of the old stable series drops dramatically following the debut of a new stable series, and because there is comparatively little development effort put into these "lame duck" versions, they are essentially duplicated data points with grossly disproportionate displacements in time. These data points then become a source of direct corruption of the data set, rather than just noise. The answer to the second question, therefore, is that we must ignore the “lame duck” points.

The remaining question is how to decide precisely when the “lame duck” phase begins.

## 7.5 Selecting a Cutoff Point

We need an objective method for dividing the development effort into “pre-lame duck” and “lame duck” periods, wherein the decision is driven purely by the data itself.

Because we are not considering development versions in our analysis, we might begin the process of selecting a cutoff point (that is, determining precisely where the “lame duck” portion begins) by dropping versions of the older stable series immediately upon the release of the first version of the next *stable* series, rather than the next development series (similar to the approach employed by Schach et al.).

As discussed in Section 7.2, however, the first release in an older stable branch following the first release of a newer branch was also of sufficient importance as to be included, and development of this version was initiated prior to the debut of the new stable series. Therefore, we also include:

- 2.00.037, the first 2.00.x release following the first 2.02.x release (2.02.000);
- 2.02.019, the first 2.02.x release following the first 2.04.x release (2.04.000); and
- 2.04.024, the first 2.04.x release following the first 2.06.x release (2.06.000).

## 7.6 Confirming Our Cutoff Points by Examining Code Churn

Code changes are measured in the number of lines added, changed, and deleted between any two successive releases of a product. This is known as *code churn*. If we sum the number of lines added, deleted, and modified between two consecutive versions to create a single churn metric, then we have a measure of the degree to which the code itself changes from version to version. Because the number of days between releases is not consistent, knowing the churn value tells us nothing about the *rate* of churn. In order to compare churn between versions, our comparison must be sensitive not only to the quantity of churn, but the number of days between releases. By measuring the *churn rate*, or churn per day between releases, we can quantify and compare the rate of code change (in lines per day). Figure 7.1 shows the churn rate for the 2.00.x series, along with indicators showing the 2.01.x development period, and the release of 2.02.000.

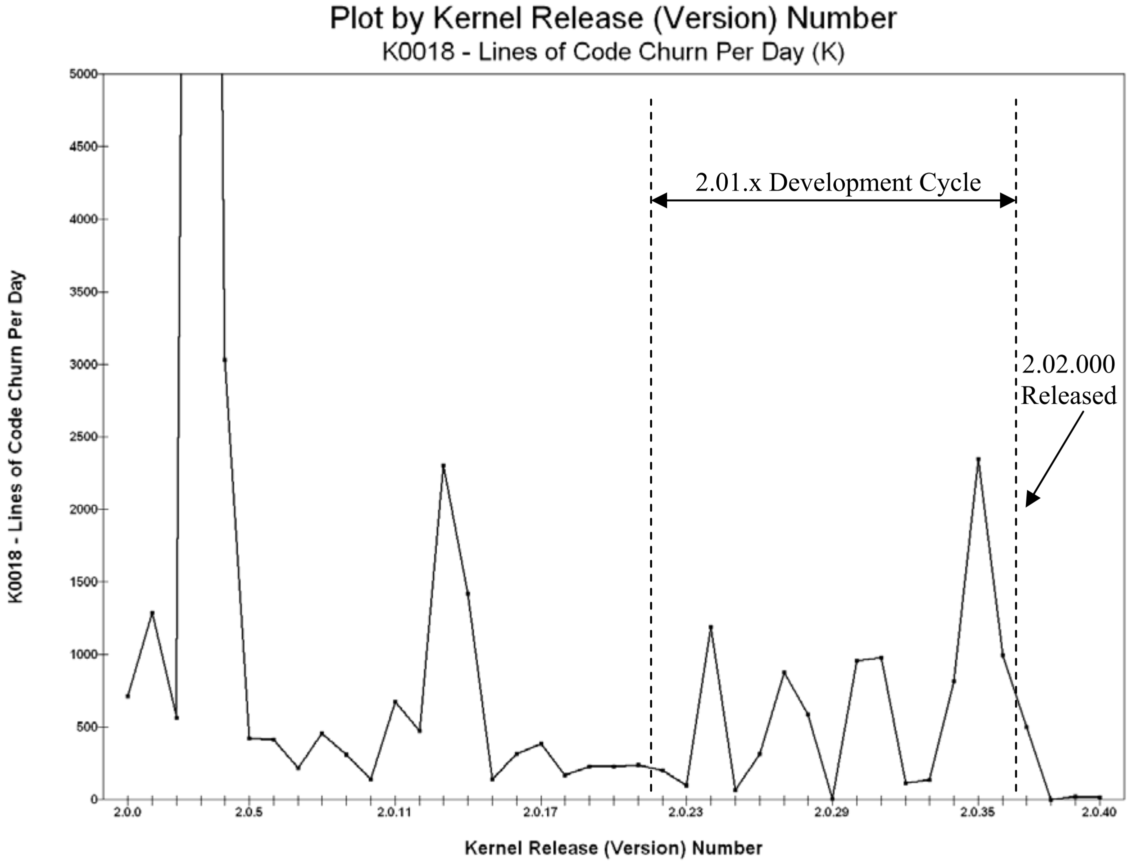


**Figure 7.1 – Churn rate for 2.00.x, plotted by version number.**

The tremendous churn in version 2.00.003 (over 20,000 lines per day) was apparently due to an overhaul of the sound subsystem. With the exception of the spike in the churn rate at version 2.00.13 (the source of which is not clear), the churn rate for 2.00.x precisely follows the general pattern described in Section 7.2 – a flurry of initial development activity, followed by a relatively consistent, somewhat lower level of activity, and, upon initiation of the next development cycle, bursts of relatively large levels of development activity. For a single version (2.00.037) following the release of the first version in the new stable series (2.02.000), there is still a moderate level of activity, after which the development activity essentially stops (versions 2.00.038 – 2.00.040).

Plotting these values with respect to version number is helpful for visualizing some relationships, but can also obscure others, because the release frequency was not consistent. Figure 7.2 shows the same data as 8.001, but plotted with respect to release date.

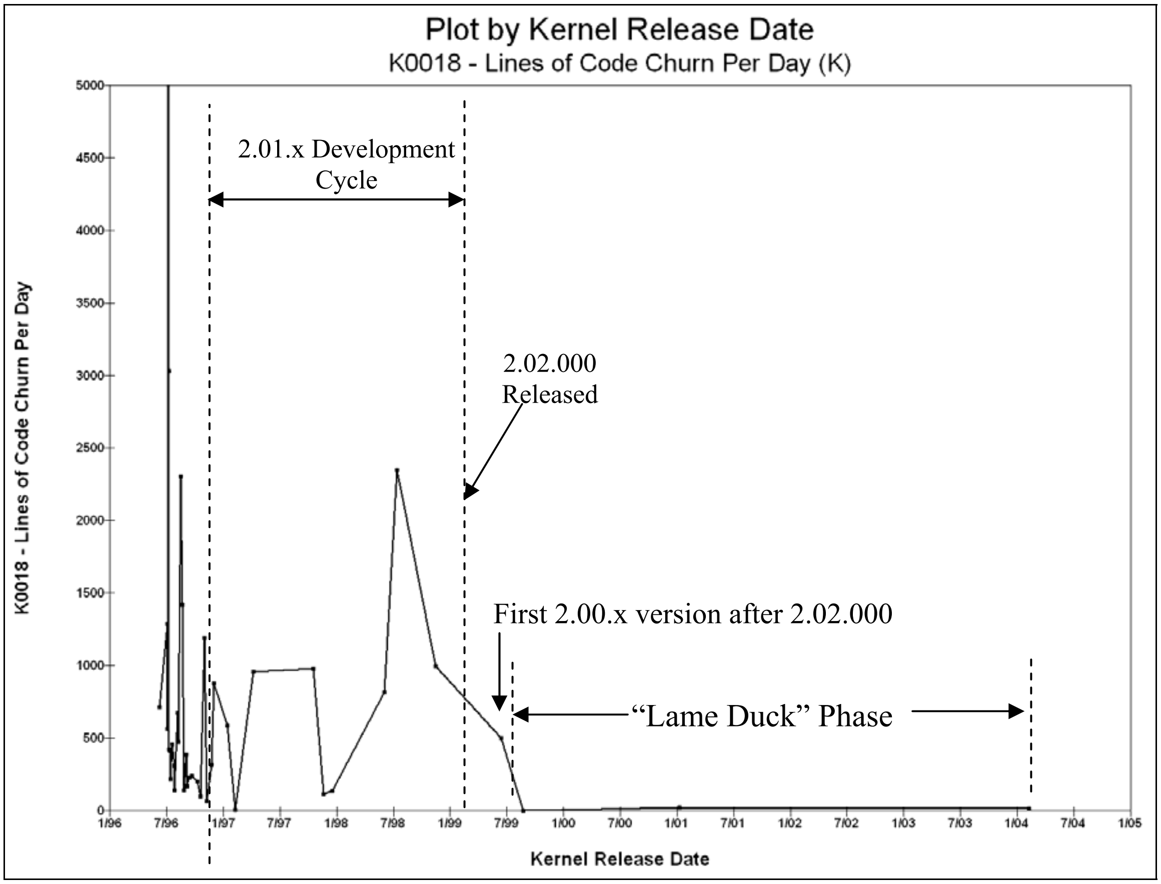


**Figure 7.2 – Churn rate for 2.00.x, plotted by release date.**

Figure 7.2 shows that the overwhelming majority of the duration of the life cycle of the 2.00.x branch was the “lame duck” phase, during which there was essentially no development taking place. If we were to use these last three data points in a time-based analysis of the code base, we would distort any trending information we consider, because the last three points are essentially duplicates of each other, although they are chronologically spaced wider than any other points in the series. Figure 7.2 confirms that our inclusion of one point past the release of 2.02.000 results in the capture of the vast majority of the total development of the 2.00.x series, without including the "lame duck" points.

We next consider the code churn in the 2.02.x kernel series, shown in Figure 7.3 plotted with respect to version number, superimposed with indicators of the 2.03.x development cycle, and the release of 2.04.000.

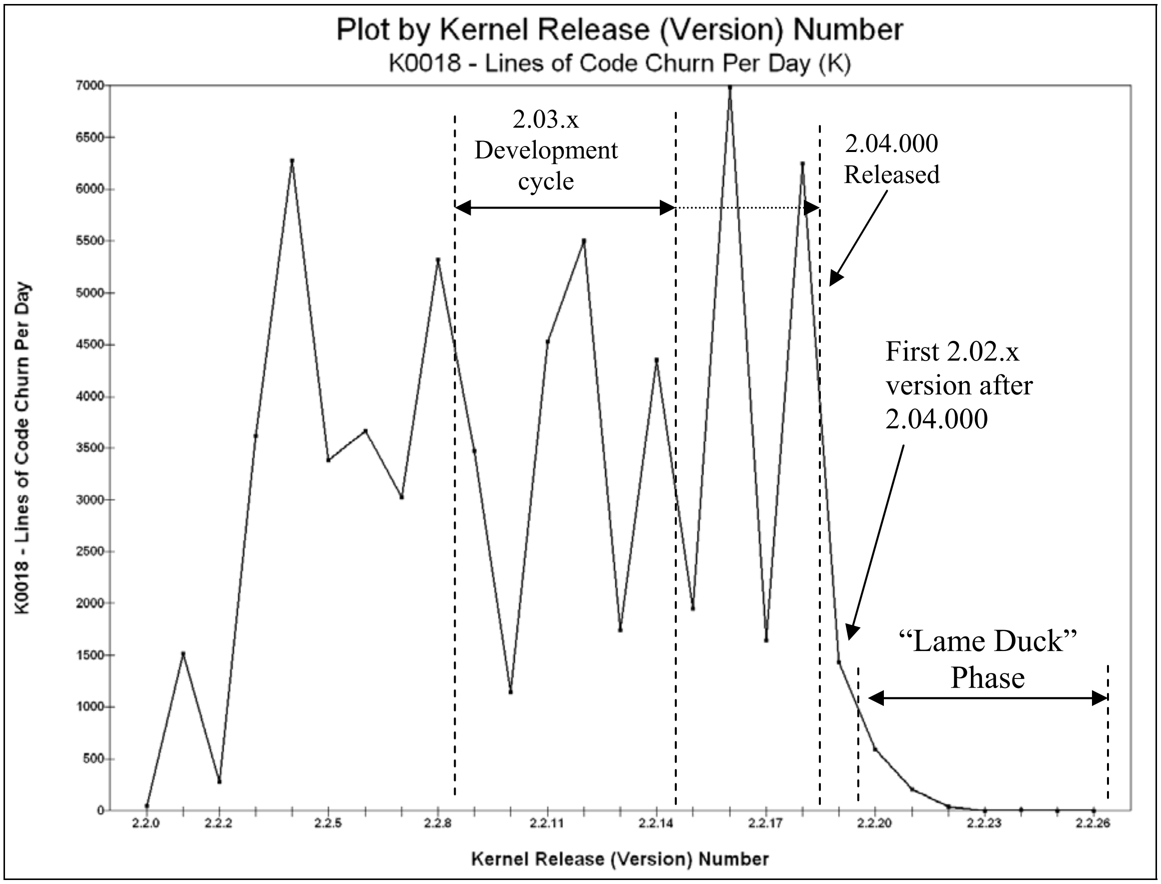


**Figure 7.3 – Churn rate for 2.02.x, plotted by version number.**

In this case, the same basic model holds, except that the initial flurry of activity led directly into the 2.03.x development cycle, which began only fifteen weeks after the release of 2.00.000 (as reflected in Figure 7.4, which shows the same data, but plotted with respect to release date, rather than version number).

In the 2.00.x series of stable kernels, the last release of the 2.01.x development cycle and the release of 2.02.000 fell between two consecutive releases of the 2.00.x kernel, so "the development cycle" was shown as a single period in Figures 7.1 and 7.2. In the 2.02.x and 2.04.x kernels, however, there was a gap between the last release of the corresponding development kernels (2.03.x and 2.05.x, respectively) and the first release of the new stable series. We consider this gap to be an extension of the development cycle, and have shown this extension for 2.02.x with a dotted arrow in Figure 7.3. We will continue to use this form of identification of the extended development cycle, which culminates in the release of the first version of the new stable cycle, in the 2.02.x and 2.04.x graphs.

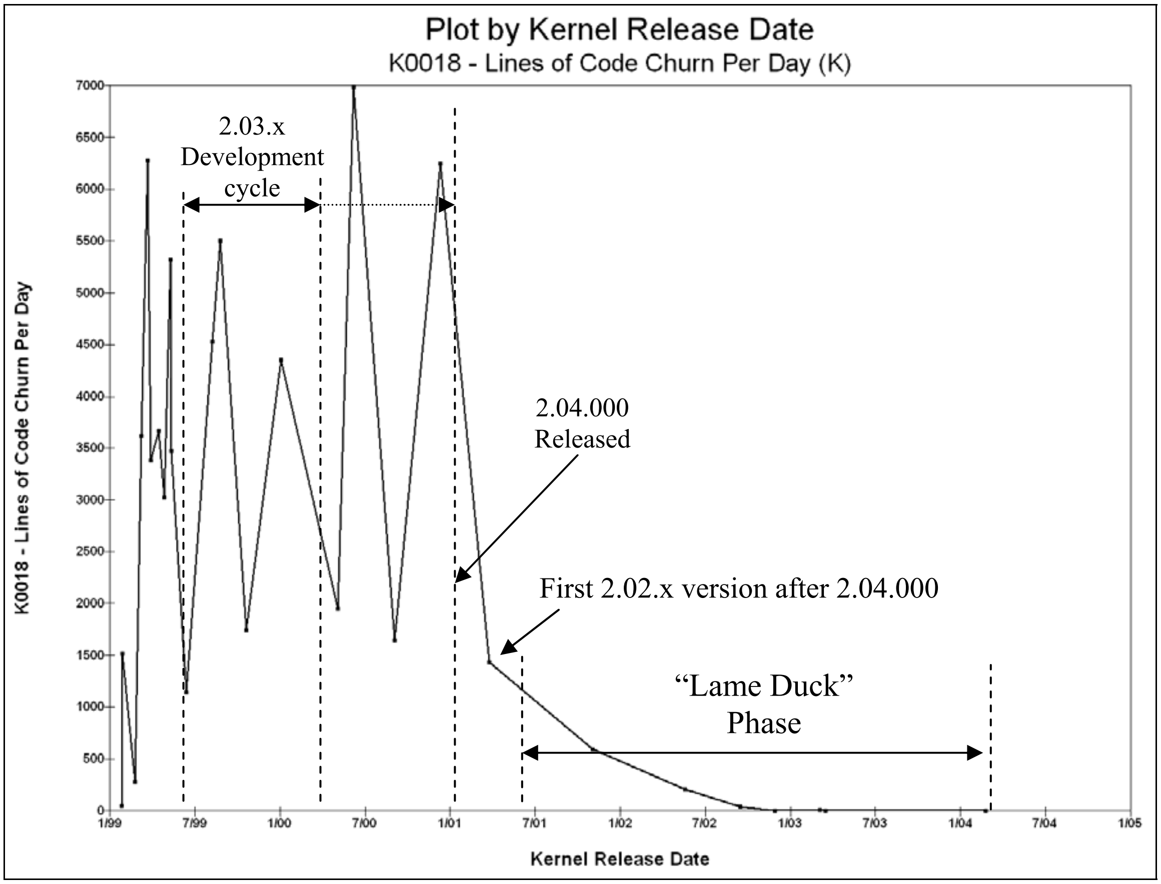


**Figure 7.4 – Churn rate for 2.02.x, plotted by release date.**

Returning to Figure 7.4, we see that it shows a similar breakdown as Figure 7.2 – a relatively brief initial period, increased development activity during the following development cycle, and a protracted "lame duck" phase beginning after the first release after 2.04.000. Figure 7.4 confirms that our inclusion of one point past the release of 2.04.000 results in the capture of the vast majority of the total development of the 2.02.x series, without including too many of the "lame duck" points.

We now repeat this process for the 2.04.x kernels. Figure 7.5 shows the churn rate for the 2.04.x kernels, plotted by version number.[3]

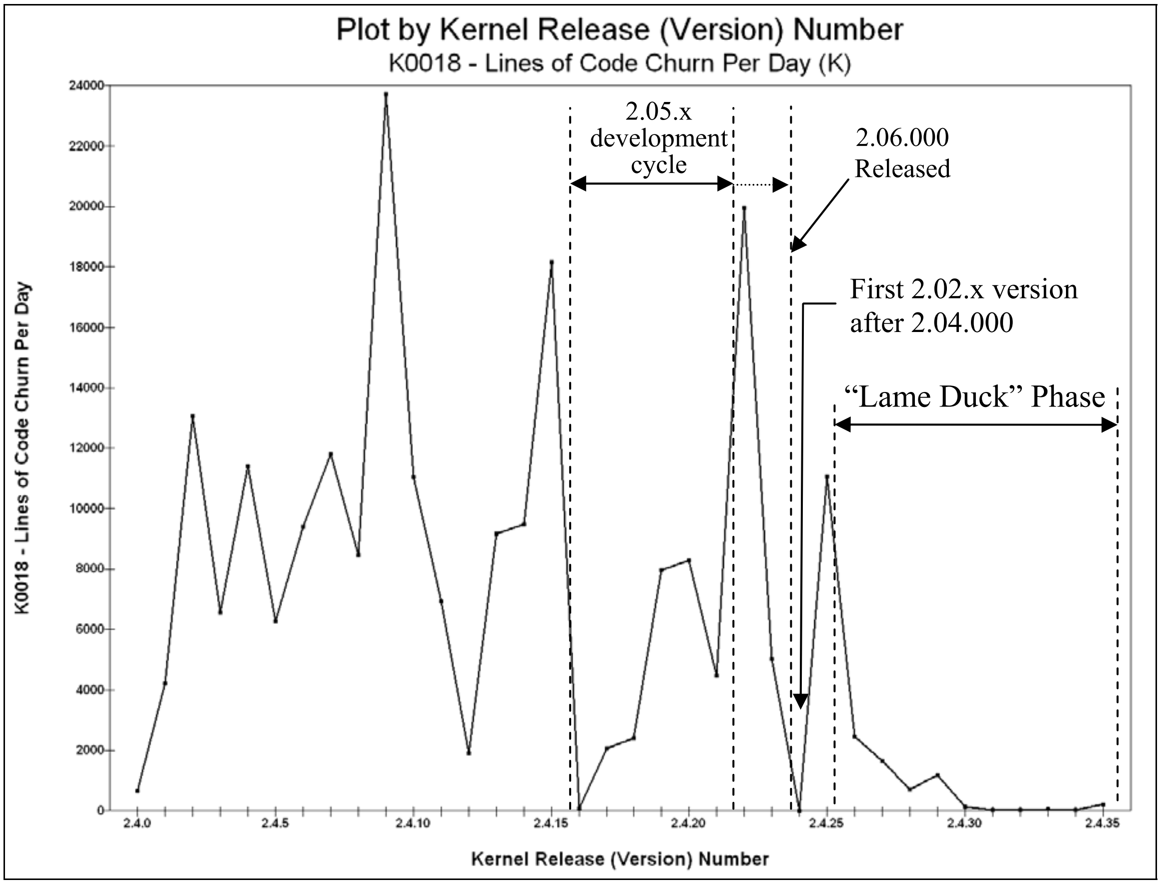


**Figure 7.5 – Churn rate for 2.04.x, plotted by version number.**

Figure 7.5 seems to indicate that the initial period and quantity of activity (prior to the initiation of the 2.05.x development series) strongly dominates the activity of the

[3] Churn values were not available for version 2.04.015. We estimated this value by dividing the churn values from the other 2.04.x kernels by the sizes of the respective patch files (the patch files *implement* the changes that the churn metric *counts*, and the patch files *were* available). This gave us a churn-per-byte-of-patch-file measure, *Q*. We then averaged the *Q* value for the 2.04.x patch files closest in size to the patch file for 2.04.015, and used this value to estimate the churn value for 2.04.015 as 327,200 lines. Version 2.04.015 was released 18 days after 2.04.014, resulting in an estimated churn per day value of 18,177.8 for 2.04.015.

development cycle. As we shall see in Figure 7.6, this is another instance of the by-version-number graph not telling the whole story.

Normally, we would expect to see increased development activity at the beginning of a development cycle, but in this case, it appears that development came to a standstill as the development cycle began (version 2.04.16). The 2.05.x development series began with 2.05.000, which was identical to 2.04.015 (except, of course, for the version number constants in the `Makefile`). Unfortunately, a serious fault was introduced in the last pre-release version of the code for 2.04.015, so 2.04.16 was released three days later, with a very low churn value (~260). Because the changes were narrowly focused on rectifying this single fault, and because the time between releases was so short, the result was a low churn rate for that one release.

Figure 7.6 shows the same data, plotted by release date, rather than by version number. Because versions 2.04.015 and 2.04.016 were released in such close temporal proximity, we have drawn the dashed line to indicate the start of the 2.05.x development cycle with a break across the solid graph line, so as to not obscure the underlying plot.

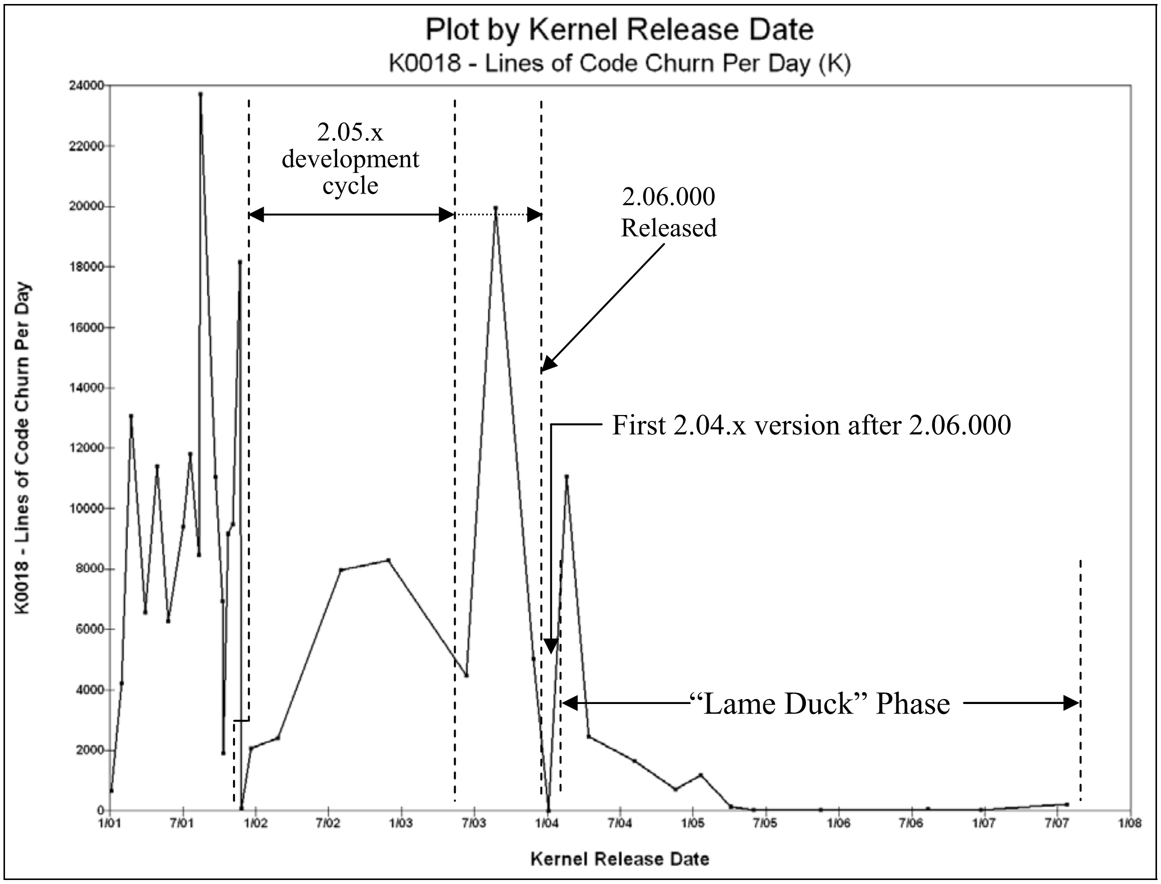


**Figure 7.6 – Churn rate for 2.04.x, plotted by release date.**

Displaying the churn per day with respect to release date shows that the development period was longer in comparison to the period that preceded it than it first appeared in Figure 7.5.

Figure 7.6 shows that, after the 2.06.x series debuted, the rate of development certainly slowed when compared to the pre-2.06.x development rate, but not nearly to the extent observed in 2.00.x and 2.02.x (compare Figures 7.2 and 7.4 to 7.6).

The lame duck phase in 2.04.x began with a higher level of development activity than in 2.02.x, which began with a higher level of development activity than the corresponding period of the 2.00.x series. We believe this is due to the fact that as Linux had continued to become a mature operating system, more and more users, particularly on server platforms, were continuing to run the 2.04.x kernels, rather than upgrade to 2.06.x. This protracted "linger" period would have kept demand for 2.04.x development higher than might have otherwise been the case.

Nevertheless, our initial assertion, namely, that placing the cutoff at one 2.04.x version past the release of 2.06.000, results in the capture of the vast majority of the 2.04.x development effort, without including too much of the "lame duck" phase, is confirmed by Figure 7.6. In the case of 2.04.x, the inclusion of an extra data point would have captured a bit more of the development effort, but making such an exception would have been a purely arbitrary (and therefore unjustifiable) decision.

Figures 7.2, 7.4, and 7.6 all show that, at the end of the life cycle for a stable kernel series, there are multiple data points for which there is virtually no development activity. Moreover, these data points tend to be spread so far out along the time axis that they obscure the underlying trend information by essentially introducing copies of existing data points, with grossly disproportionate displacements in time. These data points represent kernel versions of questionable relevance, given their "lame duck" status, and need to be excluded from the remainder of our analysis.

We have already examined the code churn rate values for the successive versions, but for the determination of the total development effort, we need the churn values themselves, rather than the churn *rates*, which we used above.

For each version of each stable kernel series, we computed the cumulative development effort, as measured by the total churn to date, as a percentage of the total eventual churn. Obviously, the cumulative churn value for the last version of a series would be 100 percent.

Figures 7.7, 7.8, and 7.9 show the cumulative churn distribution for the 2.00.x, 2.02.x, and 2.04.x series, respectively, along with indicators showing our cutoff points, and the percentage of the total development effort (as measured by churn) captured in our selected versions.

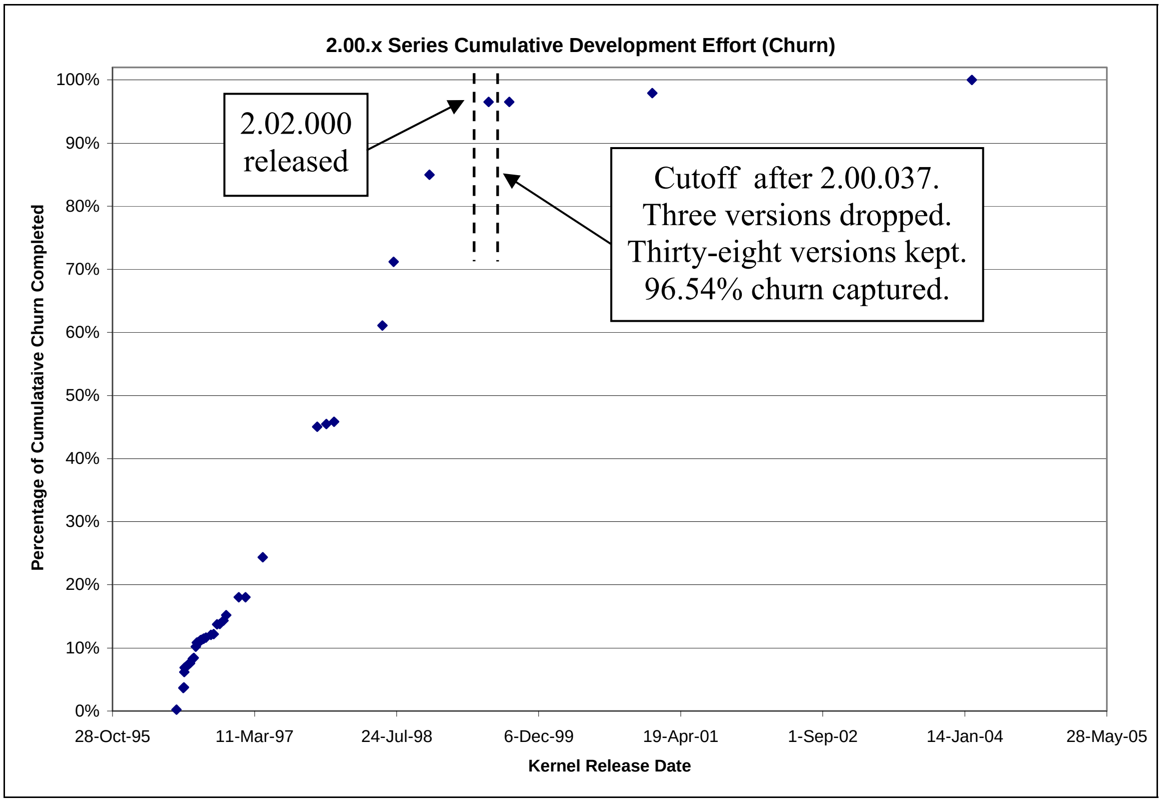


**Figure 7.7 – Cumulative churn percentages for the 2.00.x series of kernels.**

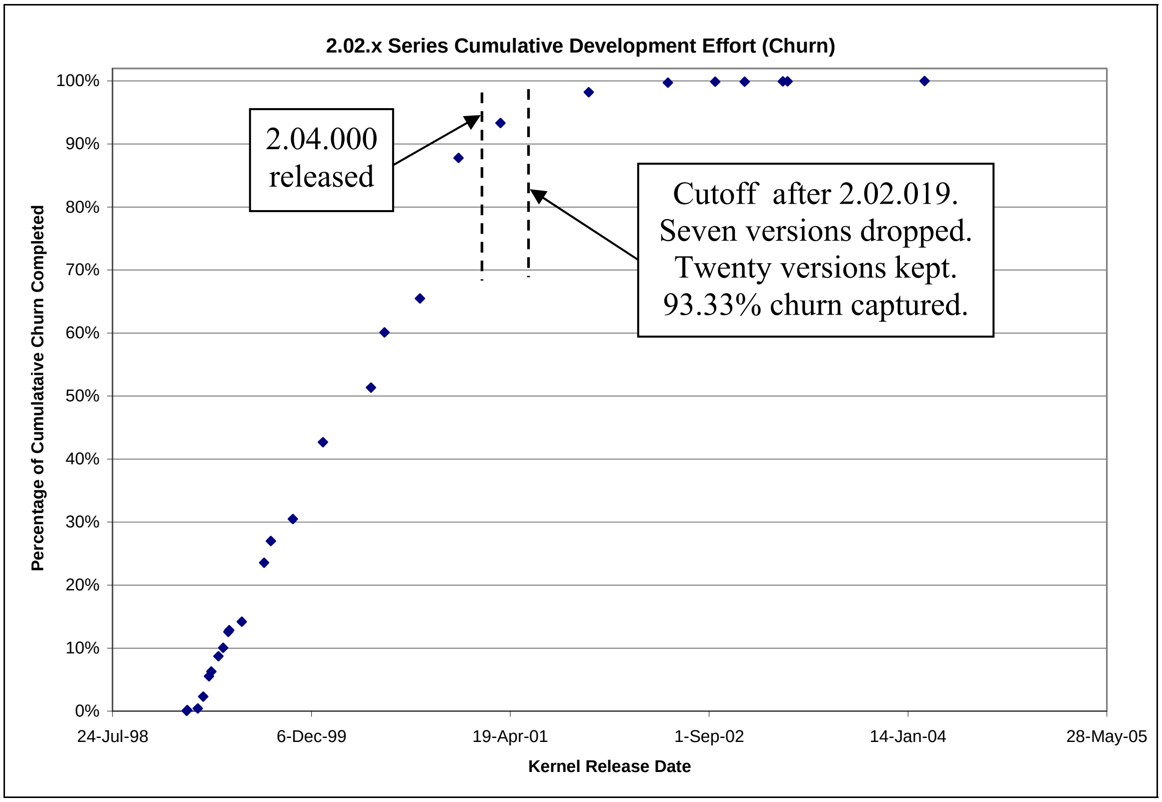


**Figure 7.8 – Cumulative churn percentages for the 2.02.x series of kernels.**

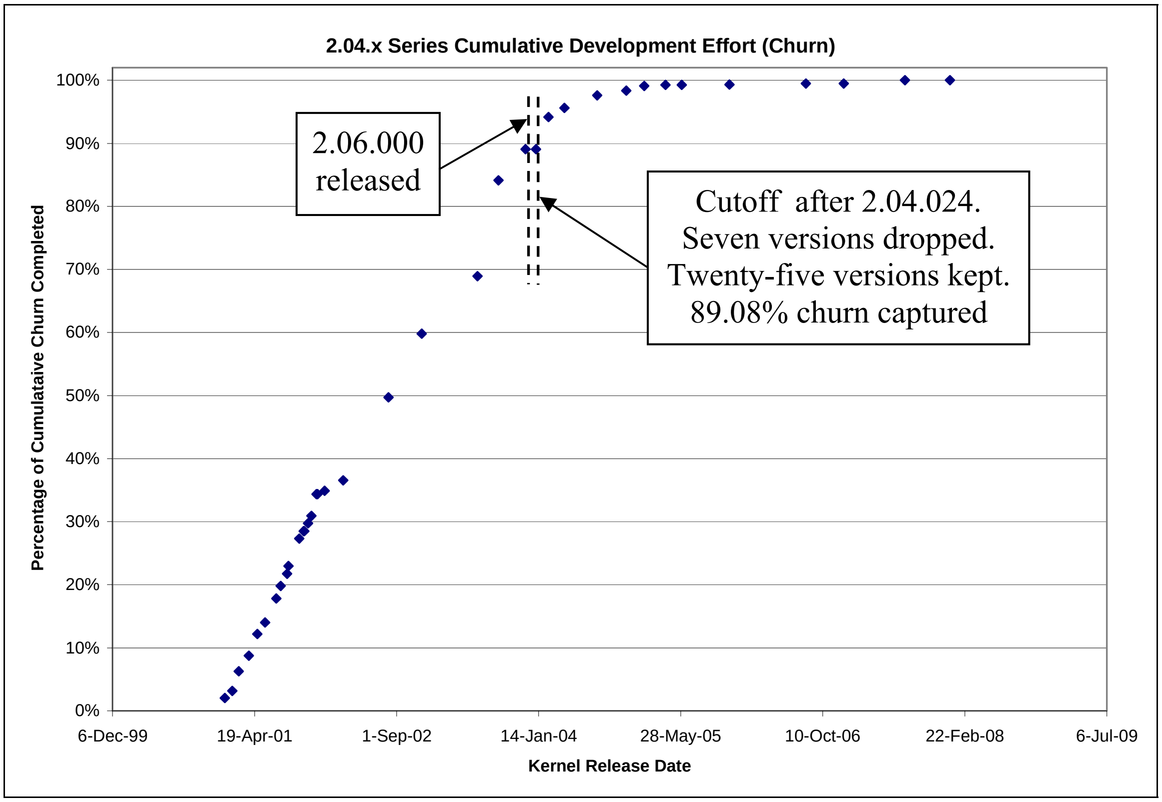


**Figure 7.9 – Cumulative churn percentages for the 2.04.x series of kernels.**

Visual inspection of Figures 7.7 through 7.9 reveals strong similarities between the three series. Each of these graphs has a decidedly linear rise in the early part of the series' life cycle, and a decidedly linear horizontal portion at the end of the series' life cycle. The fundamental differences in the three lie in the transition between the linear rise and the linear end. In 2.00.x, the transition is quite sharp. In versions 2.02.x and 2.04.x, the transition is decidedly more gradual. This may be due to the fact that, as Linux continued to mature as an operating system, more and more systems depended on Linux for operation. Consequently, continuing to run a version from the older series, rather than upgrading to the new, unproven series, may have been a more attractive option as Linux matured from 2.00.x to 2.02.x and 2.04.x.

### 7.7 Kernel Series 2.00.x, 2.02.x, and 2.04.x – One Longitudinal Study or Three?

At no time during our analysis of the three stable series did we consider each of them to be a different version of the "same" piece of software. On the contrary, even though they are all Linux kernels, they are three distinct stable series, and each series is not simply the result of evolutionary changes applied to the last version of the previous stable series. For example, consider the case of the 2.02.x series. At some time during the life cycle of the 2.00.x stable series, the 2.01.x development series began, and considerable effort was invested in the evolutionary changes in the consecutive development versions. Along the way, features in the development kernels were integrated back into the 2.00.x series. There came a point, however, at which new features were fundamentally incompatible with the underlying foundations of the 2.00.x kernels, and those features could not be "bolted onto" the 2.00.x kernels. At this point, the 2.01.x development kernels evolved into stable kernel 2.02.000. We emphasize that 2.02.000 was not an evolutionary development from 2.00.036; it was a fundamentally new kernel. Put another way, there was *evolutionary* development within a kernel series, but *revolutionary* development between the end of one stable series at the start of a new one.

Our research is restricted to the evolutionary changes in the Linux kernel, and we therefore confine our analysis to a single series at a time. In particular, correlations between various metrics must be performed on a single series at a time. Consider the following hypothetical example. The values in Figure 7.10 have zero correlation.

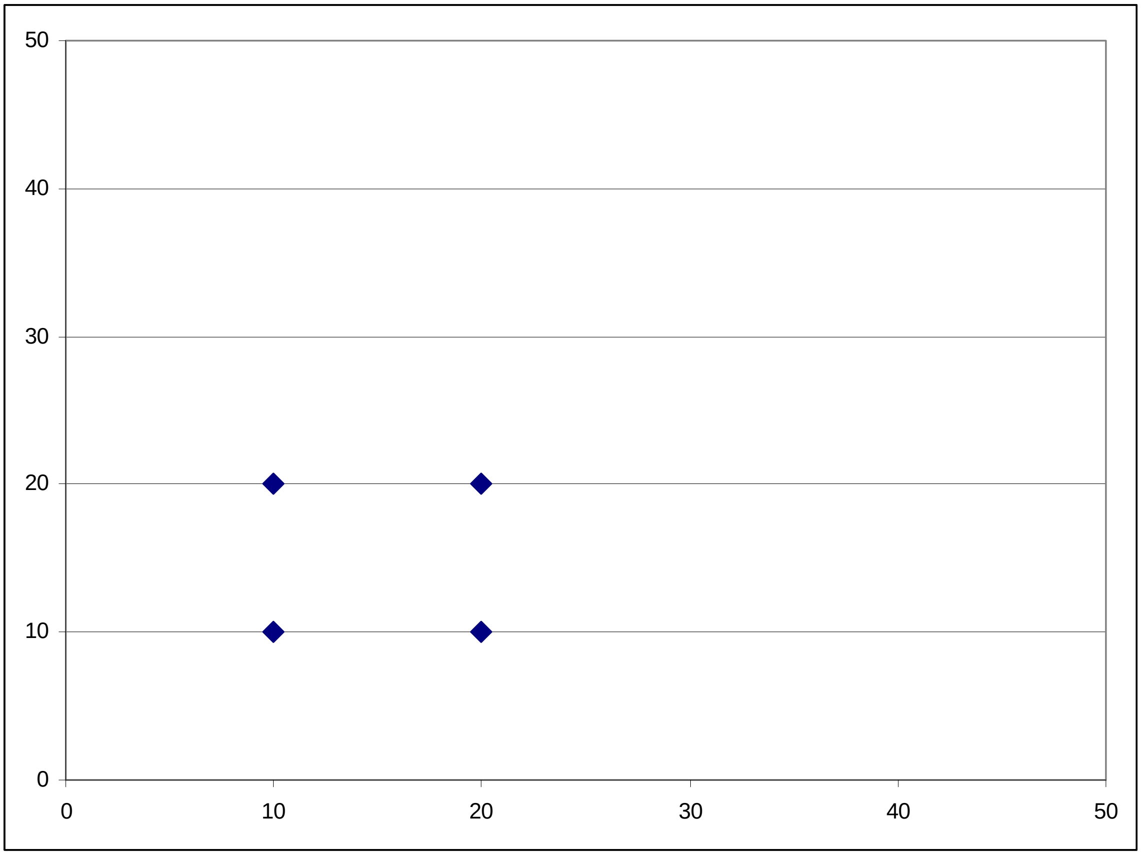


**Figure 7.10 – Four data points with zero correlation taken from some hypothetical entity.**

Now let us assume that the entity with the four data points in Figure 7.10 undergoes some sort of transformation, and the four-point cluster is shifted along the X and Y axes, at which point we re-measure the transformed entity and obtain four more data points that exhibit no correlation, and that the process repeats again. An example of the resulting data is shown in Figure 7.11.

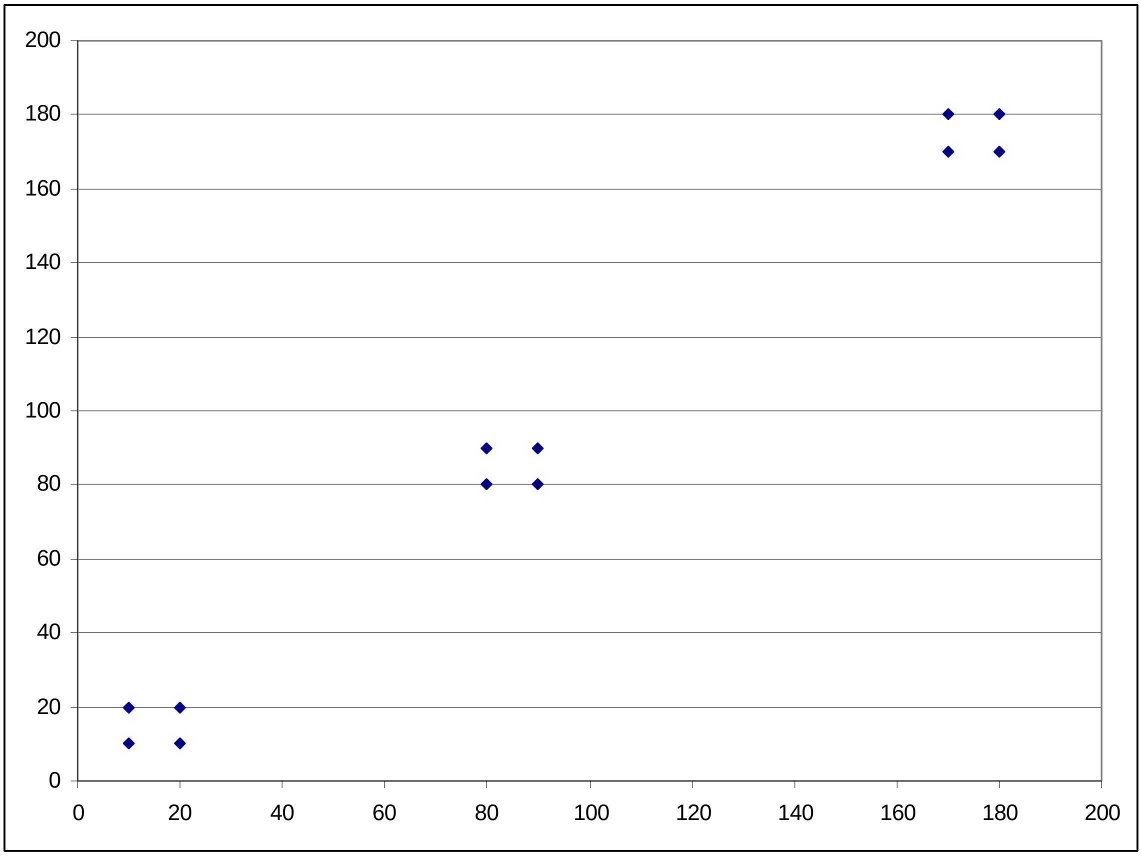


**Figure 7.11 – Three sets of uncorrelated points taken from three generations of some entity.**

We now have three clusters of data points, not from a single entity, but from three different entities, because the entity was transformed between measurements. Because the three four-point clusters are displaced so far along the X- and Y-axes relative to their within-cluster variation, a correlation analysis performed on the three generations at once would show an extremely strong correlation when, in fact, there is none. For this reason, we consider each of the three stable series of the Linux kernel independently, rather than as one long series of successive releases of the same series.

An absurd, contrived example to reinforce the point would be to measure the ages and heights of a group of dwarfs aged 20-30 and a group of former NBA centers between 80 and 90 years of age. Drawing a correlation between the two groups, we would conclude that age and height are strongly correlated, and we would (incorrectly) predict that people continue to grow in height through their adult years. The problem, of course, is that we have drawn a correlation between two clusters of data points taken from two different populations, with X and Y (age and height) variations *between* groups much larger than the age and height variations *within* the respective groups.

Similarly, we examine metrics that vary much less within the three kernel series than they do between series, simply because the series are three distinct entities, and not just multiple samples of the same entity.

In order to support our claim that the three series are different entities, we performed a Wilcoxon/Mann-Whitney U-test on the sizes of the code base (see Section 7.8) in 2.00.x and 2.02.x series, and determined with exceedingly high significance (P<<0.00001) that the 2.00.x and 2.02.x series represented different populations, rather than two sets of samples taken from the same population. Repeating this test for the 2.02.x and 2.04.x series produced similar results (again, P<<0.000001). Finally, we compared the 2.00.x and 2.04.x series, and received the same result. Repeating the test on the common coupling values (see Section 7.10) of the three series yielded the same results, statistically confirming our claim that the three series are three different entities and must be analyzed separately.

Because the three kernel series are three independent entities, in the rest of this dissertation we consider them individually.

## 7.8 Measuring the Size of the Linux Kernel

As we discussed in Section 1.1, “size” is a nebulous term, and can be measured in multiple ways. When we think of the “size” of a program, we may consider the number of lines in the source file, the total number of bytes in the source file, or even (for programs comprised of multiple modules) the number of source files themselves. Alternately, if we were concerned with run-time issues, we might consider the size of the executable file on disk, or even the memory footprint of the code as it runs (which may even be dynamic).

Schach et al. measured the size of the files in the `/linux/` subdirectory by counting the total number of lines in the source (`.c`) files. Consider, for example, the file `/linux/info.c`, which Schach et al. counted as having 39 lines of code (LOC) in kernel version 2.02.000. This file begins with five `#include` directives:

```
#include <linux/mm.h>
#include <linux/unistd.h>
#include <linux/swap.h>
#include <linux/smp_lock.h>
#include <asm/uaccess.h>
```

These header files, in turn, `#include` *other* header files. When the `#include` directives are all evaluated by the preprocessor (which also includes macro expansion and the excluding of some lines via conditional compilation), this 39-line file becomes 16,190 lines long, and the total file size mushrooms from 734 to 340,305 bytes, a 415-fold increase in the number of lines, and a 463-fold increase in the number of bytes.

Because of the magnitude of the impact that preprocessing has on the code base presented to the compiler for parsing into executable code, we consider only the preprocessed code for the files actually used in the process of building each respective kernel. That is, we consider only our kernel configurations (see Chapter 5), not the entire code base.

Once we elect to consider only the preprocessed code for each kernel, we still have multiple ways to evaluate the size of that code. We could count the total number of lines or bytes, as above. We could also elect to exclude the comment lines, under the argument that comment lines contribute nothing (directly) to the executable code ultimately generated.

Figure 7.12 shows the percentage of lines containing comments across all files in each kernel version in each series. The plotting of all three series on the same graph is a convenience to show the behavior of the three series relative to one another, but we emphasize our findings from Section 7.7 that the three series are independent entities and cannot be considered collectively.

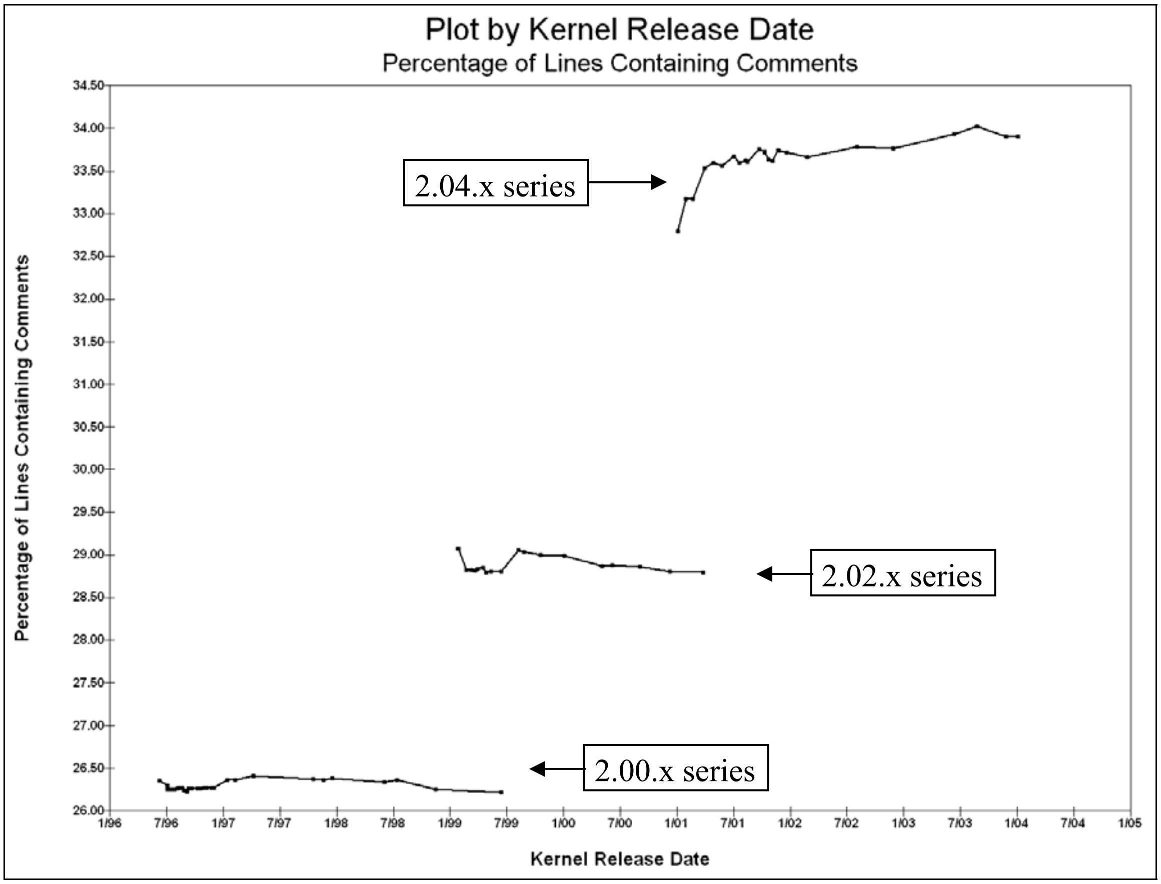


**Figure 7.12 – Percentage of comment lines in the 2.00.x, 2.02.x, and 2.04.x series of kernels.**

The Y-axis of Figure 7.12 shows the variability in the percentage of lines with comments *within* a kernel series was much less than the variability *between* series. The averages for the 2.00.x, 2.02.x, and 2.04.x series were approximately 26.28, 28.89, and 33.65 percent, respectively.

Just as comment lines contribute nothing (directly) to the executable code, C is white-space blind, and, as such, blank lines contribute nothing to the executable code. Moreover, the expansion of the nested header files through the preprocessor `#include` directive introduces an exceedingly large number of blank lines. In the preprocessed version of our sample file, `/kernel/info.c`, 76 of the first 88 lines are blank. Figure 7.13 shows the percentage of blank lines in each kernel version in each series.

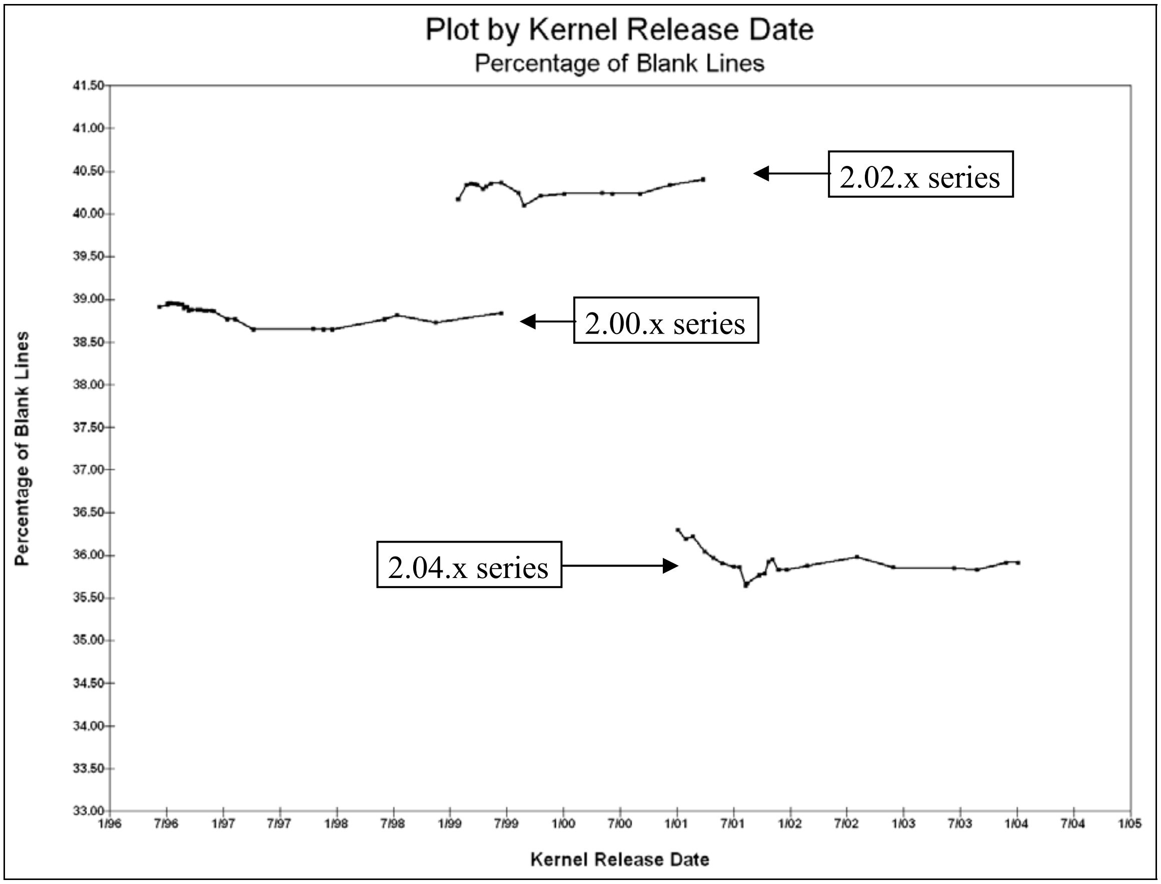


**Figure 7.13 – Percentage of blank lines in the 2.00.x, 2.02.x, and 2.04.x series of kernels.**

Figures 7.12 and 7.13 are both plotted with an 8.5 percent range in the Y-axis, facilitating visual comparisons. Curiously, the percentage of comment-bearing lines rose in all three series; however, the percentage of blank lines rose between the 2.00.x and 2.02.x series, and then *fell* sharply between the 2.02.x and 2.04.x series. Just as in Figure 7.12, the within-series variability for the percentages was considerably less than the between-series variability. The overall averages for the 2.00.x, 2.02.x, and 2.04.x series were 38.88, 40.28, and 35.90 percent, respectively.

Because neither blank lines nor comment lines actually contribute to the executable code, we will consider only Non-Comment, Non-Blank lines, or NCNB lines. The percentage of NCNB lines, relative to the total file line count is shown in Figure 7.14.

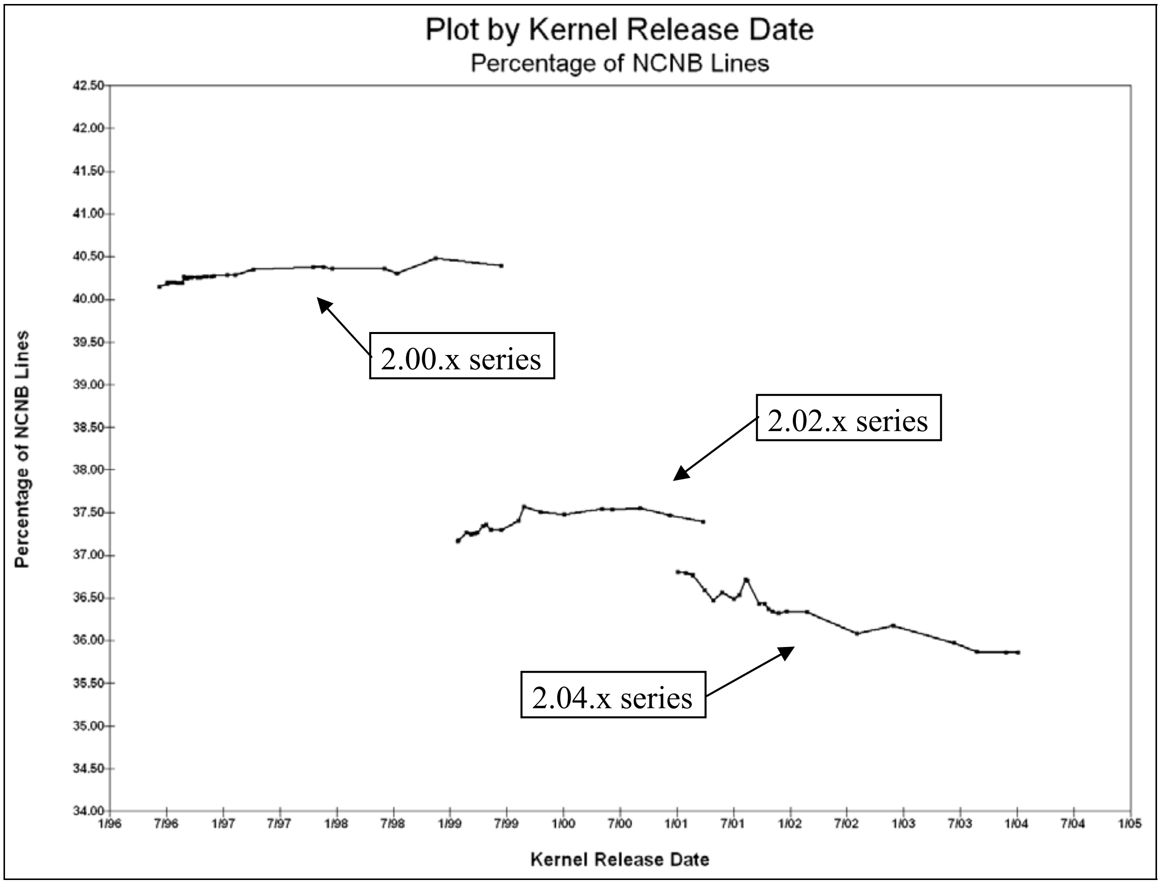


**Figure 7.14 – Percentage of NCNB lines in the 2.00.x, 2.02.x, and 2.04.x series of kernels.**

Figure 7.14 is also drawn with a Y-axis spanning an 8.5 percent range. Overall, the percentage of NCNB lines dropped much more between the 2.00.x series and the 2.02.x series than it did between the 2.02.x series and the 2.04.x series. The overall averages for the percentage of NCNB lines in the 2.00.x, 2.02.x, and 2.04.x series were 40.26, 37.37, and 36.38 percent, respectively.

Although we have narrowed our definition of "size" to mean "the number of NCNB lines," we could further classify the NCNB lines within a file as being within a function or outside all functions. We consider common coupling, which occurs only when code in two or more files refers to the same global variable. Furthermore, the only code that can refer to the contents of a global variable is executable code, and executable code is found only in functions. Accordingly, we might be tempted to consider only the NCNB lines within functions, rather than the function lines outside of all functions. There are three problems with that approach:

1) Not all lines of code within functions are executable – functions can (and do) include declarations, and we do not have a count of the number of lines of declarations. Moreover, one or many local variables can be declared on a single declaration line.
2) The *entire* code base must be maintained. The lines outside of all functions, although not executable, are nevertheless important, and require maintenance. Many of these lines, when considered across all files within a kernel, appear multiple times (because they exist in an `#include` file that gets preprocessed multiple times). As such, the number of distinct lines will necessarily be less than the total number of lines in all files. Therefore, our line counts represent an upper bound on the volume of code to be maintained.
3) This chapter represents an extension of Schach et al.'s work regarding maintainability of the Linux kernel, and Schach et al. considered the entire code base.

For all of these reasons, we use the number of NCNB lines in the entire file as the basis for measuring the size of the code base.

## 7.9 Rate of Growth of the Size of the Linux kernel

### 7.9.1 Versions 2.00.x

Schach et al. concluded that the size of the Linux kernel was growing linearly. Turning to our results, the graph in Figure 7.15 shows the sum of the number of NCNB lines in all files, plotted with respect to version number. This is the total file size, in thousands of NCNB lines, for the configured and preprocessed code base, as described in Chapter 7, taken from the files prior to our cutoff, as described in Section 7.5.

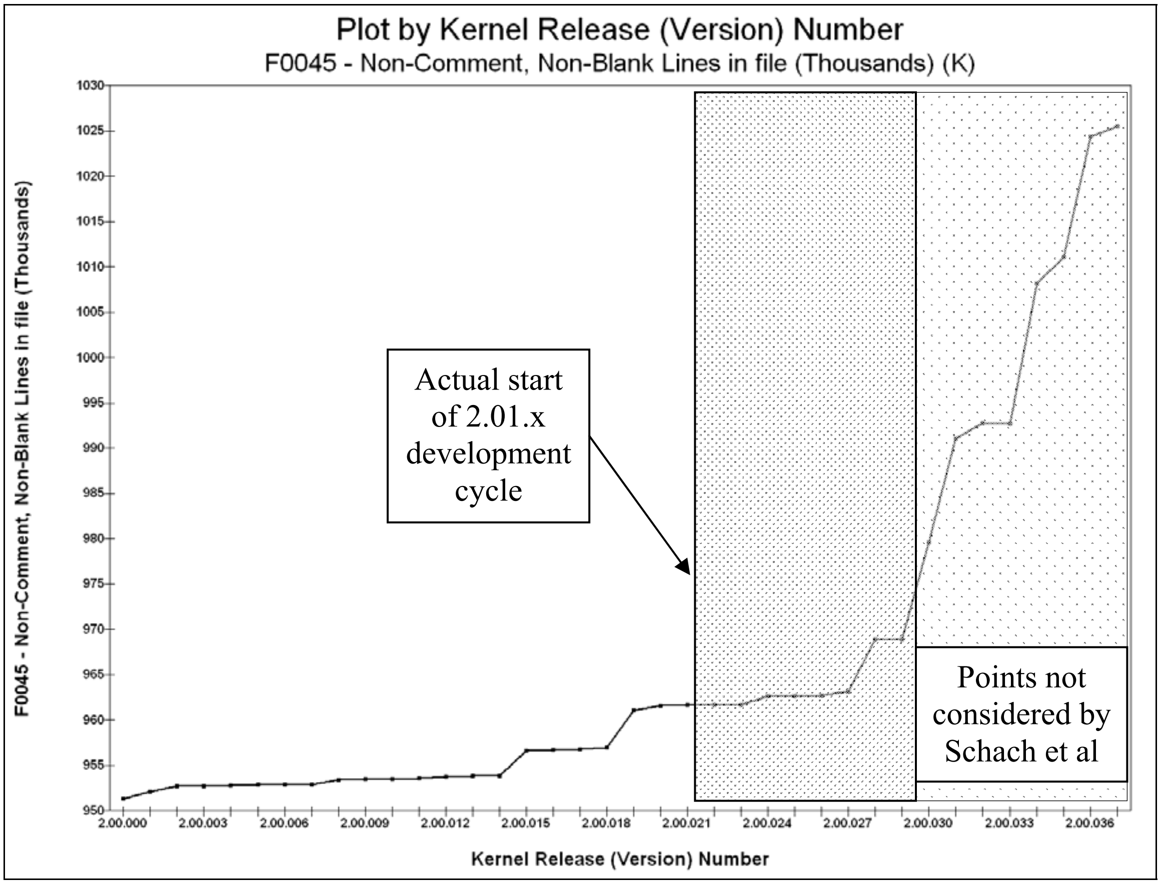


**Figure 7.15 – File Size (in thousands of NCNB lines) for 2.00.x kernels, with respect to version number**

Because of the way Schach et al. selected the versions they included in their analysis (see Section 4.3), they did not consider versions later than 2.00.030. Our selection criteria, as discussed in Section 7.3.6, excluded only versions 2.00.038 – 2.00.040. Version 2.00.040 was released after their work, but would also have been excluded under their selection criteria.

Using the selection criteria outlined in their paper, it seems that Schach et al. would also have excluded versions 2.00.021 through 2.00.029, as shown in the darker shaded area in Figure 7.15.

In the kernel versions Schach et al. examined (2.00.000 – 2.00.029), the rate of growth in the 2.00.x kernels does appear to be linear, but had they examined subsequent versions in the 2.00.x series, they might have seen the increase in versions 2.00.030 through 2.00.039.

It is also possible that, because their code base was restricted to only a subset of the files in the `/kernel/` subdirectory, that they would not have observed the same rate of increase. It could be argued that the files in the `/kernel/` subdirectory are among the least frequently modified files in the entire source tree, and that, when modification of one of those files *is* required, the modifications would tend to involve fewer lines than would be required for some other subset of the kernel code, such as for memory management, a file system, or a device driver.

Taken in its entirety, however, the rate of growth in Figure 7.15 initially appears to represent an exponential relationship. It could also be two linear relationships – one from versions 2.00.000 through 2.00.026 and another from versions 2.00.028 through 2.00.037. One possible explanation for this is that, beginning with version 2.00.028, the release frequency dropped sharply. From versions 2.00.000 through 2.00.027, the time between releases ranged from 1 to 24 days, with an average of 6.35 days. From versions 2.00.028 through 2.00.037, the time between releases ranged from 24 to 209 days, with an average of 92.4 days. The number of days between releases is shown in Figure 7.16.

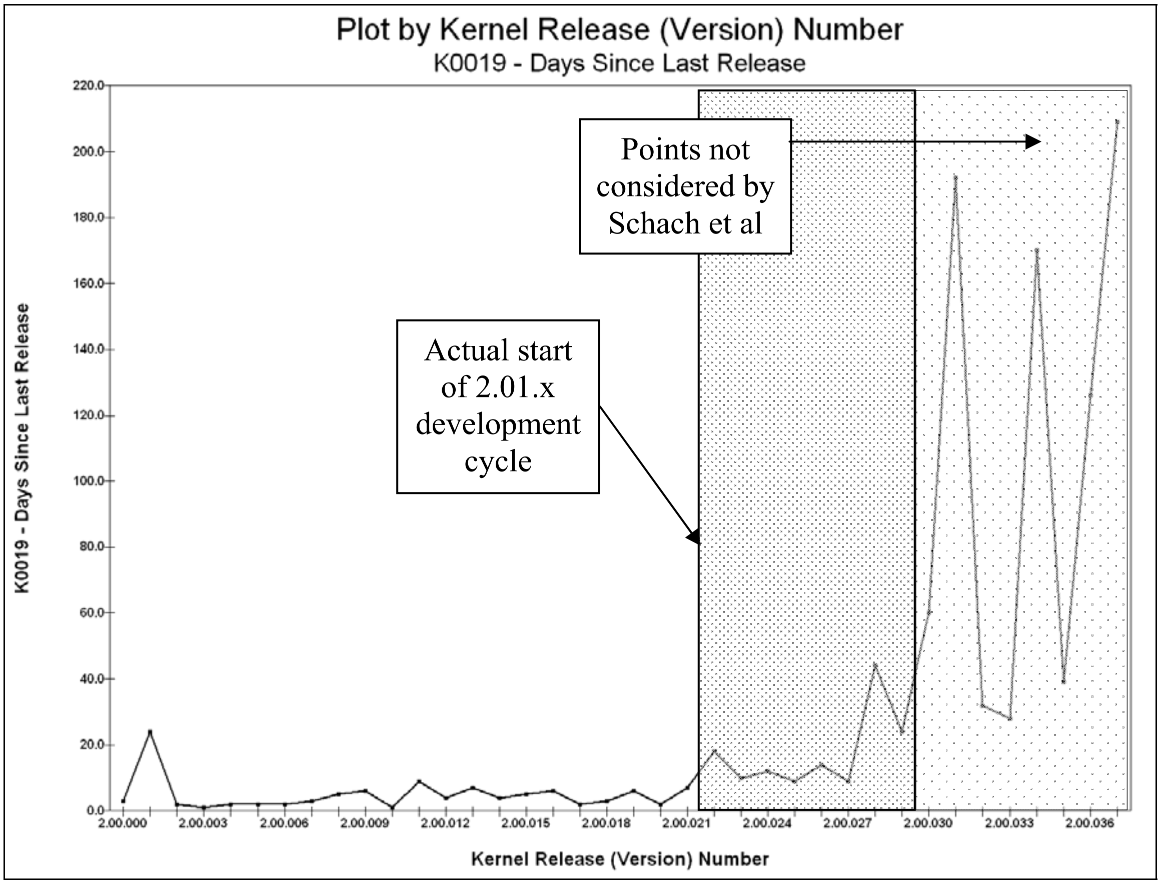


**Figure 7.16 – Number of days between releases of versions in the 2.00.x kernel series.**

Because the time between releases beginning with version 2.00.028 was consistently so much longer than the time between releases prior to version 2.00.028, the seemingly steep rise in the size of the code base in versions 2.00.030 – 2.00.037 might have simply been an extension of the initial linear growth, and the apparent steep rise could simply be a manifestation of the compressed time scale in the X-axis, and that expanding the X-axis to compensate would reveal a continued linear relationship.

The high level of inconsistency in the release dates makes discussions of the “rate of growth,” when measured against the version number, somewhat meaningless, because “rate” implies “difference per time period,” and we cannot tell what the time period between versions is just by examining the graph. This is a major part of the motivation for us to revisit Schach et al.

If we change the X-axis in Figure 7.15 from Kernel Version *Number* to Kernel Release *Date*, we get the graph shown in Figure 7.17.

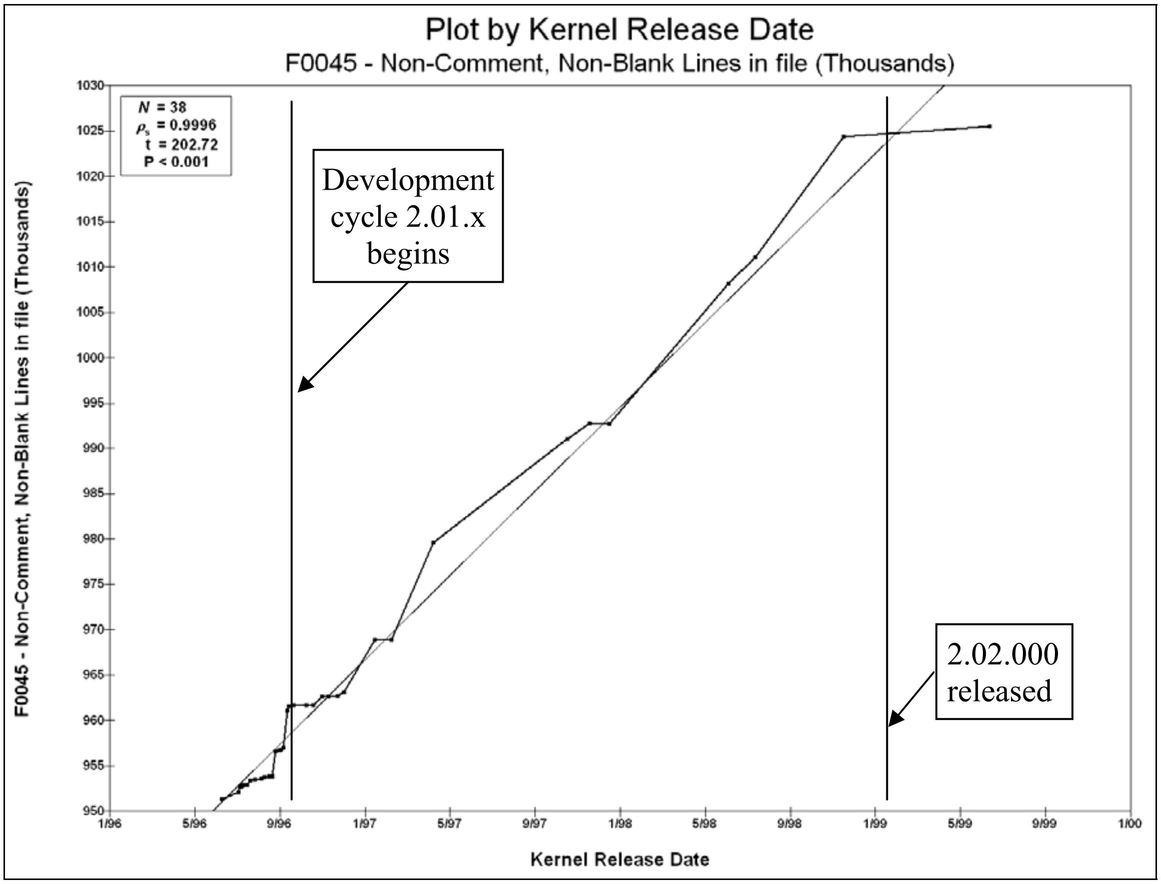


**Figure 7.17 – File size (in thousands of NCNB lines) for 2.00.x kernels, with respect to release date.**

This file size data exhibits a *much* stronger linear relationship with the release date than was shown with respect to version number. Although Figure 7.17 shows $P<0.001$, the actual value of $P$ is less than 1.0E-308, the limit of the IEEE double-precision floating point arithmetic. The rate of growth in the 2.00.x series of kernels, when considered with respect to release date, has an exceedingly highly significant linear correlation.

In summary, the 2.00.x kernels did seem to grow linearly with respect to time, at least until they reached the end of their life cycle.

### 7.9.2 Versions 2.02.x

We now turn our attention to the rate of growth of the 2.02.x kernels, first by version number, as shown in Figure 7.18:

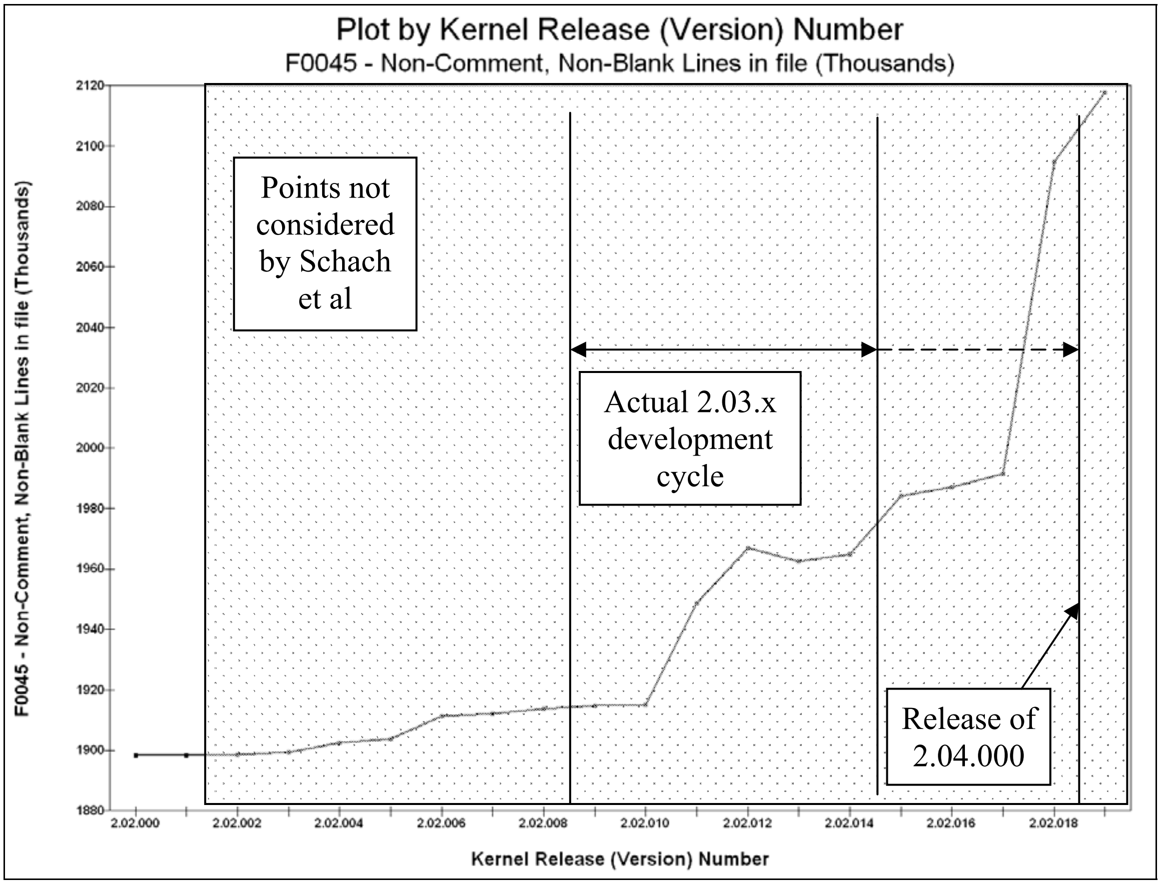


**Figure 7.18 – File Size (in thousands of NCNB lines) for 2.02.x kernels, with respect to version number**

The 2.02.x kernels exhibit the same life-cycle pattern as the 2.00.x kernels – relatively consistent growth in the early versions, until the commencement of the subsequent 2.03.x development cycle, after which the size grew at a seemingly substantially increased rate, when viewed relative to version numbers. This continued until the 2.04.000 kernel was released, at which time the development of 2.02.x slowed abruptly as the 2.02.x series entered its twilight versions.

If we plot the file sizes with respect to release date, rather than version number, we have a somewhat better linear fit. Figure 7.19 shows the file sizes with respect to the release date; the vertical lines indicate the 2.03.x release cycle and the release of 2.04.000, just as in Figure 7.18.

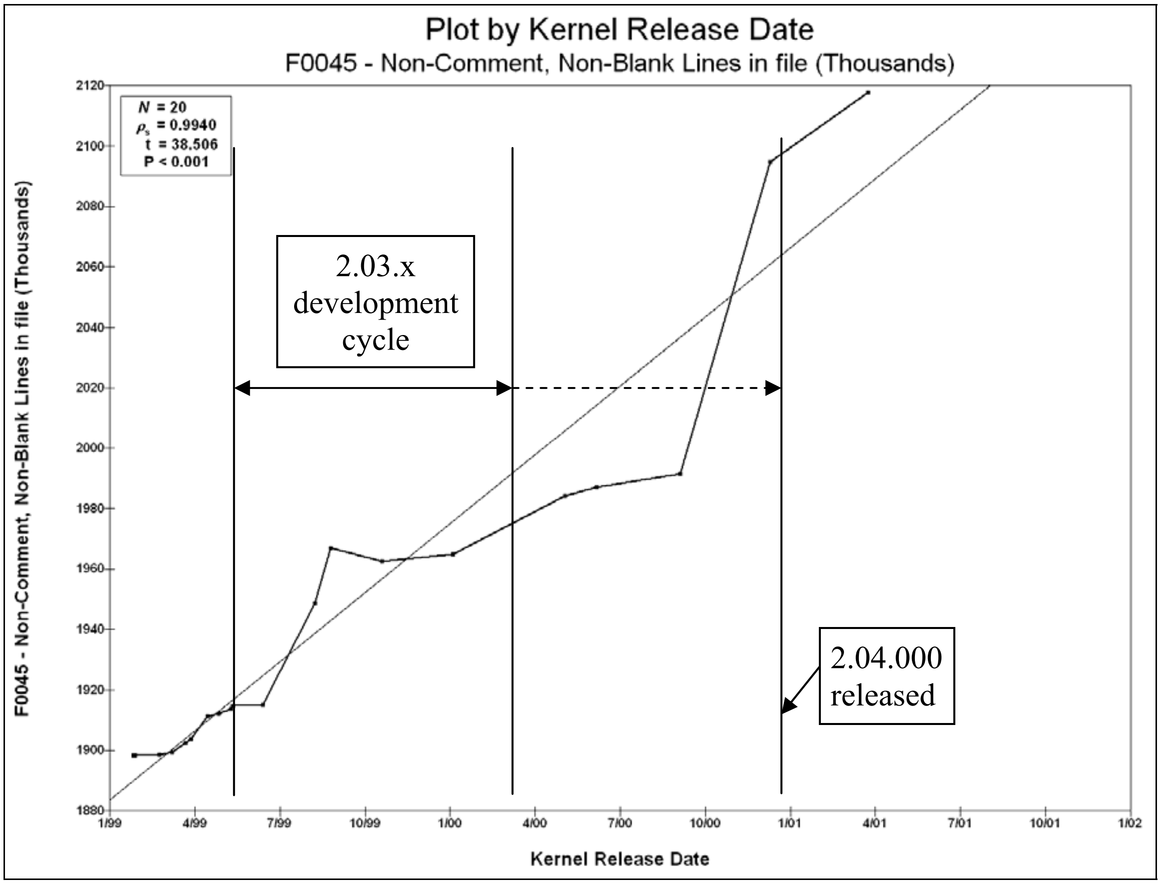


**Figure 7.19 – File Size (in thousands of NCNB lines) for 2.02.x kernels, with respect to version number**

As discussed above, for all practical purposes, the 2.00.x series ended with the release of 2.02.000. Users were migrating to the new kernel, development of the aging 2.00.x series had slowed dramatically, and releases were much less frequent. The same phenomenon occurred with the 2.02.x kernels.

Just as with the 2.00.x kernels, there is a much better linear fit when the data is considered with respect to release date than version number, because the time between releases was much longer beginning with the 2.03.x development cycle. Figure 7.19

shows a very highly significant ($P < 0.001$) correlation between release date and size of the code base, as measured in NCNB lines.

### 7.9.3 Versions 2.04.x

We now consider the 2.04.x kernels, first with respect to kernel version number. The 2.04.x kernels had not yet been released when Schach et al. performed their analysis, but we are including the 2.04.x results, analyzed in the same manner as the 2.00.x and 2.02.x series. Figure 7.20 shows the sum of the file sizes with respect to version number.

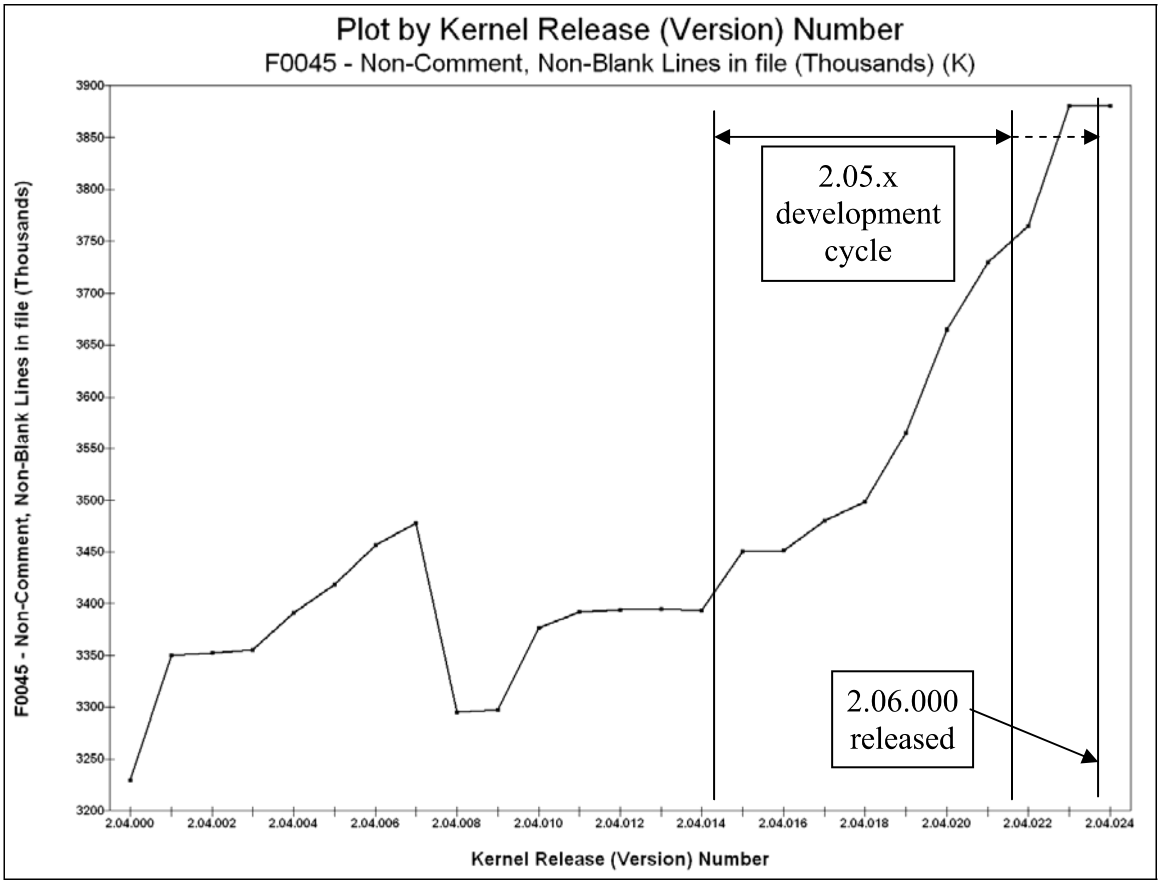


**Figure 7.20 – File Size (in thousands of NCNB lines) for 2.04.x kernels, with respect to version number**

The 2.05.x development cycle began on Nov. 21, 2001, with the release of 2.05.000, which fell between the releases of versions 2.04.014 and 2.04.015 (Nov. 5 and Nov. 30, 2001, respectively)

The drop in the size of the kernel between version 2.04.007 and 2.04.008 was because support for version 4.0 of DRM in XF86 (the X-Windows system) was dropped in 2.04.008 (version 4.1 was current at that time). DRM is the "Direct Rendering Manager," an API that provides an interface to the video hardware to accelerate graphics operations, particularly as used in gaming and other animation-dependent applications. According to the `changelog` for 2.04.008, there were several "cleanup" items, but the only large subsystem listed as having its support dropped was DRM in XF86.

An examination of the files in the two versions reveals that twenty files were deleted from the `/drivers/char/drm` subdirectory between 2.04.007 and 2.04.008. These twenty files accounted for the loss of 197,829 NCNB LOC. There were also six files in the same subdirectory modified with a total net gain of 9,049 NCNB LOC. The remaining 373 files had a total net gain of 6,423 NCNB LOC. As the Linux kernel continues to mature, and support for older subsystems (or older *versions* of subsystems) is occasionally deprecated and the code is cleaned up, it seems natural that the volatility in the size of the code base will continue to increase, or that there will continue to be substantial, sporadic jumps. We examine the effects of such changes in Section 7.10.3.

We now consider the rate of growth of the 2.04.x series, with respect to release date. This is plotted in Figure 7.21.

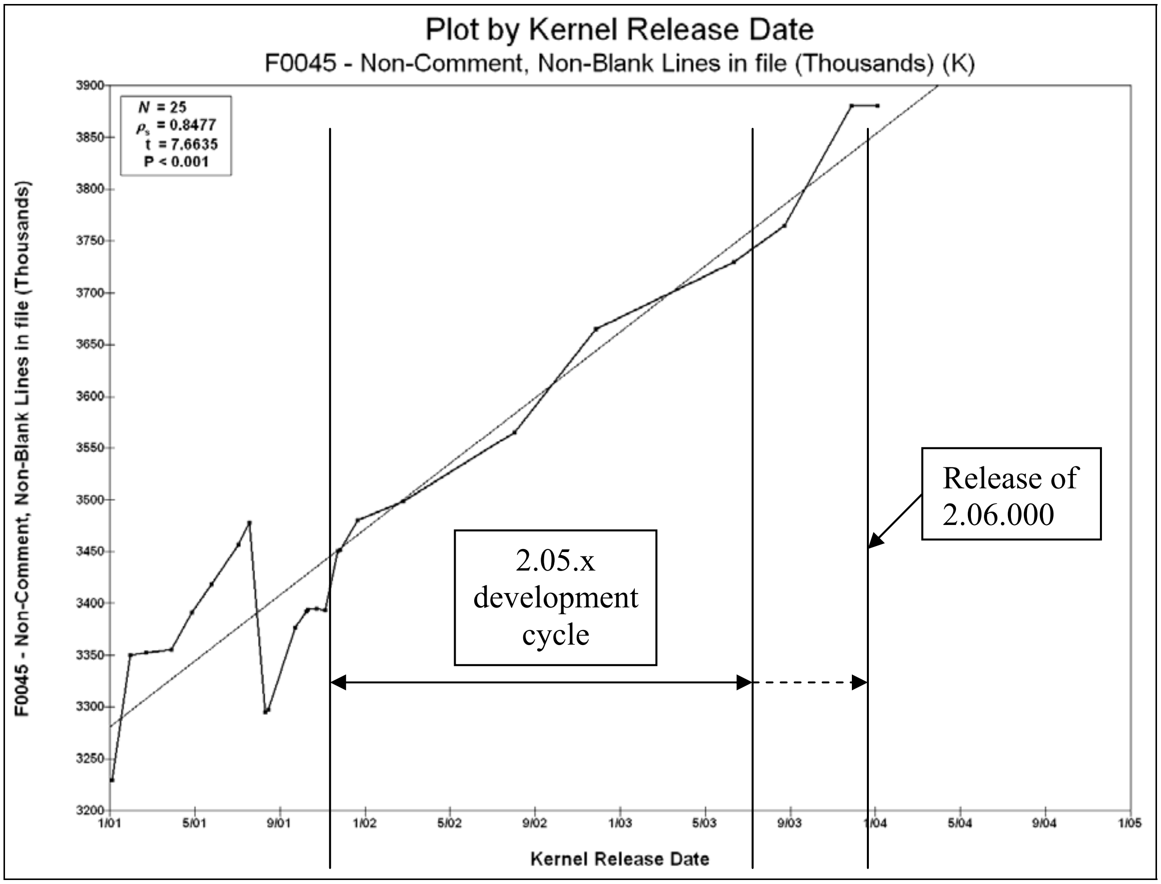


**Figure 7.21 – File Size (in thousands of NCNB lines) for 2.04.x kernels, with respect to release date**

With the exception of the drop between versions 2.04.007 and 2.04.008, Figure 7.21 shows a strongly linear relationship with respect to release date, and substantially increased release intervals coincident with the 2.05 development cycle.

### 7.9.4 All Versions

We next plot the 2.00.x, 2.02.x, and 2.04.x stable versions by release date all on one graph. This is shown in Figure 7.22.

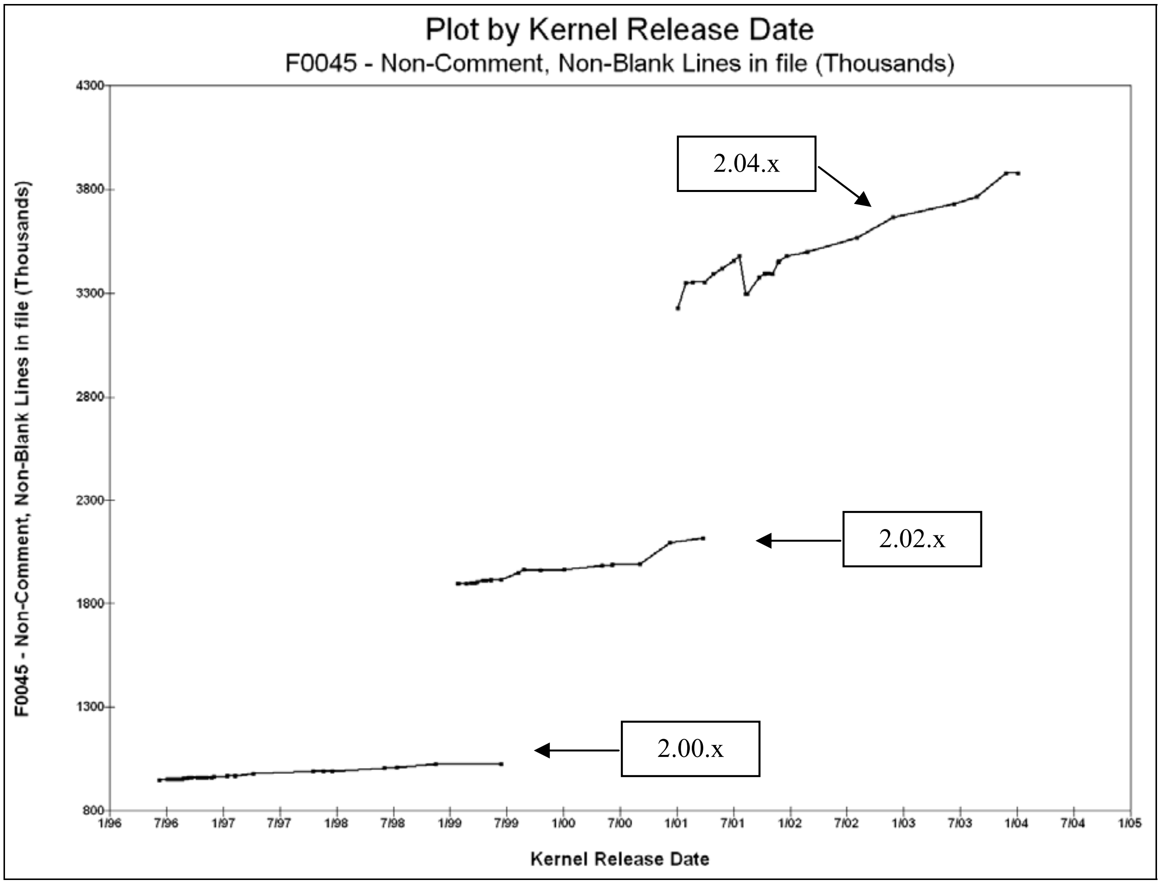


**Figure 7.22 – File Size (in thousands of NCNB lines) for 2.00.x, 2.02.x, and 2.04.x kernels, by release date**

The three stable series of kernels shown in Figure 7.22 each exhibit linear growth *within* a series. As we discussed in Section 7.7, we cannot consider the three series simultaneously. The simultaneous display of all three series in Figure 7.22 is merely a convenience to visualize the three groups relative to each other.

Figure 7.22 illustrates the problem to which we referred in Section 7.6. Because the three series were fundamentally different products, although the end of the 2.00.x series falls chronologically at the beginning of the 2.02.x series, the two cannot be directly compared. For the same reason, we cannot compare the 2.02.x series with the 2.04.x series.

The reason for this nonlinearity (and the huge discontinuity between the end of one series and the start of the next) is that, as previously pointed out in Section 7.1, the development series were more than just test environments for the newly-developed features that were ultimately incorporated into the versions in the stable series. Rather, the development series were periods during which truly new kernels evolved, and, where possible, *components* of the development series kernels were included in the stable series. However, the development series were never incorporated as a whole into the stable series kernels. The 2.02.x kernel was the product of evolutionary growth from the 2.01.132 development kernel; not revolutionary growth from the 2.00.037 kernel.

In summary, we have confirmed Schach et al.'s conclusion that the size of the stable series of the Linux kernel grows linearly, at least with respect to release date, *within* a given series, but our findings do not support the assertion that overall, the stable kernels are growing linearly. We now turn to the question of common coupling.

### 7.10.1 Common Coupling In the 2.00.x Kernels

We now consider the incidence of common coupling, starting with the 2.00.x kernels. Figure 7.23 shows the common coupling counts for the 2.00.x kernels.

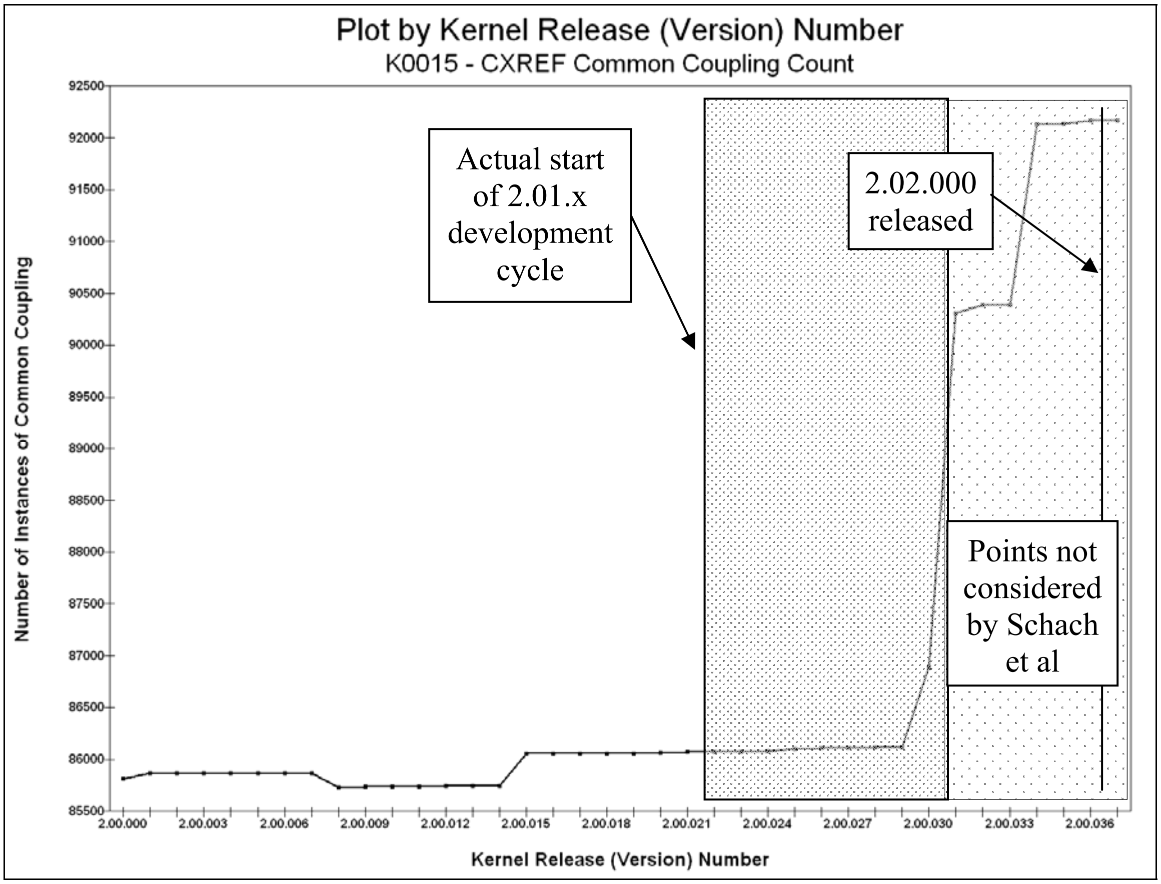


**Figure 7.23 – Common coupling for 2.00.x kernels, with respect to version number**

Figure 7.23 shows clearly that the growth of common coupling in the 2.00.x stable versions passes through three distinct nearly-linear phases. We observe:

1. A relatively low rate of growth prior to the next development cycle;
2. An extremely high growth rate during the development cycle; and
3. An extremely low rate of growth following the release of the first version of the next stable series.

Figure 7.24 shows the same data plotted with respect to release date.

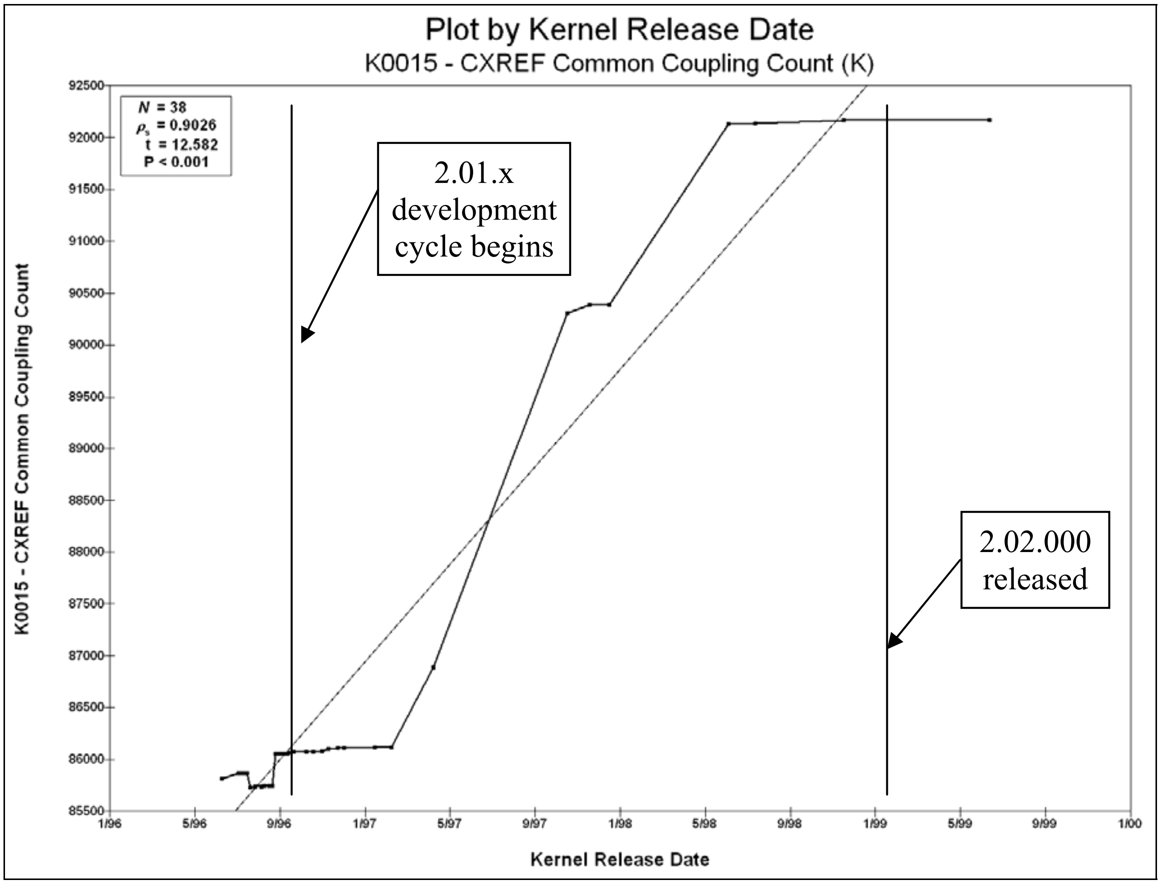


**Figure 7.24 – Common coupling for 2.00.x kernels, with respect to release date**

Figures 7.15 and 7.17 showed two completely differently shaped graphs when the file size was considered by version number and then by release date. Analogously, Figures

7.23 and 7.24 show that it is hard to determine the true rate of growth of common coupling using version numbers as a reference for the rate of change; a much more meaningful basis is the release date.

When the rate of change of common coupling in the 2.00.x series is considered with respect to release date, the result is a very highly significant (P<0.001) correlation.

### 7.10.2 Common Coupling In the 2.02.x Kernels

Figure 7.25 shows the common coupling values for the 2.02.x kernels, plotted with respect to version number.

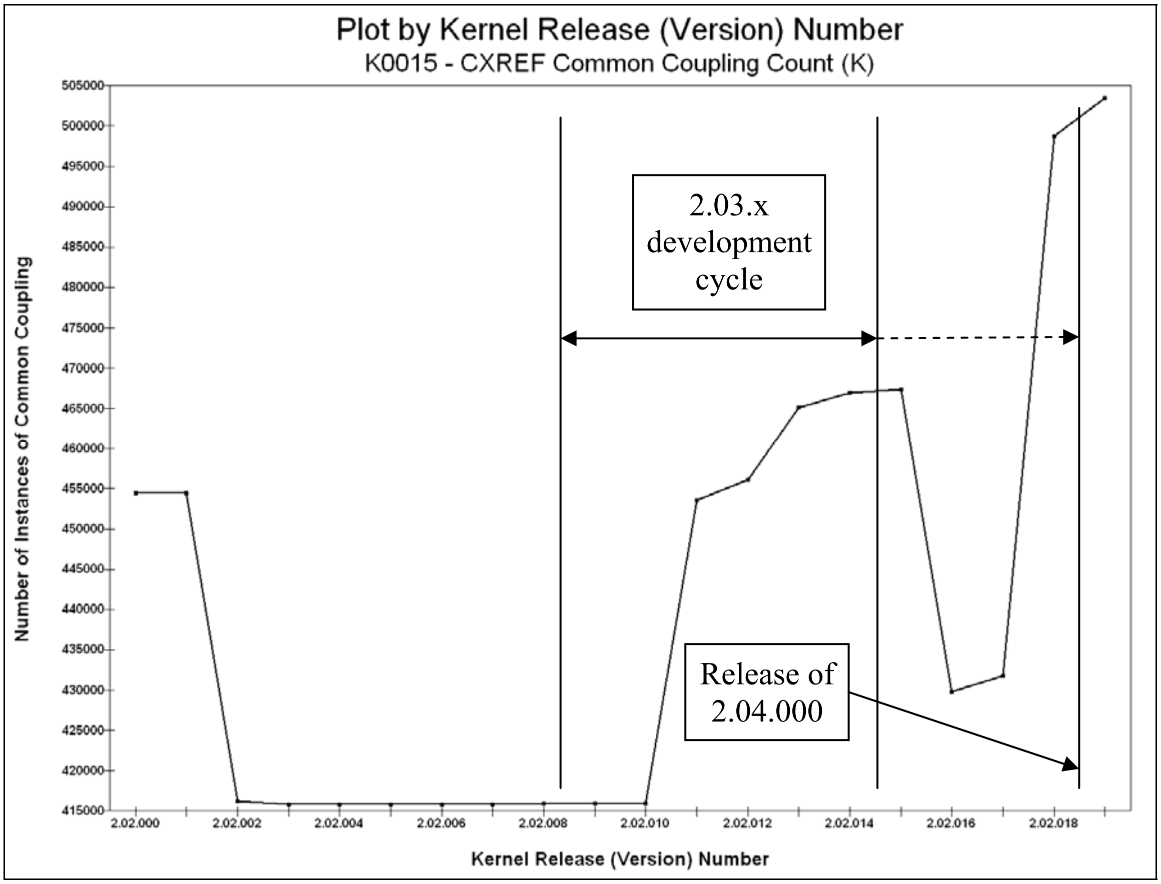


**Figure 7.25 – Common coupling for 2.02.x kernels, with respect to version number**

Figure 7.25 shows a trend unlike what we have observed before, in that the value measured was not relatively consistent in all versions prior to the initiation of the next development cycle. In this case, there was a sharp drop in the common coupling beginning with version 2.02.002, although the common coupling did remain constant after that until well into the development cycle, at which time it began the expected process of volatile change with a general rising trend, followed by a consistent value in the final versions.

It is not clear why the common coupling dropped in version 2.02.016, but the *rise* in common coupling in version 2.02.018 was most likely due to the growth of the code base arising from the debut of support for the following items (all of which are incorporated in our configuration):

- NFS (Network File System) server support
- USB (Universal Serial Bus) devices, including
    - UHCI (the USB Universal Host Controller Interface) and
    - Hot-Pluggable support for USB devices

Figure 7.26 plots this same data with respect to release date, rather than version number.

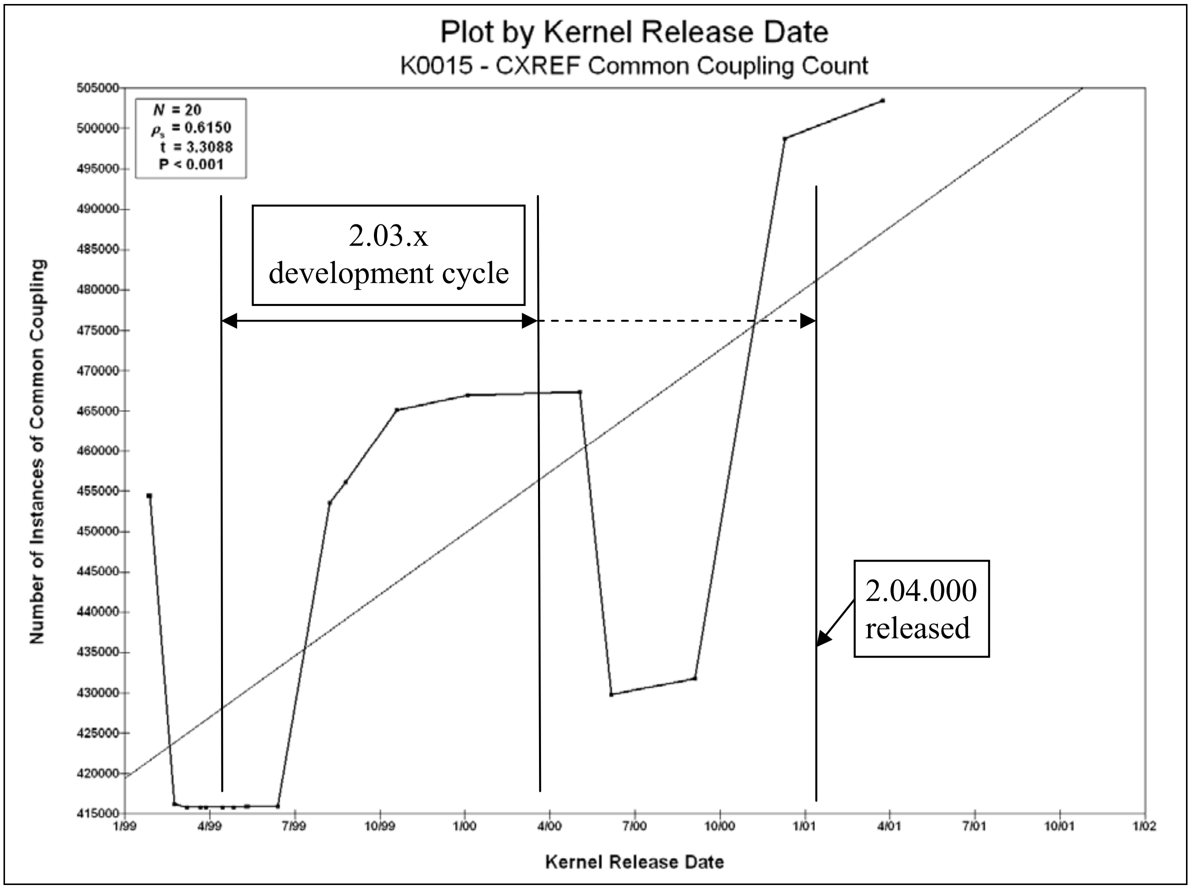


**Figure 7.26 – Common coupling for 2.02.x kernels, with respect to release date**

The fact that the first ten releases of the 2.02.x kernels occurred in one-sixth of the time of the next ten releases paints a somewhat different picture in Figure 7.26 than the one suggested by Figure 7.25. We are unsure of the source of the anomalies during the development cycle and with the first two data points, but, overall, the data does exhibit an overall pattern of linear growth, with the actual data oscillating back and forth across the best-fit line, and a very highly significant correlation (P<0.001).

### 7.10.3 Common coupling in the 2.04.x Kernels

Figure 7.27 shows the common coupling values for the 2.04.x kernels, plotted with respect to version number.

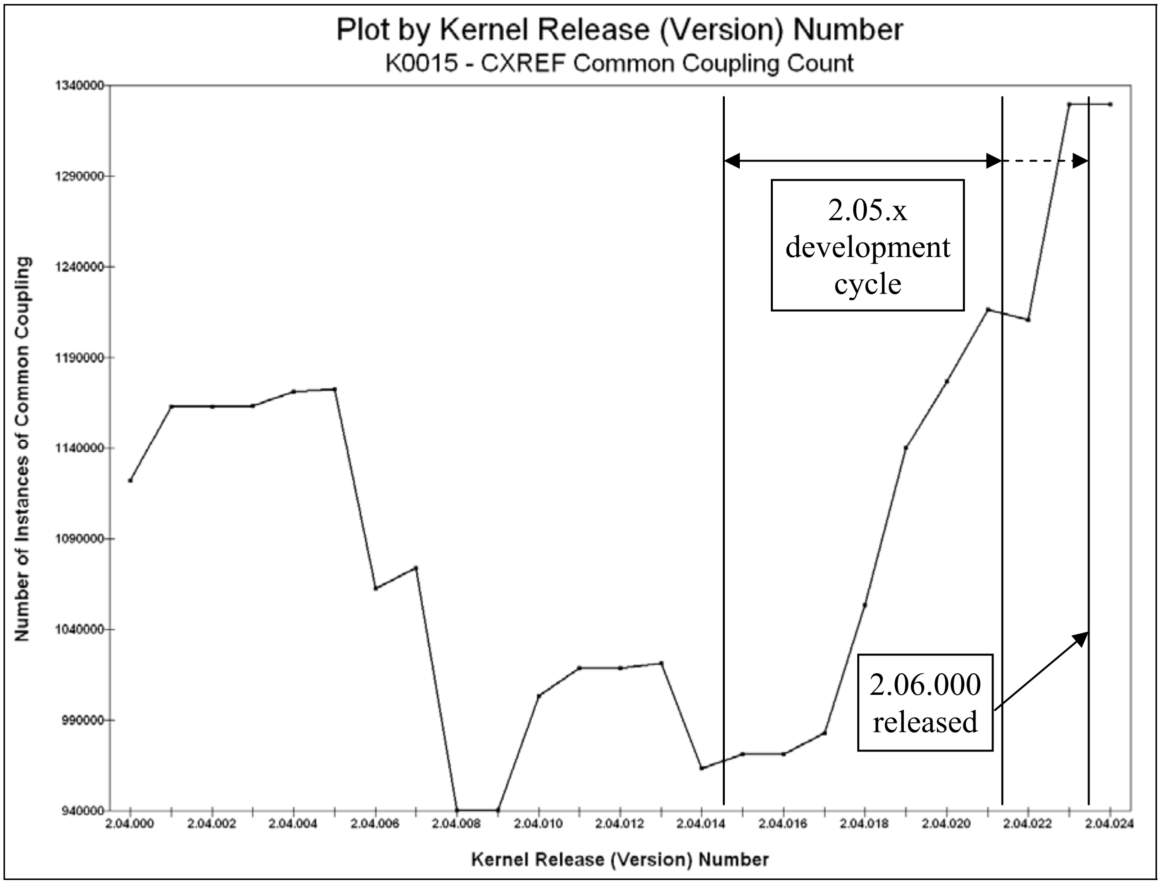
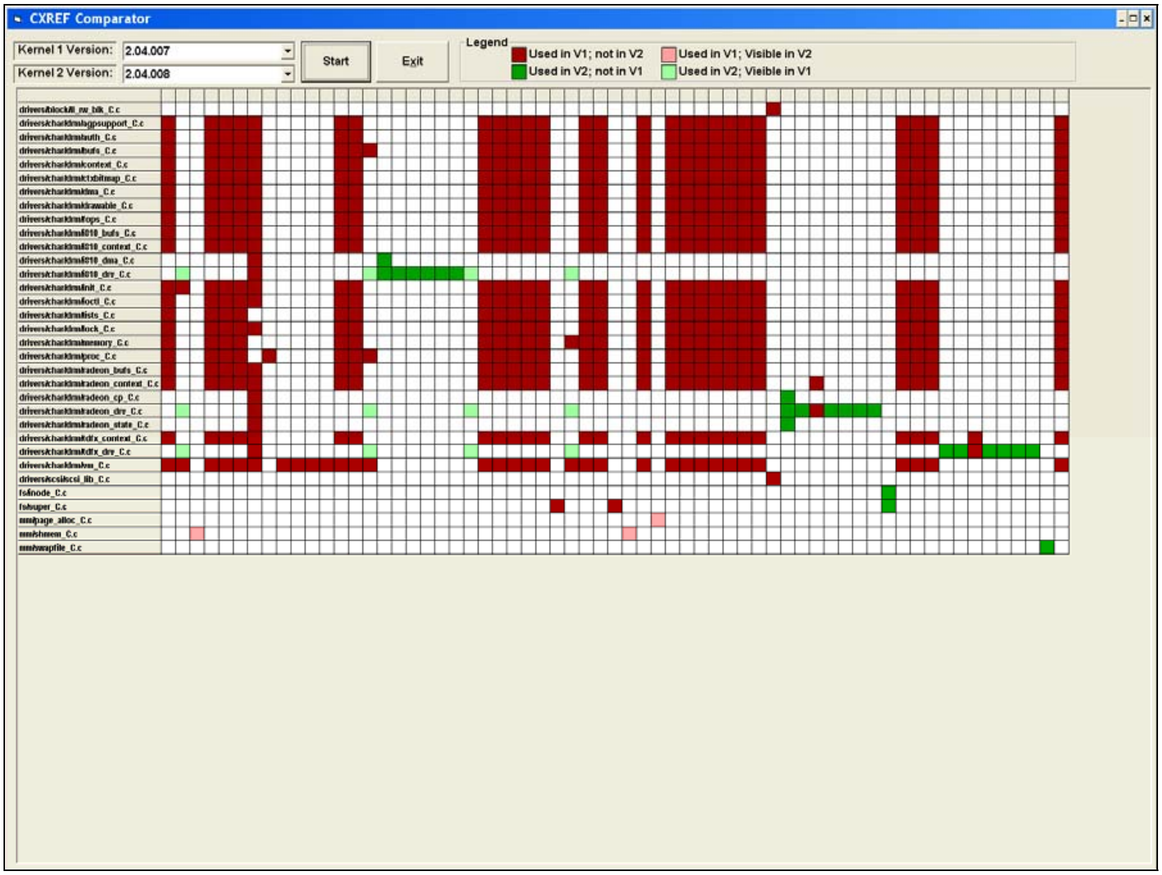


**Figure 7.27 – Common coupling for 2.04.x kernels, with respect to version number**

Figure 7.27 shows that, prior to the initiation of the 2.05.x development cycle, common coupling rose slightly and then decreased (decreasing a total of about 20 percent between versions 2.04.005 and 2.04.008). Figure 7.20 shows that the size of the code base *grew* between versions 2.04.005 and 2.04.006, yet the common coupling *decreased* between the same two versions. The code size and the common coupling *both* decreased between versions 2.04.007 and 2.04.008, so the drop in common coupling does not, in this case, seem to be related solely to having a smaller code base. This effect was interesting enough to explore further.

We developed another tool, `CXREF_Comparator`, to show the changes in common coupling between two (presumably consecutive) versions of the kernel, and applied it first to the versions 2.04.007 and 2.04.008. As we determined in examining the drop in the code size for these two versions (see Figure 7.26), there were twenty files related to the support of DRM 4.0 that were deleted in 2.04.008, because DRM 4.1 was then current. Although it seems reasonable to assume that the loss of twenty source files (out of about 400) might naturally include the loss of some global variables, this was by no means guaranteed. Figure 7.27 shows that the drop in common coupling between these two versions was approximately 130,000 (from 1,070,000 to 940,000). The output from `CXREF_Comparator` for kernel versions 2.04.007 and 2.04.008 is shown in Figure 7.28.



Kernel 1 Version: 2.04.007

Kernel 2 Version: 2.04.008

Legend

🟥 Used in V1; not in V2

🟩 Used in V2; not in V1

🟥 Used in V1; Visible in V2

🟩 Used in V2; Visible in V1

drivers\block\ll_rw_blk_C.c
drivers\char\drm\agpsupport_C.c
drivers\char\drm\auth_C.c
drivers\char\drm\bufs_C.c
drivers\char\drm\context_C.c
drivers\char\drm\ctxbitmap_C.c
drivers\char\drm\dma_C.c
drivers\char\drm\drawable_C.c
drivers\char\drm\fops_C.c
drivers\char\drm\i810_bufs_C.c
drivers\char\drm\i810_context_C.c
drivers\char\drm\i810_dma_C.c
drivers\char\drm\i810_drv_C.c
drivers\char\drm\init_C.c
drivers\char\drm\ioctl_C.c
drivers\char\drm\lists_C.c
drivers\char\drm\lock_C.c
drivers\char\drm\memory_C.c
drivers\char\drm\proc_C.c
drivers\char\drm\radeon_bufs_C.c
drivers\char\drm\radeon_context_C.c
drivers\char\drm\radeon_cp_C.c
drivers\char\drm\radeon_drv_C.c
drivers\char\drm\radeon_state_C.c
drivers\char\drm\tdfx_context_C.c
drivers\char\drm\tdfx_drv_C.c
drivers\char\drm\vm_C.c
drivers\scsi\scsi_lib_C.c
fs\inode_C.c
fs\super_C.c
mm\page_alloc_C.c
mm\shmem_C.c
mm\swapfile_C.c

**Figure 7.28 – `CXREF_Comparator` output for kernel versions 2.04.007 and 2.04.008**

The rows in the `CXREF_Comparator` output represent files, and the columns represent global variables. The columns are not labeled, but when the program is running, clicking on a column header causes the program to pop up the name of the GV corresponding to that column.

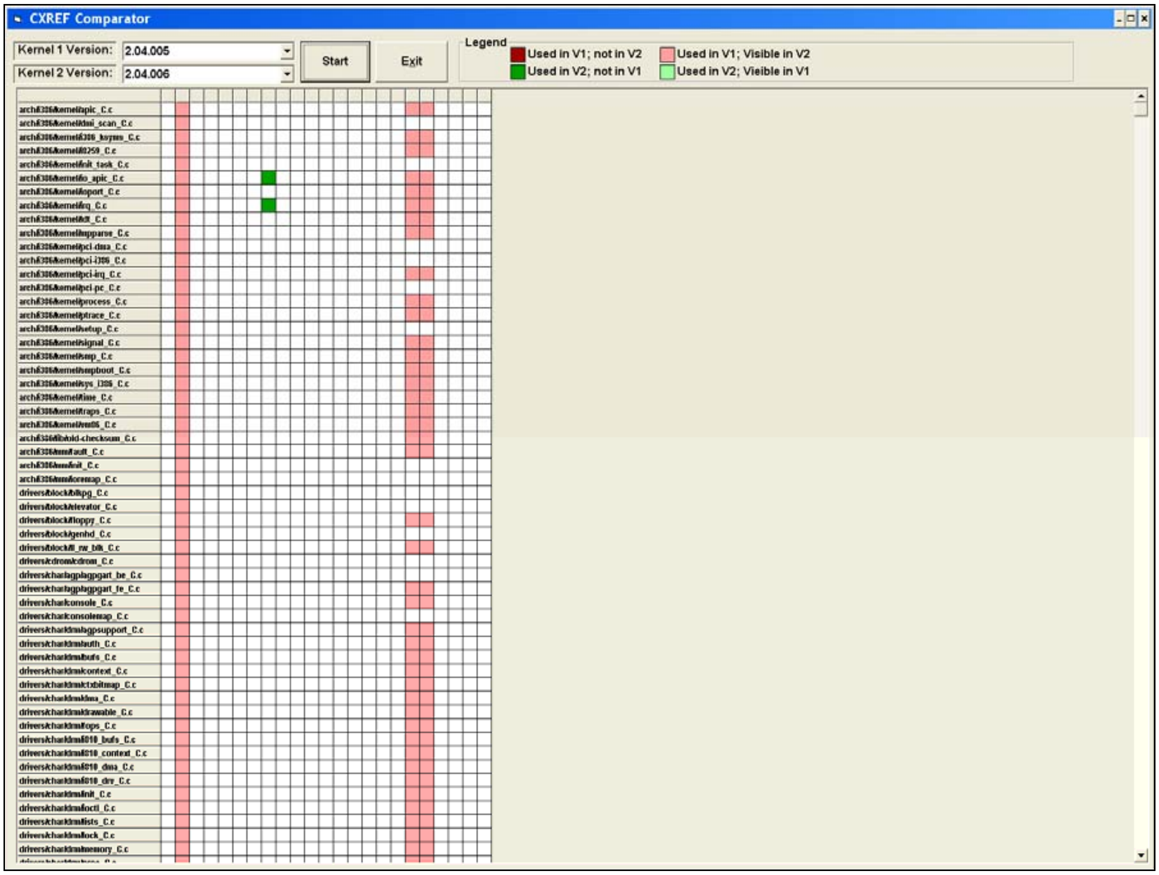
`CXREF_Comparator` shows dark red when global variables change from being used to not appearing at all between versions (as would be the case when files are deleted). Using our other tool, `CXREF_Analyzer`, we determined that the twenty-six GVs appearing in the twenty deleted files also appeared in many *other* files. For example, the

GVs `dcache_lock` and `tqueue_lock` both appeared in 381 files in 2.04.007, but in only 361 files in 2.04.008. Because the number of instances of common coupling when $N$ files share a *single* GV is $\frac{(N^2 - N)}{2}$, when $N$ changes from 381 to 361, $\frac{(N^2 - N)}{2}$ changes from 72,390 to 64,980, decreasing the total common coupling by 7,410. Repeating this calculation over all GVs represented in the twenty deleted files accounts for 133,640 instances of common coupling.

We have actually accounted for slightly *more* than the total decrease in common coupling between versions 2.04.007 and 2.04.008, but the excess is offset by the new instances of common coupling created by the introduction of the variables shown in green in Figure 7.28.

Having determined conclusively that the decrease in common coupling between kernel versions 2.04.007 and 2.04.008 was due to the removal of the code supporting the deprecated DRM 4.0 functionality, we next turned our attention to the drop in common coupling between kernel versions 2.04.005 and 2.04.006, when the size of the code base did *not* decrease. Moreover, the 399 files comprising the 2.04.006 kernel were the same 399 files as the 2.04.005 kernel – no new files were introduced in 2.04.006, and no files from 2.04.005 were deleted.

Figure 7.29 shows the `CXREF_Comparator` output for versions 2.04.005 and 2.04.006.



Kernel 1 Version: 2.04.005
Kernel 2 Version: 2.04.006

arch/i386/kernel/apic_C.c
arch/i386/kernel/dmi_scan_C.c
arch/i386/kernel/i386_ksyms_C.c
arch/i386/kernel/i8259_C.c
arch/i386/kernel/init_task_C.c
arch/i386/kernel/io_apic_C.c
arch/i386/kernel/ioport_C.c
arch/i386/kernel/irq_C.c
arch/i386/kernel/ldt_C.c
arch/i386/kernel/mpparse_C.c
arch/i386/kernel/pci-dma_C.c
arch/i386/kernel/pci-i386_C.c
arch/i386/kernel/pci-irq_C.c
arch/i386/kernel/pci-pc_C.c
arch/i386/kernel/process_C.c
arch/i386/kernel/ptrace_C.c
arch/i386/kernel/setup_C.c
arch/i386/kernel/signal_C.c
arch/i386/kernel/smp_C.c
arch/i386/kernel/smpboot_C.c
arch/i386/kernel/sys_i386_C.c
arch/i386/kernel/time_C.c
arch/i386/kernel/traps_C.c
arch/i386/kernel/vm86_C.c
arch/i386/lib/old-checksum_C.c
arch/i386/mm/fault_C.c
arch/i386/mm/init_C.c
arch/i386/mm/ioremap_C.c
drivers/block/blkpg_C.c
drivers/block/elevator_C.c
drivers/block/floppy_C.c
drivers/block/genhd_C.c
drivers/block/ll_rw_blk_C.c
drivers/cdrom/cdrom_C.c
drivers/char/agp/agpgart_be_C.c
drivers/char/agp/agpgart_fe_C.c
drivers/char/console_C.c
drivers/char/consolemap_C.c
drivers/char/drm/agpsupport_C.c
drivers/char/drm/auth_C.c
drivers/char/drm/bufs_C.c
drivers/char/drm/context_C.c
drivers/char/drm/ctxbitmap_C.c
drivers/char/drm/dma_C.c
drivers/char/drm/drawable_C.c
drivers/char/drm/fops_C.c
drivers/char/drm/i810_bufs_C.c
drivers/char/drm/i810_context_C.c
drivers/char/drm/i810_dma_C.c
drivers/char/drm/i810_drv_C.c
drivers/char/drm/init_C.c
drivers/char/drm/ioctl_C.c
drivers/char/drm/lists_C.c
drivers/char/drm/lock_C.c
drivers/char/drm/memory_C.c

**Figure 7.29 – `CXREF_Comparator` output for kernel versions 2.04.005 and 2.04.006**

Figure 7.29 shows that the vast majority of the decrease in common coupling between versions 2.04.005 and 2.04.006 was due to three GVs that went from being *used* to merely *visible*. Figure 7.27 shows that the decrease in common coupling totaled approximately 110,000. The three variables shown in Figure 7.29 whose status changed from used to visible were `contig_page_data`, `tasklet_hi_vec`, and `tasklet_vec`.

According to `CXREF_Analyzer`:

`contig_page_data`: is visible in 0 files, used in 356 files (2.04.005),
is visible in 354 files, used in 3 files (2.04.006).

`tasklet_hi_vec`: is visible in 0 files, used in 222 files (2.04.005),
is visible in 221 files, used in 1 file (2.04.006).

`tasklet_vec`: is visible in 0 files, used in 222 files (2.04.005),
is visible in 221 files, used in 1 file (2.04.006).

Note: "used in" presumes "visible in," so even though it says "visible in 0 and used in 356," it was obviously "both visible and used in 356, and visible-ONLY in none."

By dropping these counts from 356 to 3, and 222 to 1:

| | | | |
|---|---|---|---|
| (356*356 – 356) / 2 = 63,190 | (3 * 3 – 3) / 2 = 3 | | Net: –63,187 |
| (222*222 – 222) / 2 = 24,531 | (1 * 1 – 1) / 2 = 0 | | Net: –24,531 |
| (222*222 – 222) / 2 = 24,531 | (1 * 1 – 1) / 2 = 0 | | Net: –24,531 |
| | | Total: | –112,249 |

Because these three GVs went from being used in 222 – 356 files to being used in only 1 – 3, we conclude that the GVs were not actually required in all of the files in which they had previously appeared. According to the change log for 2.04.006, there were several areas in which cleanup of the code was performed, and it is most likely that the function(s) that referred to these three GVs were changed with respect to the `#include` tree, and that the code that referred to these GVs was not actually called in all files.

We have again accounted for slightly more (112,249) than the actual drop in common coupling (109,815). As with the previous case, this is offset by the modest gains in common coupling caused by the introduction of approximately 10 new GVs.

The third substantial drop in common coupling in the 2.04.x series, as indicated by Figure 7.27, occurred between versions 2.04.013 and 2.04.014. Using CXREF_Comparator and CXREF_Analyzer revealed that the global variable swapper_space went from being used in 338 files to being used in 9 files, for a net decrease in common coupling of 56,917. The difference between this decrease and the actual difference of 58,014 is easily accounted for by the minor contributions of 17 other GVs. In the interest of brevity, the CXREF_Comparator output for this case is omitted.

Returning now to our analysis of common coupling throughout the 2.04.x series, Figure 7.30 plots the 2.04.x kernels' common coupling with respect to release date.

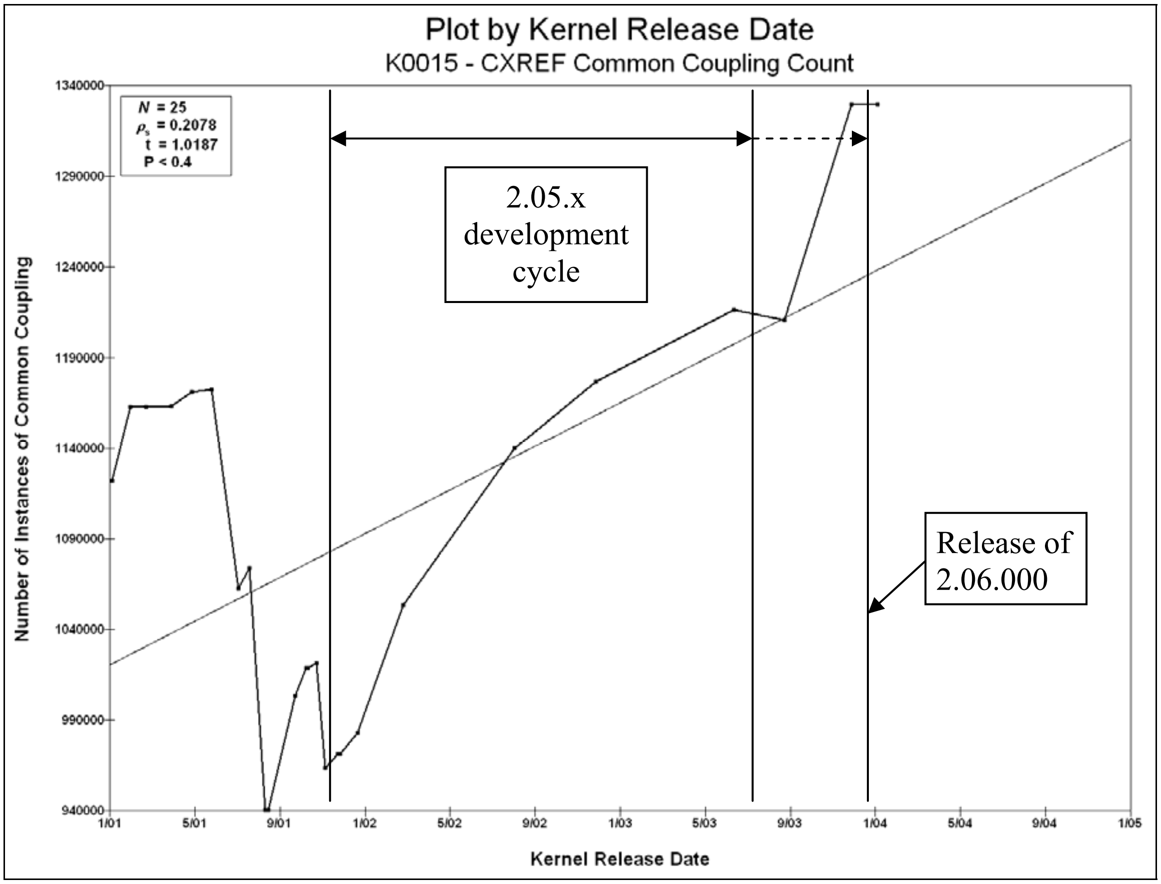


**Figure 7.30 – Common coupling for 2.04.x kernels, with respect to release date**

Figure 7.30 shows that there is still what appears to be generally linear growth in the common coupling in the 2.04.x kernels. However, the three drops in common coupling we analyzed above change the trajectory of common coupling in the 2.04.x series profoundly. The result is that common coupling is no longer significantly correlated with release date. Two points bear emphasis:

1) The goal of performing a correlation is to account for the variation observed in one variable associated with another variable. In this case, we have a poor correlation because of three incidents (changes to the code) that had profound impact on the common coupling. Having accounted for the portion of the variability in common coupling due to these three changes, we believe that the remainder of the variation *is* accounted for by a generally linear relationship between release date and common coupling.

2) Just as growth is inevitable in software as users demand new features (and new code must be written to implement that functionality), it is also inevitable that software must either undergo cleanup or become so littered with obsolete code that further maintenance becomes exceedingly difficult, error-prone, or both. In this case, we have observed that the cleanup performed on the 2.04.x series of kernels caused such volatility in the common coupling that the common coupling was no longer significantly correlated with release date. As a program as feature-rich as the Linux kernel continues to mature, it seems obvious that a point will come at which the jettisoning of old code has as significant an impact in some

ways as the creation of new code. In this case, that impact appeared in common coupling. Although the general trend over time is that both size and coupling are increasing, they do not increase monotonically.

As we discussed in Section 7.2, development series kernels reach a point at which the new developments can no longer be bolted onto the stable kernels, and that at that point the entire stable kernel must be jettisoned so that the features in the development series may incorporated into the mainstream, stable kernels. This is the point at which a new stable series debuts with a .000 release number.

It seems that what happened in the 2.04.x series was such a jettisoning; however, in this case, it did not require scrapping the series. Rather, the existing series was cleaned up and no other changes were required.

### **7.10.4 All Versions**

For the sake of completeness, we next plot the common coupling values for all of the 2.00.x, 2.02.x, and 2.04.x stable versions by release date on one graph, as we did for size in Figure 7.22. This is shown in Figure 7.31.

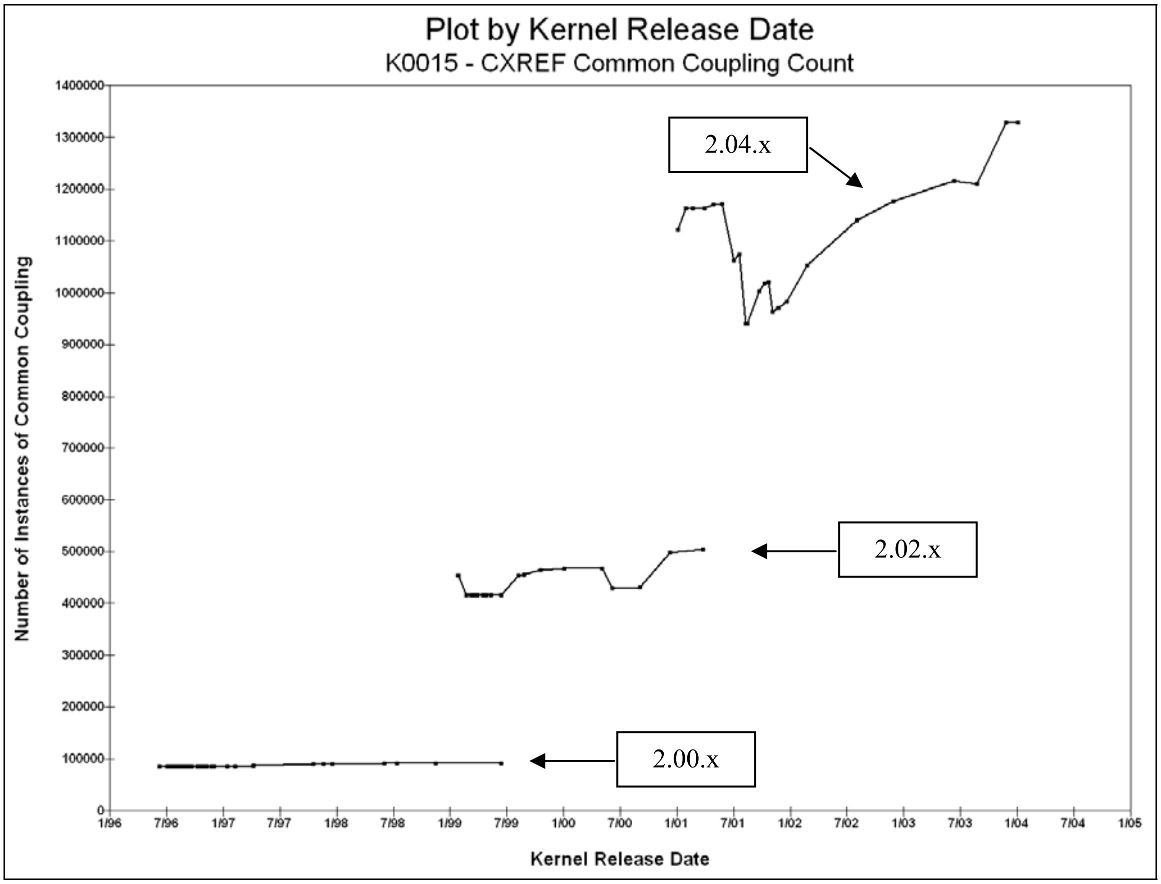


**Figure 7.31 – Common Coupling for 2.00.x, 2.02.x, and 2.04.x kernels, by release date**

## 7.11 Examining Schach et al.'s data with respect to release date

In light of our conclusions, we now revisit the work of Schach et al., and examine their data with respect to release date, rather than version number.

Schach et al. graciously provided us with their raw data, which we first checked against the steps detailed in their study. Schach et al. stated that they had "downloaded all 391 versions of Linux available on the Web," but there appear to have been several versions omitted altogether, a few versions included that their rules should have eliminated, and some eliminated that should have been included.

We were provided with data on 391 versions; however, the 1.00.x series consisted of ten versions, 1.00.000 – 1.00.009, released between 13-Mar-1994 and 16-Apr-1994. Schach et al.'s data lists only a single version labeled "1.0." It is not clear to which of these ten versions this refers; however the newest file date on the files in the `/kernel/` directory is 04-Mar-1994. Because these files all predate the release of version 1.00.000, we will assume that Schach et al's "1.0" data refers to 1.00.000. The other nine versions of the 1.00.x kernel series are not mentioned in their data.

The next data point Schach et al. provide is labeled "1.1." Because the newest file date they cite is 06-Apr-1994, and because this matches the release date of kernel version 1.01.000, we will assume that their reference to "1.1" is intended to mean version 1.01.000.

Schach et al. then list the other versions they examined, but the following 65 versions are not listed at all in their data

1.01.001
1.01.002
1.01.003
1.01.004
1.01.005
1.01.006
1.01.007
1.01.008
1.01.009
1.01.010
1.01.011
1.01.012
1.01.014
1.01.015
1.01.016
1.01.017
1.01.018
1.01.019
1.01.020
1.01.021
1.01.022
1.01.024
1.01.025
1.01.026
1.01.027
1.01.028
1.01.030
1.01.031
1.01.032
1.01.034
1.01.036
1.01.037
1.01.038
1.01.039
1.01.040
1.01.041
1.01.042
1.01.043
1.01.044
1.01.046
1.01.047
1.01.048
1.01.049
1.01.050
1.01.051
1.01.053
1.01.054
1.01.055
1.01.056
1.01.057
1.01.058
1.01.059
1.01.060
1.01.061
1.01.062
1.01.065
1.01.066
1.01.068
1.01.069
1.01.072
1.01.077
1.03.001
2.01.025
2.01.026
2.01.125

Schach et al. eliminated versions 2.00.030 – 2.00.039. According to the rules set forth in their section "Successive Versions," they should also have eliminated versions 1.02.011, 1.02.012, and 1.02.013, because they were released in 1995 on 26-Jun, 25-Jul, and 02-Aug, respectively, all subsequent to the release of 1.03.000 on 12-Jun.

Similarly, they list versions 2.00.022 – 2.00.029, released 08-Oct-96 through 07-Feb-97, despite the fact that 2.01.000 was released on 30-Sep-96. According to their rule, they should have eliminated these kernel versions.

Schach et al. excluded versions 2.02.002 – 2.02.008, released 23-Feb-99 through 11-May-99; however, version 2.03.000 was released on 11-May-99. As such, they should have included these versions.

In the interest of an apples-to-apples comparison, we used the same 365 data points Schach et al. used.

The size of the code base with respect to time is plotted in Figure 7.32.

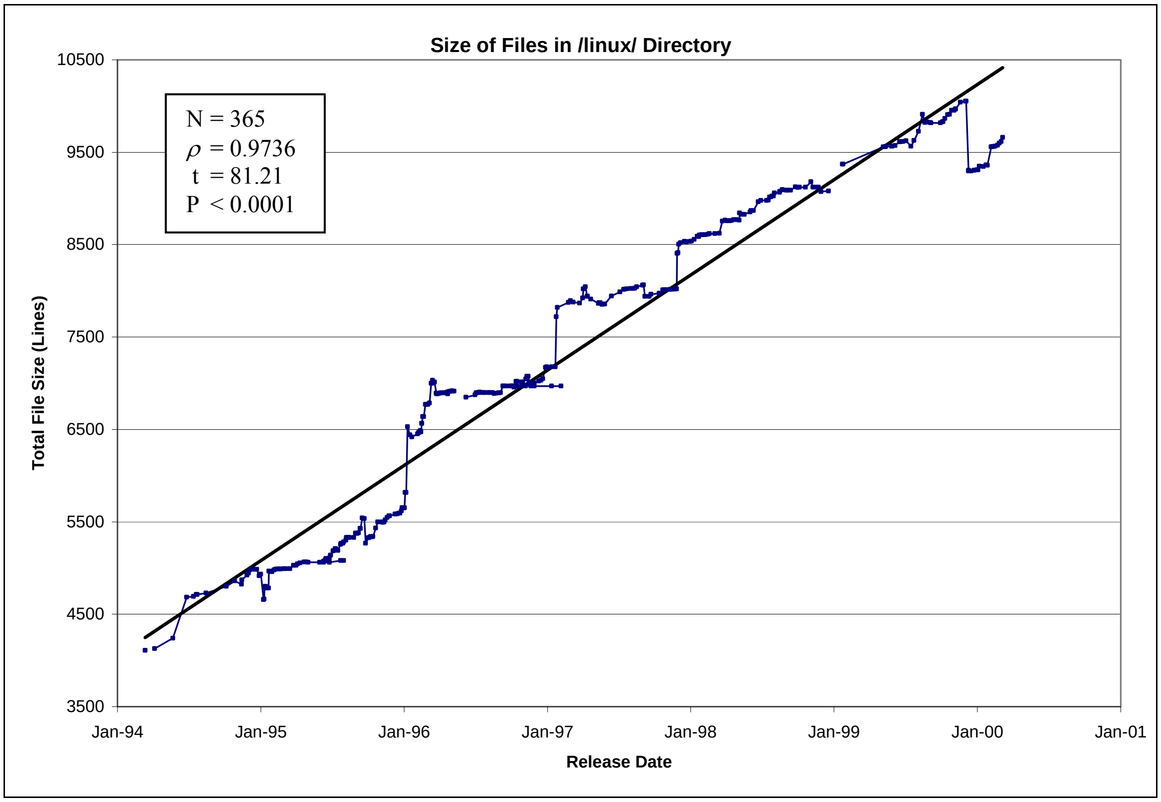


**Figure 7.32 – Size of code base of Schach et al, with respect to kernel release date.**

Because the relationship in Figure 7.32 is extremely highly significant (P < 0.0001), we have shown that Schach et al.'s data shows linear growth of the size of the code base, as measured in total lines of code in the `/linux/` directory of 365 kernel versions they examined, with respect to release date.

Figure 7.33 shows the common coupling data, as counted by Schach et al, but plotted by release date.

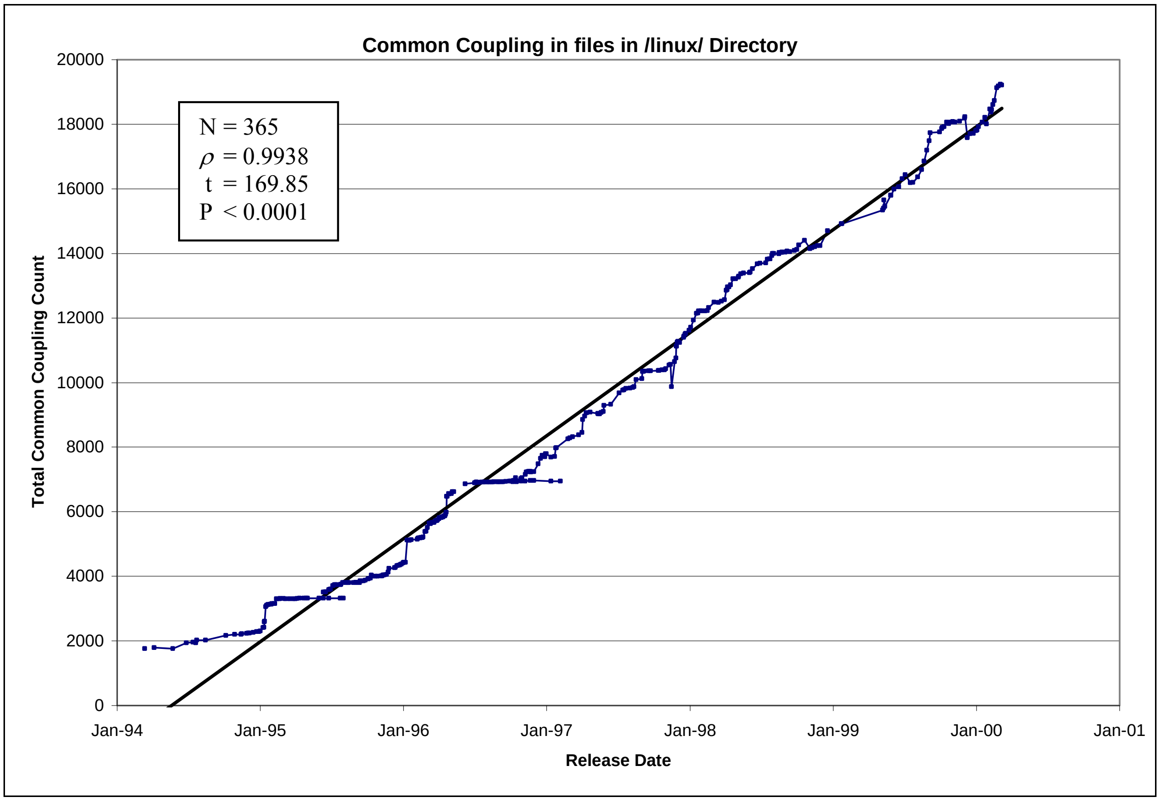


**Figure 7.33 – Common Coupling in code base of Schach et al, with respect to kernel release date.**

Because the relationship in Figure 7.33 is extremely highly significant (P < 0.0001), we have shown that Schach et al.'s data shows linear growth in the incidence of common coupling as measured in the source files contained in the `/linux/` directory of the 365 kernel versions they examined, with respect to release date.

## 7.12 Summary and Conclusions

Our reexamination of the work of Schach et al. was intended to provide a more in-depth examination of both the rate of growth of the size of the Linux kernel, and also the rate of growth of common coupling between modules in the kernel. We looked at common coupling between kernel modules and *all* modules, not just inside the kernel.

More specifically, we wished to examine the choice of independent variable when considering growth in open-source software. Schach et al., in keeping with previous research (see Section 2.6), used sequential version numbers as their independent variable, and determined that common coupling was growing exponentially. When viewed with respect to time (or, more precisely, release date), however, we have shown this to be a linear relationship both with Schach et al.'s data, and using our data. The lone exception to this occurred in the 2.04.x series, where three instances of cleanup so profoundly reduced the linear growth in coupling that the correlation was no longer statistically significant.

For closed-source software, an analysis using version number seems to be a reasonable approach, because the business environment under which closed-source development typically takes place is based on (somewhat) routine business cycles and planned release schedules. In the case of mass-market products, customers come to expect updates at somewhat regular intervals.

The open-source development paradigm, on the other hand, typically has no connection to a business plan, and there may be no release schedule at all, let alone a regularly-planned cycle of releases. With releases potentially coming at quite irregular intervals, examining the rate of growth with respect to version number fails to capture the *rate* of growth, because the release dates have no specific temporal pattern.

Table 7.2 summarizes the aspects of these two projects that were handled differently.

**Table 7.2. Differences Between Schach et al. and Thomas Studies.**

| Parameter | Schach et al. | Thomas |
|---|---|---|
| Kernels examined | 1.00.x – 1 version | |
| | 1.01.x – 36 versions | |
| | 1.02.x – 14 versions | |
| | 1.03.x – 100 versions | |
| | 2.00.x – 30 versions | 2.00.x – 38 versions |
| | 2.01.x – 130 versions | |
| | 2.02.x – 2 versions | 2.02.x – 20 versions |
| | 2.03.x – 52 versions | |
| | | 2.04.x – 25 versions |
| | Total: 365 versions | Total: 83 versions |
| | 318 Development versions<br>47 Stable versions | All 83 are Stable versions |
| | Mixed 1.x and 2.x kernels | All 2.x kernels |
| Source code base | `.c` files in the `/kernel/` subdirectory<br>1.00.000: 12 of 19 files ( 99.7 KB)<br>2.03.051: 17 of 24 files (221.7 KB) | Preprocessed code for entire kernel, using a consistent configuration<br>2.00.000: 225 files ( 46.8 MB)<br>2.04.024: 406 files (233.8 MB) |
| Basis for trends | Sequential version number | Release date (time) |
| Basis for size | Total Lines in `.c` files.<br>Header (`.h`) files not included | NBNC (Non-blank, non-comment) lines in preprocessed code |

Schach et al. concluded that the size of the Linux kernel grows linearly with respect to version number. When we examined the Linux kernels with respect to time (rather than version number), we also found a linear rate of growth.

Turning now to the incidence of common coupling, Schach et al. found an exponential relationship between incidence of common coupling and version number. When we examined the relationship of common coupling versus release date (rather than version number), we again found a linear rate of growth, rather than an exponential rate.

We do not dispute the correctness of the results in [Schach et al., 2002], nor do we disagree with the conclusions of their paper, based on those results. However, we feel that it was incorrect for Schach et al. to choose version number as the independent variable, because the versions were not released at equal time intervals. Instead, they should have chosen time (release date) as the independent variable. Choice of independent variable is our major point of contention. In addition, however, we are uncomfortable with other aspects of their research, including their decision to examine both development and stable versions, and that their basis for measuring code size was limited to the files in the `/kernel/` subdirectory, yet they examined common coupling between the files in the `/kernel/` subdirectory and the rest of the entire code base.

Other points of contention are less clear-cut. For example, we chose to measure size using NCNB lines in the preprocessed code with a consistent configuration, whereas Schach et al. looked at the total number of lines in the `.c` files. Both approaches are relevant with respect to measuring maintainability. In our opinion, use of preprocessed code is a superior approach, but use of `.c` files is certainly not incorrect.

Finally, with the passage of time, more versions of the kernel have been released, so there is simply more data to examine. Since the work of Schach et al., the stable Linux kernels 2.02.018, 2.02.19, and 2.04.000 through 2.04.024 have all been released.

In summary, using our approach, we found that both size and incidence of common coupling in Linux grow linearly with time, and that the choice of independent variable for longitudinal studies of open source software should be time, rather than version number.

# CHAPTER VIII

# CODE METRICS

## Contribution of this Chapter

4. A number of “classical” metrics have been argued to be of questionable value as general predictors of software quality and maintainability. We performed a validation study using a longitudinal analysis of the Linux kernel to determine if metrics of questionable general validity might have increased merit when applied to sequential versions of a single product. We concluded that only LOC (Lines of Code) was useful as a predictor of common coupling, which has previously been validated with respect to various quality and maintainability attributes. McCabe’s Cyclomatic Complexity, Halstead’s Volume and Effort, and Oman’s Maintainability Index all failed to consistently have a statistically significant correlation with common coupling, the “gold standard.”

## 8.1 Introduction and Metrics Overview

In this section we briefly review the various metrics we measure. This overview is a summary of material presented and discussed in more detail in Chapter 2. The metrics we consider are:

- Lines of Code (LOC)

- McCabe’s Cyclomatic Complexity (MCC)
- Halstead’s Software Science Metrics (Volume and Effort)
- Oman’s Maintainability Index (MI)
- Common Coupling
- Klocwork’s Risk Metric

### 8.1.1 Lines of Code (LOC)

This metric was discussed in detail in Section 2.3.4. For “lines of code,” we measure the total number of non-comment, non-blank (NCNB) lines of code from the preprocessed source code of all files for a consistently-configured kernel, as discussed in Section 7.8. The number of NCNB lines for each file is provided directly by Klocwork. All subsequent references to “LOC” are considered synonymous with “NCNB Lines,” unless otherwise explicitly stated.

### 8.1.2 McCabe’s Cyclomatic Complexity (MCC)

Cyclomatic Complexity was discussed at length in Section 2.3.2. Our values of Cyclomatic Complexity are reported directly by Klocwork. We sum the Cyclomatic Complexity from all files in a given version of the kernel.

### 8.1.3 Halstead Software Science Metrics (Volume and Effort)

As discussed in Section 2.3.3, computing Halstead's Volume and Effort metrics begins with the following four "primitive" metrics, which are collected for us by Klocwork (the corresponding Process-Level Metrics, or PLM numbers, are shown in parentheses):

[8.01] $N_1$ = Total number of operators (PLM #P0132)
[8.02] $N_2$ = Total number of operands (PLM #P0130)
[8.03] $n_1$ = Number of unique operators (PLM #P0133)
[8.04] $n_2$ = Number of unique operands (PLM #P0131)

Halstead combined the number of unique operands and operators, and called this the "vocabulary" from which the program is composed. Similarly, the total Vocabulary and Length of a program can be expressed as the total number of operators and operands used:

[8.05] $n = n_1 + n_2$ (Vocabulary)
[8.06] $N = N_1 + N_2$ (Length)

From these, we can derive the higher-level metric Volume (V):

[8.07] $V = N\log_2(n) = (N_1 + N_2)\log_2(n_1 + n_2)$

Given that the vocabulary can be represented by tokens whose individual Lengths are $\log_2(n_1 + n_2)$ bits, multiplying the Length (the number of times the tokens appear) by the number of bits required to represent the tokens gives a measure of the total *Volume* of the code.

Halstead defined Effort in terms of alternate versions of the $N_1$, $N_2$, $n_1$, and $n_2$ metrics he called the "potential" operator and operand counts (both total and unique). These were defined rather loosely by Halstead:

> "The most succinct form in which an algorithm could ever be expressed would require the prior existence of a language in which the required operation was already implemented, perhaps as a subroutine or procedure. In such a language, the implementation of that algorithm would require nothing more than the naming of operands for its arguments and resultants." [Halstead 1977].

It would seem that, in a well-structured program, the "number of operands for [the algorithm's] arguments and resultants" would be simply the number of parameters in the argument list for the function implementing the algorithm. In a more modern setting, when applied to a non-trivial piece of code, however, the situation becomes somewhat less clear.

For the trivial algorithms Halstead used as examples, it was easy to examine the algorithm and determine the entire set of input and output parameters. For something like the memory management subsystem in Linux, however, the entire set of input and output parameters is considerably less obvious. Moreover, for complex code broken into multiple functions, we ultimately have the question of granularity – should we consider each function independently, or should we consider only the high-level algorithm?

For example, one could argue that Quicksort's parameters consist entirely of the list to sort and the sort direction (ascending or descending), but Quicksort is typically implemented as a high level, recursive "driver" and a helper function to partition the subsets of the list. From Halstead's definition, it is unclear whether we should consider Quicksort to be the entire algorithm, viewed as a "black box" or as two algorithms, each handling different levels of detail. If we use the latter approach, then we have two parameter lists to consider, rather than one.

The existence of global variables (and module-level `static` variables) makes it impossible to programmatically examine the true number of parameters to a function, and the number of functions we have to consider makes manual examination impractical. Therefore, computing Effort, purely as Halstead defined the required "potential" parameters, is not practical.

Fortunately, however, the SEI (Software Engineering Institute) defines the *difficulty* (D) of a module as[4]:

$$[8.08] \quad D = \frac{n_1}{2} * \frac{N_2}{n_2}$$

This measure is proportional to the number of unique operators ($n_1$), and also to the rate at which the same operands are used repeatedly ($N_2 / n_2$). Programs with a large number of operators, and/or in which the same operands are used many times would seem to inherently be more prone to errors, and thus more difficult to develop and maintain.

[4] http://www.sei.cmu.edu/str/descriptions/halstead_body.html, accessed March 16, 2008.

Using this definition for Difficulty leads us directly to the equation for Effort: Volume times Difficulty:

$$[8.09] \quad E = V * D = [(N_1 + N_2)\log_2(n_1 + n_2)] * \left[\frac{n_1 N_2}{2n_2}\right]$$

One condition under which it is impossible to compute the Effort for a given piece of code is if $n_2$ is zero (because we cannot divide by zero). Similarly, we cannot compute Volume if both $n_1$ and $n_2$ are zero (because $\log_2(0)$ is not defined). As we will see in Section 8.1.3.1, this second condition occurs frequently in Linux.

Klocwork computes the Halstead primitives, but it does not provide the derived Effort or Volume. Accordingly our first task was to compute these values.

#### 8.1.3.1 Functions defined with the `extern inline` attribute

A large number of the functions in the 2.00.x and 2.02.x kernels are defined using the `extern inline` attribute. The `gcc` compiler documentation[5] says the following regarding the `extern inline` attribute:

> "By declaring a function `inline`, you can direct GNU `gcc` to integrate that function's code into the code for its callers. This makes execution faster by eliminating the function-call overhead. … If you specify both `inline` and `extern` in the function definition, then the definition is used only for inlining. In no case is the function compiled on its own, not even if you refer to its address explicitly. … This combination of `inline` and `extern` has almost the effect of a macro."

[5] http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc_4.html#SEC92, accessed March 16, 2008

In other words, `extern inline` functions are code templates that become executable code only within the confines of the function that calls the `extern inline` function; they are not compiled and linked like "regular" functions. A non-obvious side effect of this handling of `extern inline` functions is that, because they do not represent a single instance of executable code, they have no executable statements (in and of themselves), and they therefore have no operators and no operands with respect to computing Halstead's metrics. If a "regular" function `X` calls some `extern inline` function `Y`, then, at the point of the call to `Y`, the operator counts, operand counts, statement counts, and all other metrics associated with `Y` are registered as occurring within function `X`.

A natural place for a programmer to use `extern inline` is for low-level library functions, particularly if they involve assembler code. Such functions are typically short, and the performance penalty imposed by the overhead of the subroutine call linkage is highly undesirable. In fact, the call overhead may even require more machine instructions than the code in the function being called. Because such functions are, by definition, very low-level routines, they tend to be found buried near the bottom of the `#include` tree. Consequently, they also tend to appear in the majority of the files after preprocessing.

An extremely large proportion of the functions (approximately 62 percent) in the 2.00.x and 2.02.x series kernels were defined with the `extern inline` attribute. There was eventually a concerted effort to reduce the number of `extern inline` functions, apparently initiated some time late in the 2.03.x development cycle. The 2.04.000 kernel, despite being considerably larger, had only 17.68 percent of its functions defined with `extern inline`, and by the time 2.04.035 was released, the percentage of `extern inline` functions was down to 0.41 percent.

The net effects of the `extern inline` functions are as follows:

1) There are lines of code present in the preprocessed code base that do not necessarily produce object code.
2) When computing the operator and operand counts for Halstead's metrics, the contribution of code in `extern inline` functions appears in the functions from which the `extern inline` function is called, rather than the `extern inline` function itself.
3) Because the preprocessor still `#include`s the `extern inline` functions, they are part of the code base, and do count towards the total number of lines, blank lines, comment lines, and NCNB lines.
4) Because the `extern inline` functions appear in `#include` files, they still require maintenance, but to the extent that they appear in files in which they might not actually be used, they inflate the line count. Therefore, our line counts are upper bounds on the actual number of lines of code to be maintained.

### 8.1.4 Oman's Maintainability Index (MI)

As discussed in Section 2.3.5, Oman's Maintainability Index (MI) is a measure of the maintainability of a piece of code. The lower the MI, the more difficult (and, presumably, costly) it is to maintain that code. Oman did not explicitly discuss what kind of maintenance his index described; however, we assume that it was intended to be broadly applicable to corrective, perfective, and adaptive types of maintenance.

Oman's MI was designed as a 100-point scale, with a minimum value of 25 (low maintainability, or more difficult/time-consuming/costly to maintain) and a maximum value of 125 (high maintainability).

### 8.1.5 Common Coupling

Common coupling, the "gold standard" metric against which we performed our validation of the above metrics, is discussed at length in Section 2.3.1. We validate all other metrics against common coupling.

### 8.1.6 Klocwork's Risk Metric

As discussed in Section 2.3.6, the Klocwork CASE tool we used has its own "risk" metric, that predicts, on a scale from 0 to 1, "the likelihood of fault insertion during modification or being inherent when created." This is some sort of compound metric, derived from multiple lower-level metrics, presumably in a manner similar to Oman's MI. The precise formula used is proprietary and confidential. Klocwork's documentation indicates that the desirable range for this metric is 0.0 – 0.1, which is called the "green" range. The second range, 0.1 – 0.2, is color-coded "yellow," presumably to indicate "warning." The third range, 0.2 – 1.0, is color-coded "red," presumably for "high risk."

This metric is defined at the file level. The remainder if this chapter deals with metrics at the kernel level. For this reason, we examine Klocwork's Risk metric now.

Figure 8.1 shows a histogram of this metric, taken over all files in all kernel series (2.00.x, 2.02.x, and 2.04.x):

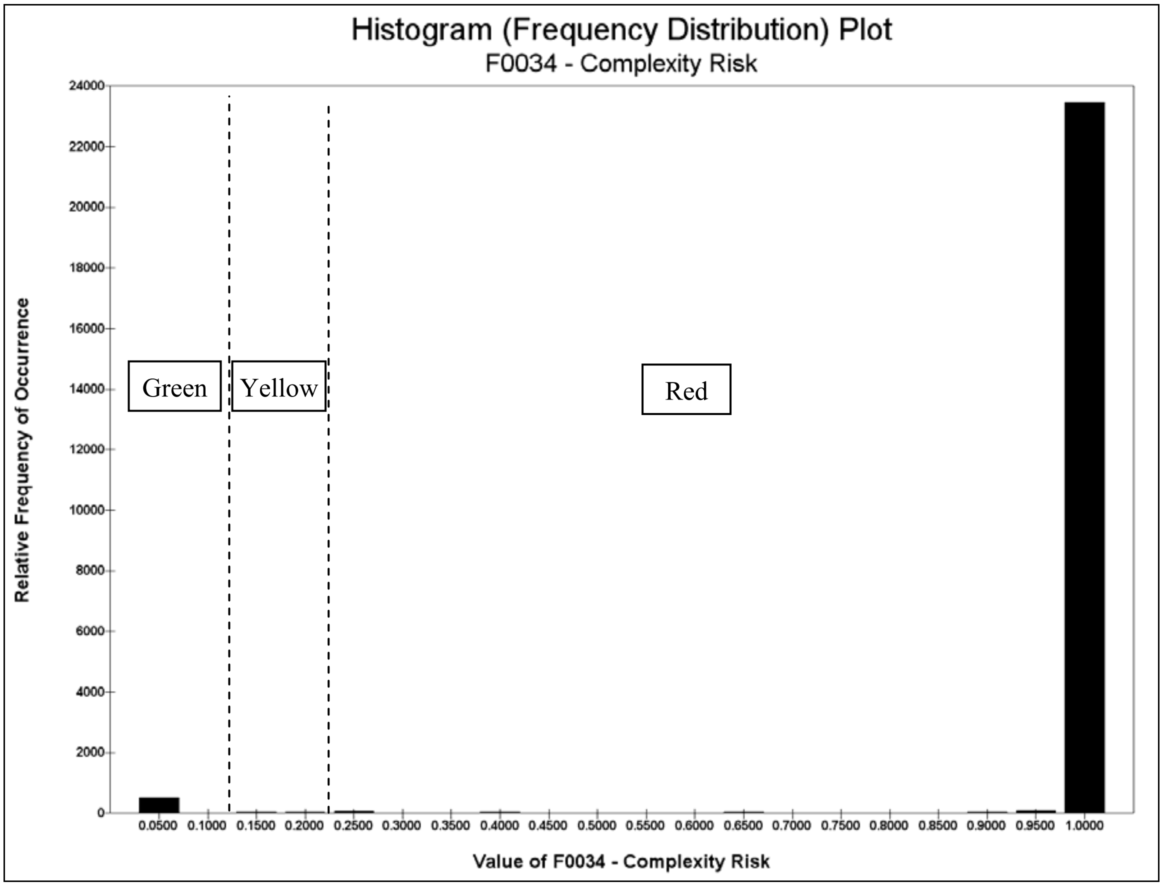


**Figure 8.1 – Distribution of Klocwork's Risk metric**

Figure 8.1 clearly shows that, regardless of the kernel series, the overwhelming majority (96.21 percent) of the values are not only in the "red" range, but are actually at the full-scale value of 1.000. We performed a second frequency distribution, placing the Risk values into 100 buckets, each of width 0.01. Table 8.1 shows the result. Rows corresponding to buckets with zero samples have been omitted.

**Table 8.1 – Frequency Distribution of Klocwork's Risk Metric**

| Bucket | Number of Files | Percentage of Total | Cumulative Percentage |
|---|---|---|---|
| 0.06 | 392 | 1.61 | 1.61 |
| 0.08 | 120 | 0.49 | 2.11 |
| 0.16 | 40 | 0.16 | 2.27 |
| 0.22 | 30 | 0.12 | 2.39 |
| 0.26 | 45 | 0.19 | 2.58 |
| 0.28 | 8 | 0.03 | 2.61 |
| 0.39 | 38 | 0.16 | 2.77 |
| 0.50 | 18 | 0.07 | 2.84 |
| 0.65 | 38 | 0.16 | 3.00 |
| 0.75 | 5 | 0.02 | 3.02 |
| 0.77 | 2 | 0.01 | 3.03 |
| 0.80 | 2 | 0.01 | 3.04 |
| 0.84 | 18 | 0.07 | 3.11 |
| 0.89 | 16 | 0.07 | 3.18 |
| 0.93 | 6 | 0.02 | 3.20 |
| 0.94 | 63 | 0.26 | 3.46 |
| 0.96 | 5 | 0.02 | 3.48 |
| 0.98 | 8 | 0.03 | 3.51 |
| 0.99 | 66 | 0.27 | 3.79 |
| 1.00 | 23,382 | 96.21 | 100.00 |
| Total: | 24,302 | | |

Because Klocwork's Risk metric so overwhelmingly categorizes the majority of the source files in all three series of the Linux kernel as being at maximum risk, we conclude that one of two things must be true:

1) The entire Linux kernel is as fragile as a house of cards, or
2) This metric is of severely limited value as we applied it.

We applied this metric to preprocessed C source code. If Klocwork's Risk metric is sensitive to some attribute inherent in preprocessed code that would not otherwise be present in the raw (unpreprocessed) source code, it is possible that the distribution we obtained is due to our using preprocessed code, rather than raw source.

Regardless of the source of this unusual distribution, the distribution itself is so strongly biased towards the maximum of its range that we are unable to attempt to use it for any further analysis.

## 8.2 Computing the Metrics at the Kernel Level

Once the metrics of interest have been identified for a longitudinal analysis, the next question becomes one of *granularity*. In this context, "granularity" refers to whether we are considering a metric (both its *measurement* and its *interpretation*) at a high level or a low level, with respect to the body of code being measured. Metrics captured at the highest level would be considered kernel-wide. An entire (single) version of the Linux kernel would be considered a project, which is made up of any number of files (or modules). The files are then, in turn, composed of zero or more functions (or procedures or routines), each made up of some number of individual lines of source code. Obviously, we could even parse the individual lines into tokens, and then classify the tokens as keywords, identifiers, operators, and so forth, but for our purposes, we examined the code at the kernel (or project) level.

When considering an entire project such as the Linux kernel, in order to compare some aspect of one version of the kernel with another version, the most natural, convenient, and simple way to make such a comparison is to gather a single metric from each version as a whole, and then compare the two numbers (assuming, of course, that there is some scale on which "greater than," "less than," and "equal to" are defined for the metric in question).

Implicit in this approach, however, is that in examining two versions, we *must* be able to distill the metric in question down to a single number, which will hopefully be representative of the entire kernel.

Some metrics are *defined* at a single level, below which they are undefined, and above which they must be aggregated to be compared; other metrics are defined at multiple levels. We now examine our code metrics as measured at the kernel level, individually.

### 8.2.1 Lines of Code (LOC)

The number of lines of code is unambiguously defined at the kernel, file, *and* function levels. We can readily count the number of lines in a given function. Similarly, we can count the total number of lines in a file, and the total number of lines across all files within a given version of the kernel. We used the total number of NCNB lines from all files in a given kernel version as our value for the LOC metric.

### 8.2.2 McCabe's Cyclomatic Complexity

McCabe's Cyclomatic Complexity (MCC) is defined as the number of independent paths through a piece of executable code. Because the smallest autonomous unit of executable code is the function, MCC is defined at the function level. McCabe has defined the way to aggregate the MCC value – the MCC of multiple functions is their sum [McCabe, 1976]. Therefore, we sum the MCC values for all functions in a file to get the file's MCC value, and we sum the MCC values from the file level to obtain a kernel-level value of MCC.

### 8.2.3 Halstead's Software Science Metrics (Volume and Effort)

Halstead's Volume and Effort metrics are defined at the function level. We aggregate them to the file level by summing them over all functions in the file. This makes sense intuitively – the volume of a group of functions is the sum of the volume of the individual functions, and the effort required to maintain a group of functions is the sum of the effort required to maintain the individual functions. Likewise, we aggregate Halstead's values to the kernel level by summing the respective values from the file level.

### 8.2.4 Oman's Maintainability Index (MI)

Oman's Maintainability Index (MI) is defined at the file level. As such, the MI value for a single function is undefined – we cannot take the definition of a metric down to a finer level of granularity than that at which a metric is originally defined.

Unlike LOC, MCC, Volume, and Effort, we cannot aggregate the MI upward to a coarser level of granularity simply by summing the MI values across all component files. Oman's original MI was designed to fit data to a 100-point scale, ranging from 25 to 125. Therefore, summing multiple instances of MI values is nonsensical. For this reason, and the fact that MI itself is based on averages, we use the average MI value, taken over all files in a kernel, to aggregate the MI value from the file level to the kernel level.

### 8.2.5 Common Coupling

Common coupling occurs between two (or more) files when both (all) files reference the same global variable. As in [Schach et al, 2002], we compute the total number of instances of common coupling by considering a global variable only once per file (multiple occurrences of a given global variable within a file count as one instance). We then count the total number of files sharing the global variable in question, and compute the number of instances as discussed Section 2.3.1. These values are summed across all global variables used within a given version of the kernel to obtain a single value that represents the total number of instances of inter-file common coupling kernel-wide.

Now that we have established a means of measuring each of these metrics at the kernel level, can perform our validations of each metric with respect to common coupling. First, however, we examine the range and distribution of each of these metrics.

## 8.3 Examination of the Individual Metrics at the Kernel Level

In this section, we present an overview of the respective individual metrics at the kernel level, examining their range, approximate distributions, and rates of change. All three Linux stable series (2.00.x, 2.02.x, and 2.04.x) appear together in the figures of Section 8.3. This is so that the reader can visually compare the three series; the figures should *not* be interpreted as depicting one overall series (see Section 7.7).

### 8.3.1 LOC (NCNB Lines)

As shown in Chapter 7, the size of the Linux kernel grows linearly within a series, with much less variability within a series than exists between series. Figure 8.2 shows the number of LOC from each of the three kernel series.

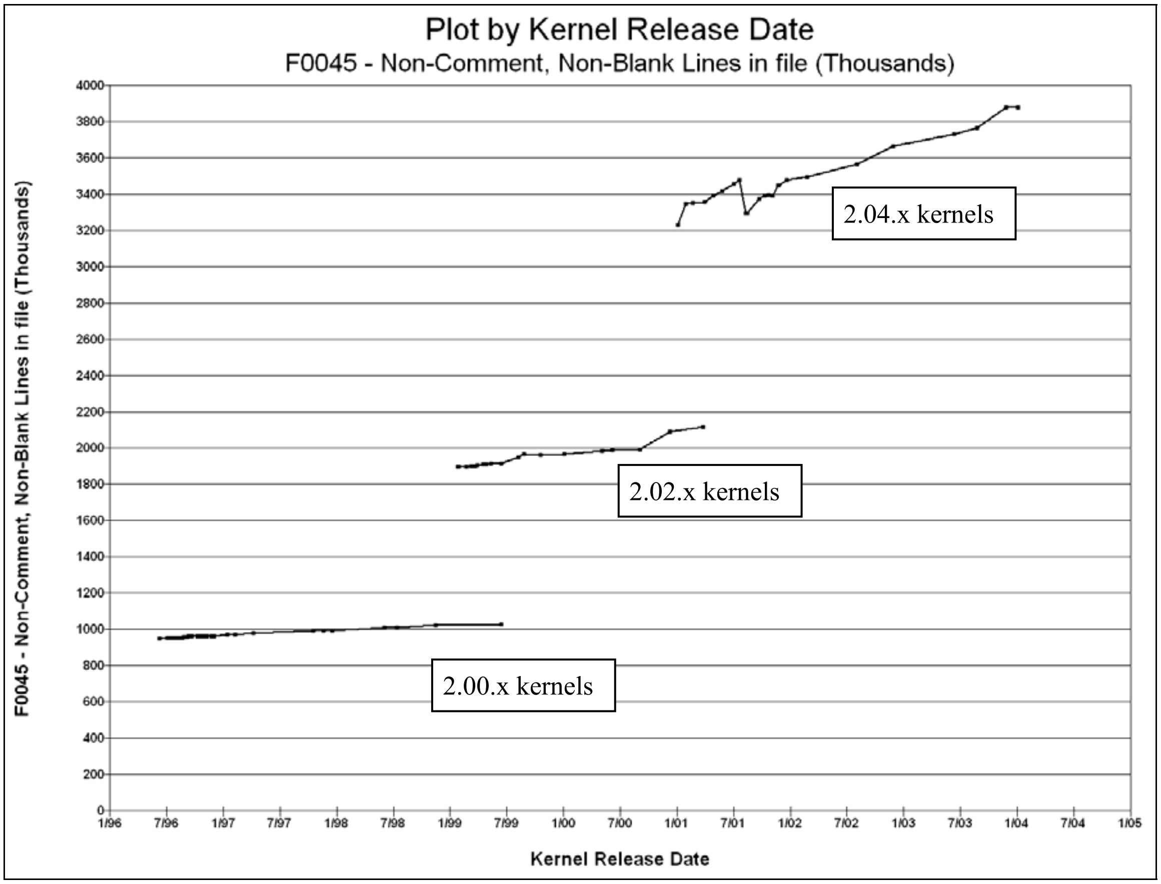


**Figure 8.2 – LOC in each kernel, plotted by series.**

Figure 8.1 shows that the 2.00.x series kernels contained approximately 1,000,000 LOC, the 2.02.x series contained about 2,000,000 LOC, and the 2.04.x series contained about 3,500,000 LOC. We reiterate that "LOC" is synonymous with "NCNB" lines for our purposes.

### 8.3.2 McCabe's Cyclomatic Complexity (MCC)

Because the MCC is proportional to the number of branches in the code, it stands to reason that the 2.02.x and 2.04.x kernel series, with their larger code bases, would also have higher MCC values. This is confirmed in Figure 8.3, which shows the MCC values for the respective kernels in each series.

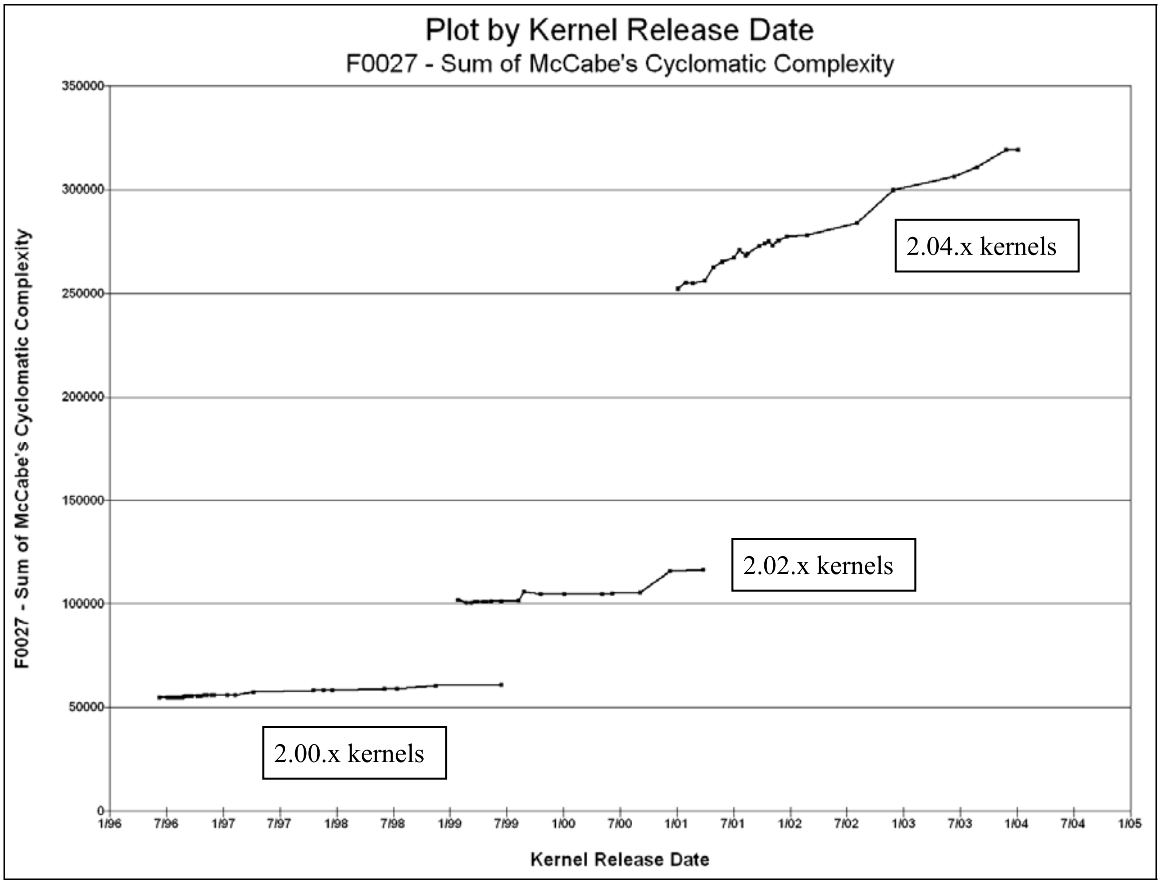


**Figure 8.3 – MCC in each kernel, plotted by series.**

Just as with LOC in Figure 8.2, Figure 8.3 shows seemingly linear growth in each kernel series, with more difference between series than within series. We are not concerned with the actual rate of growth in this chapter; rather, we are concerned with correlating the respective metrics with common coupling, for each of the three stable series separately. Plotting the actual values in this section is simply an exercise to familiarize ourselves with the high-level behavior of these metrics; in-depth analysis follows in Section 8.4.

### 8.3.3 Halstead Volume

Figure 8.4 shows the total Halstead Volume for all kernels in all three series.

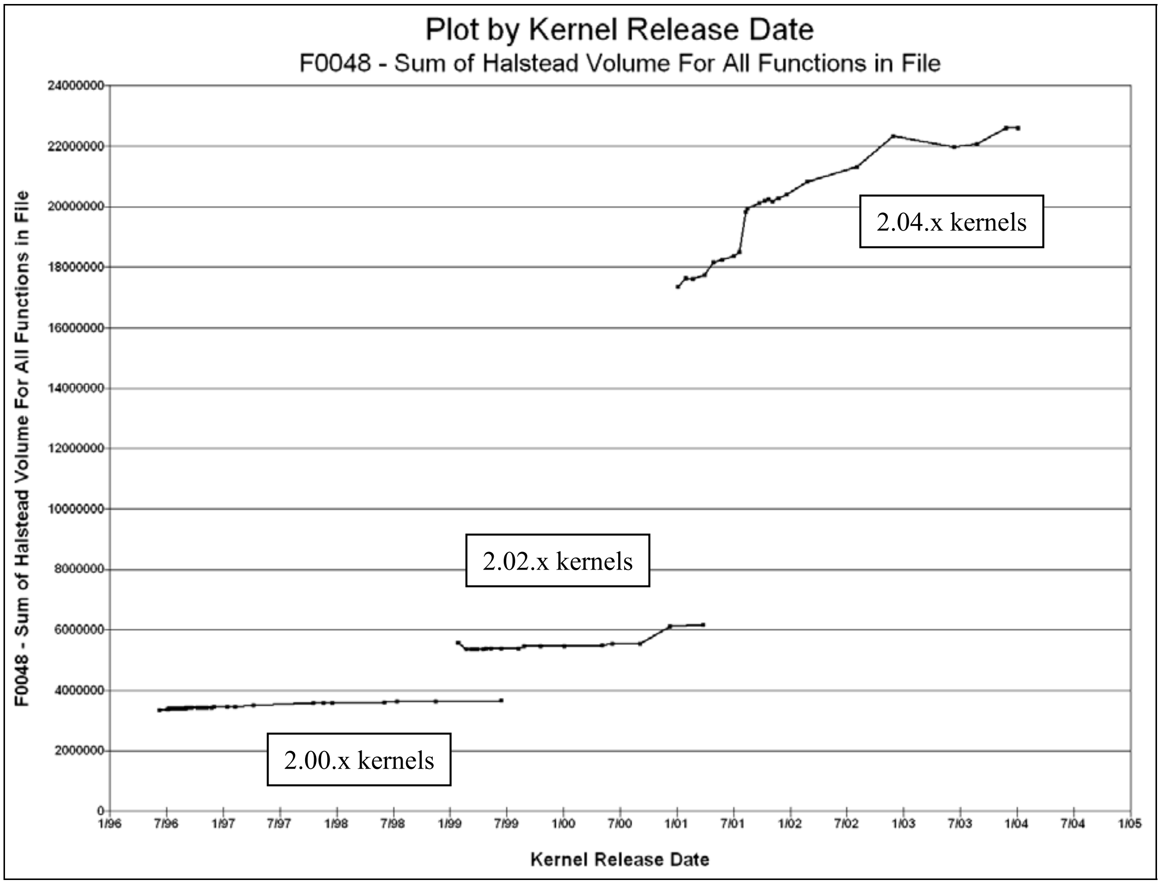


**Figure 8.4 – Halstead Volume in each kernel, plotted by series.**

Figure 8.4 shows a trend similar to that in Figure 8.3 – much less within-series variability among the total Halstead Volume values than between-series, with increasing levels between the 2.00.x, 2.02.x, and 2.04.x series. The Volume of the 2.02.x series of kernels is much closer overall to that of the 2.00.x series than it is to the 2.04.x series; this is probably due to the differences in the number of `extern inline` functions, as described in Section 8.1.3.1.

### 8.3.4 Halstead Effort

Figure 8.5 shows the total Halstead Effort for all kernels in all three series.

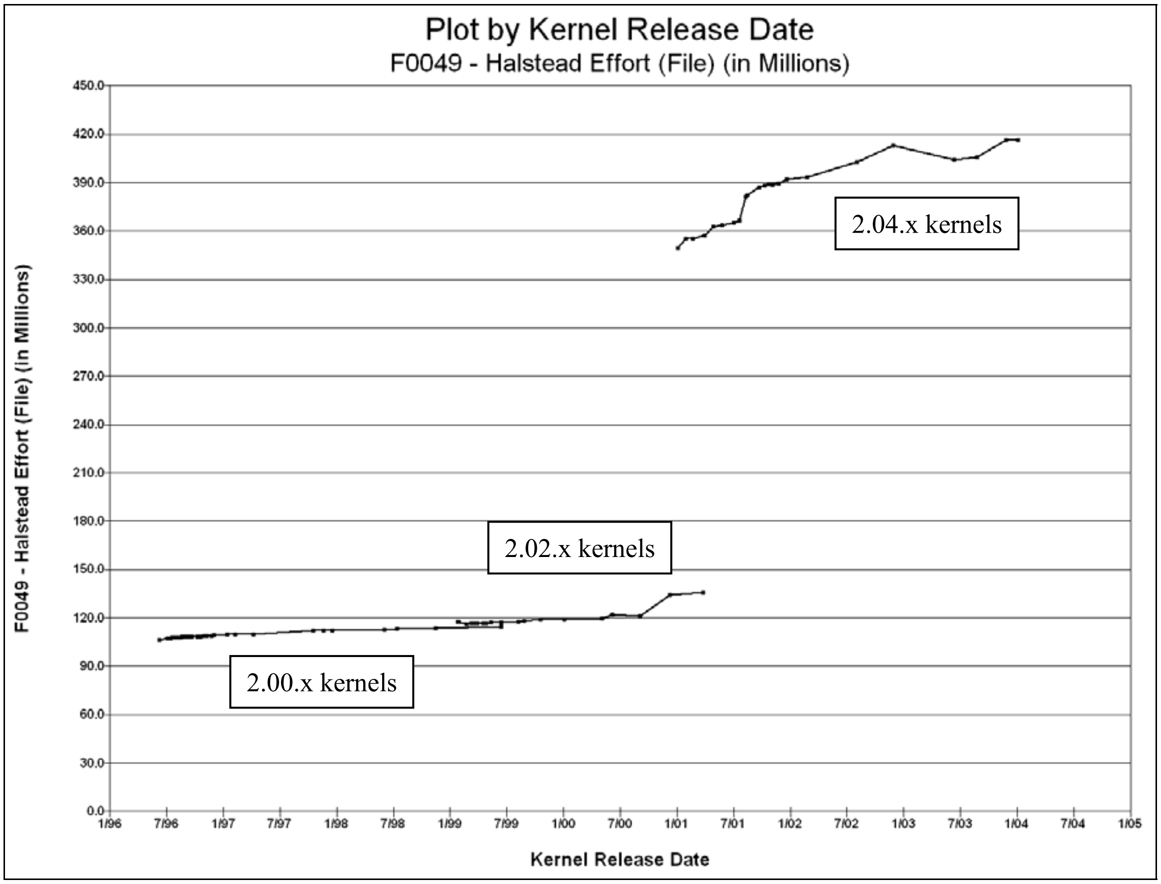


**Figure 8.5 – Halstead Effort in each kernel, plotted by series.**

Figure 8.5 shows generally linear growth within a series, and differences between series, but the relative differences between the overall average values in the 2.00.x and 2.02.x series is much less than we have observed in the previous metrics. Again, this is probably due to the differences in the number of `extern inline` functions, as described in Section 8.1.3.1.

### 8.3.5 Oman's Maintainability Index (MI)

Figure 8.6 shows the average Maintainability Index for all kernels in each of the three series.

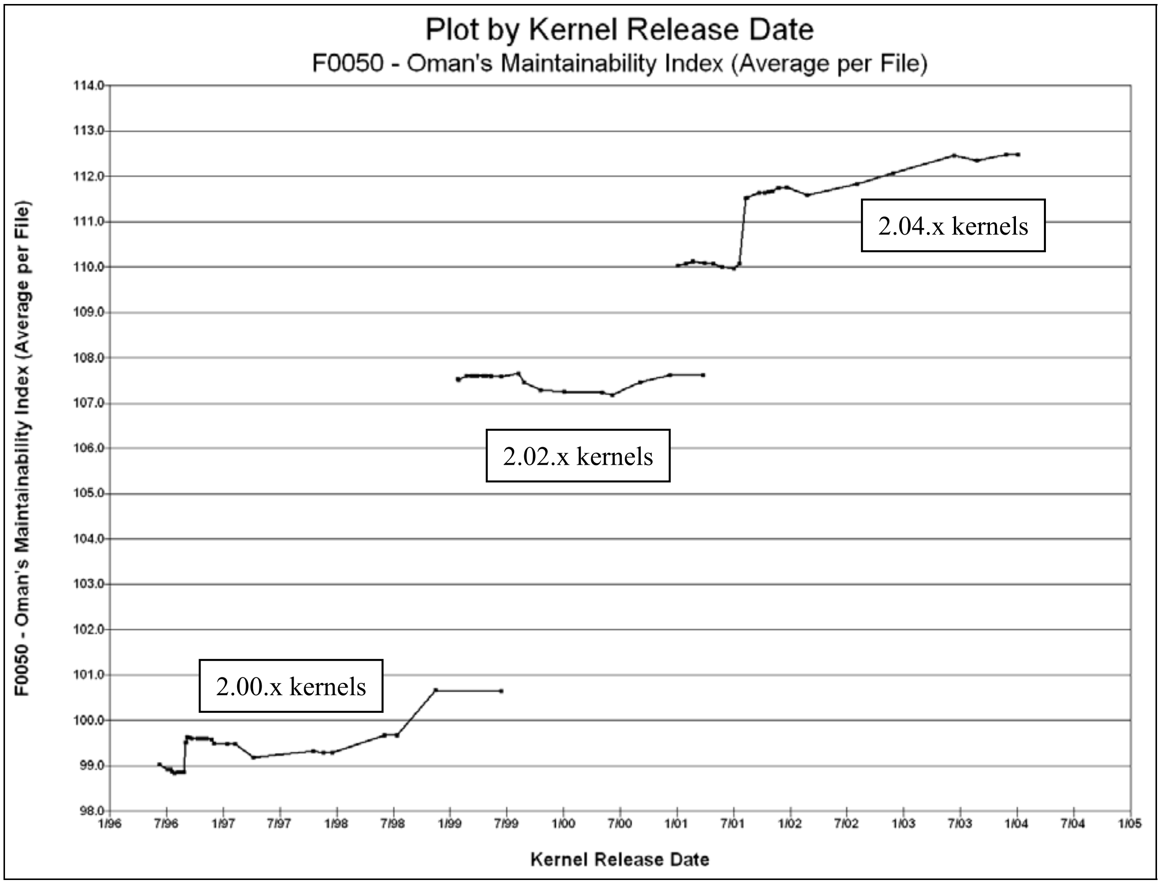


**Figure 8.6 – Oman's Maintainability Index (MI) in each kernel, plotted by series.**

Figure 8.6 shows that the MI values display much less variability within kernel series than they do between series, and that the MI values rose between the 2.00.x and 2.02.x series, and then again between the 2.02.x and 2.04.x series, but by a lesser margin.

The unexpected (and non-obvious) result revealed in Figure 8.5 is that MI is rising in the three kernel series. MCC measures complexity, where higher values are undesirable. LOC, Volume, and Effort are all directly influenced by the size of the code base. With ever-increasing functionality as the kernels have evolved, we naturally expect the code base to grow, and therefore, these other metrics would be expected to rise, and Figures 8.2 through 8.5 confirm this. Larger, more complex programs are inherently more difficult to maintain, but MI kept rising. MI runs in the opposite direction to that of the other metrics, with low values of MI corresponding to lower maintainability, so low values of MI are undesirable. With the other metrics rising (where higher values are undesirable), we would expect MI also to be moving towards the undesirable end of its range; i.e., falling.

As discussed in Section 2.3.5, the formula for computing MI is:

$$MI = 171 - 5.2\ln(\overline{HV}) - 0.23\overline{MCC} - 16.2\ln(\overline{LOC}) + 50\sin(\sqrt{2.46\,perCM})$$

*where:*

$\overline{HV}$ is the average Halstead Volume per module for the program.
$\overline{MCC}$ is the average McCabe's Cyclomatic Complexity value.
$\overline{LOC}$ is the average number of lines of code per module for the program.
$perCM$ is the percentage of lines containing comments in the program.

The coefficients on the Volume, MCC, and LOC terms are all negative, so we would expect MI to decrease as the code base gets larger. The only term that raises MI is the one related to the number of comments.

We ignore blank lines, and compute perCM as $\frac{comment}{comment + NCNB}$ lines. For the three kernel series (2.00.x, 2.02.x, and 2.04.x), we have perCM values of approximately 0.390, 0.435, and 0.480, respectively, which makes the $50\sin\left(\sqrt{2.46\,perCM}\right)$ term 41.17, 42.65, and 43.94, respectively. The MI values in Figure 8.5 have a range of approximately 13 units, and documentation term has a range of only approximately +2.8 units (as we move from 2.00.x to 2.04.x), so 80 percent of the variation still remains unidentified.

The answer lies in the fact that MI uses *averages* of the Volume, MCC, and LOC values, and Figures 8.2 through 8.5 show the *sums* of these metrics. The average Volume for all functions in all kernel versions is shown in Figure 8.7.

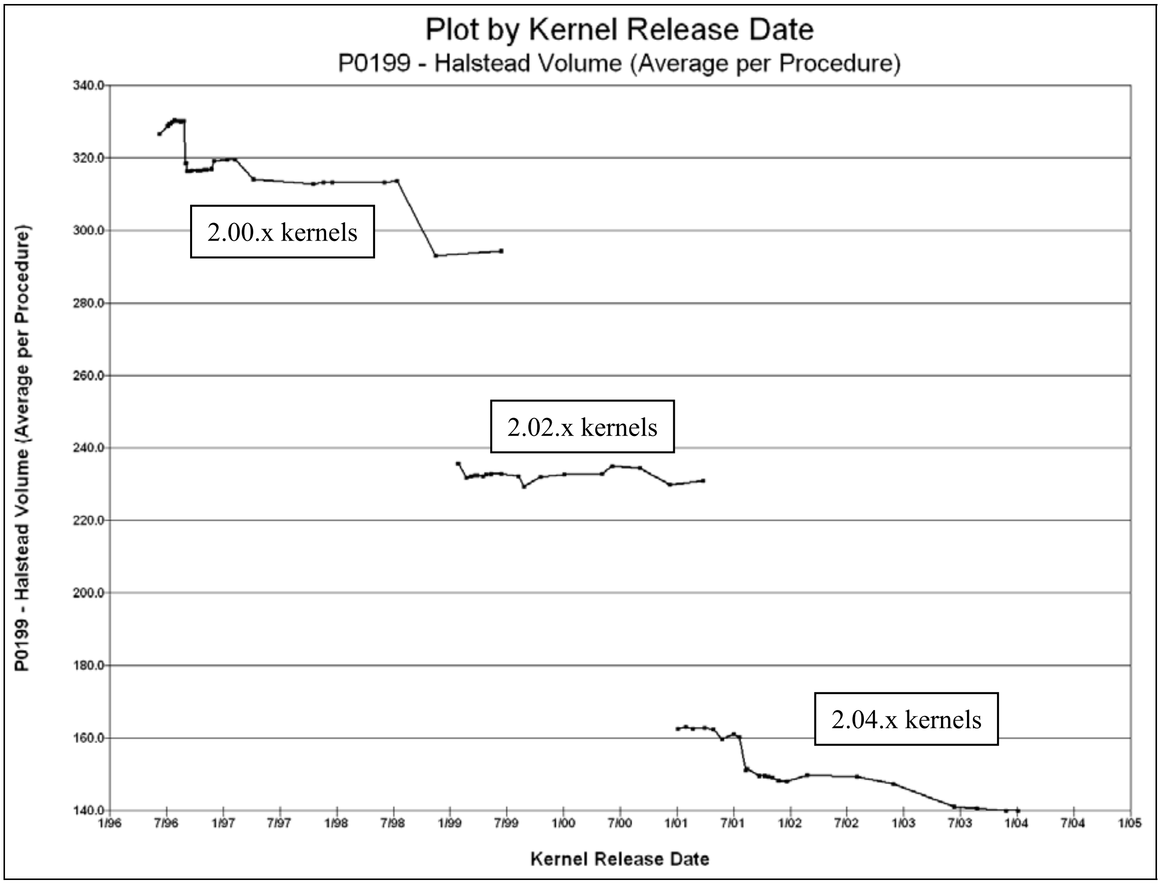


**Figure 8.7 – Average Volume for all functions in each kernel, plotted by series.**

Figure 8.7 shows that the average Volume decreases from series to series. The three series have average Volume values of approximately 320, 235, and 150, respectively. The term in the MI formula related to Volume is -5.2 ln(aveV). This corresponds to contributions of -30.0, -28.4, and -26.1 units, respectively. The volume-related term in the MI formula therefore accounts for approximately +3.9 (as we go from the 2.00.x to 2.04.x series) of the 13 units of variability in Figure 8.6.

Similarly, the average LOC and MCC values are shown in Figures 8.8 and 8.9.

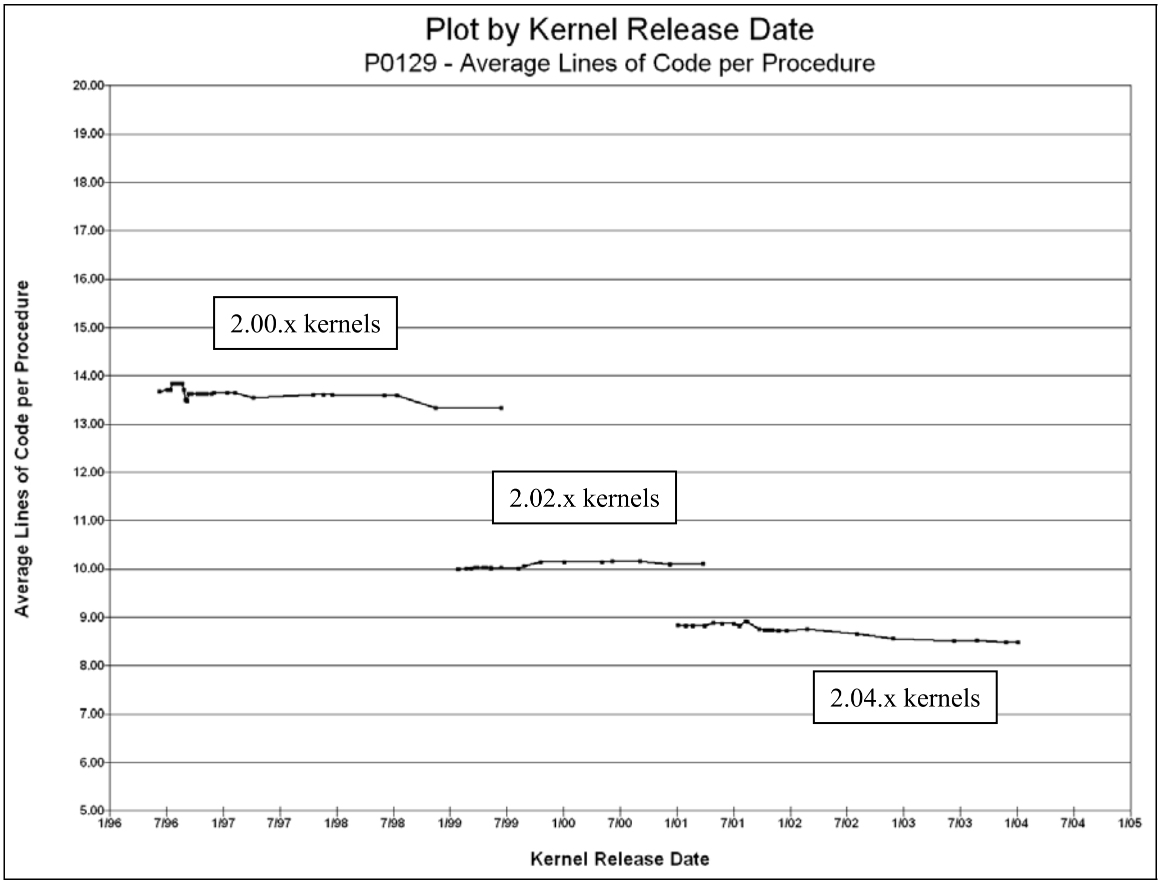


**Figure 8.8 – Average LOC (NCNB Lines) for all functions in each kernel, plotted by series.**

Figure 8.8 shows that the average LOC per function decreases from series to series.

The LOC averages range from approximately 13.5 in 2.00.x kernels to approximately 8.5 in the 2.04.x kernels. The term in the MI formula related to LOC is –16.2 ln(LOC). The average LOC terms for the three kernel series are therefore approximately –42.2, –37.3, and –34.7 units, respectively. The LOC term, therefore, contributes approximately +7.5 of the 13 units of variation we observed between the 2.00.x and 2.04.x series of kernels.

Finally, Figure 8.9 shows the average MCC value per function in all three kernel series.

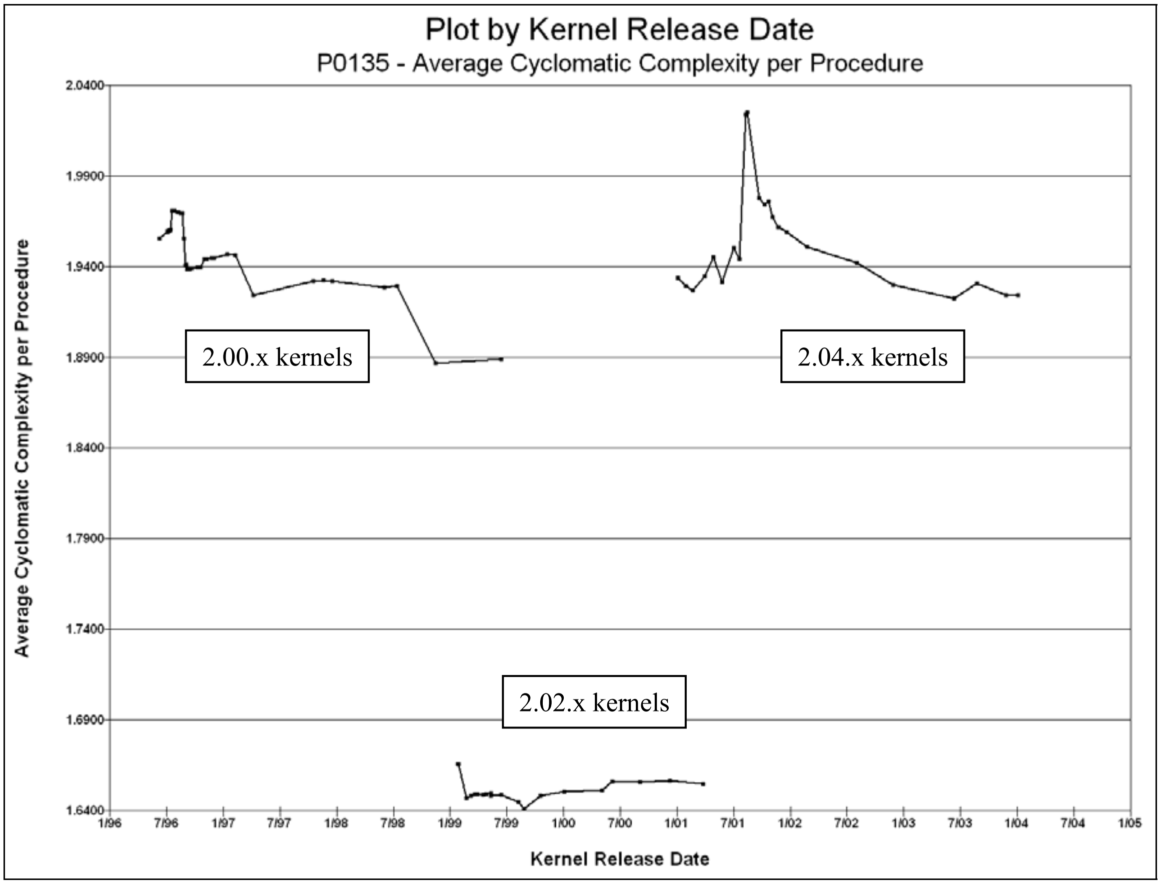


**Figure 8.9 – Average MCC for all functions in each kernel, plotted by series.**

Figure 8.9 shows that the average MCC values dropped sharply in the 2.02.x series kernels, but the 2.00.x and 2.04.x kernels had approximately the same average MCC values over the range of Figure 8.9. The cause for the drop in the 2.02.x series is not readily apparent.

The coefficient in the formula for MI on the average MCC term is –0.23. Given that the approximate ranges in Figure 8.9 are 1.94, 1.65, and 1.95 for the three series, respectively, the values of this term for the 2.00.x, 2.02.x, and 2.04.x series are –0.45, –0.38, and –0.45. Recall that the range in Figure 8.6 is approximately 13 units.

Accordingly, the 0.07-unit range caused by this shift in the average MCC value has an extremely minor impact on the total variation.

Figure 8.8 shows that the average number of LOC in a function consistently decreased from series to series, yet Figure 8.2 shows that the total number of lines in each kernel series increased. Clearly, the number of functions must have been increasing, and Figure 8.10 confirms this.

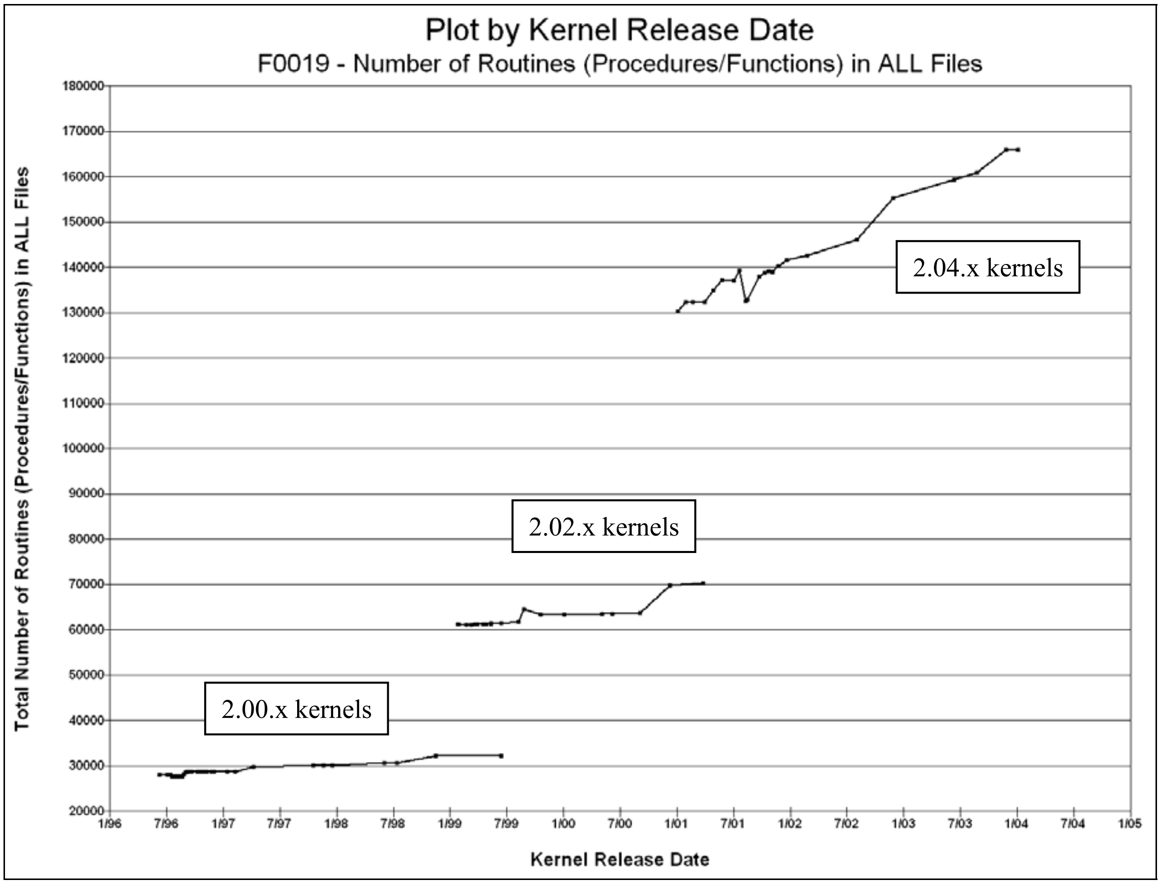


**Figure 8.10 – Total Number of functions in each kernel, plotted by series.**

If the average number of lines per function is going down, but the number of functions and the total number of lines per kernel are going up, then the increasing number of functions outweighs their decreasing average size.

From the discussions of Figures 8.7 through 8.10, we have determined that, of the approximately 13 units by which the MI rose between the 2.00.x and 2.04.x series kernels, we can attribute approximately…

…+3.9 units to changes in average Volume per function,

…+7.5 units to changes in average LOC per function,

… 0.0 units to changes in average MCC per function, and

…+2.8 units to changes in the percentage of lines containing comments.

Because these values were rough approximations, the total is greater than 13 units. It remains clear, however, that the majority of the variation is attributable to the decrease in the average LOC per function. Because the term in the MI formula related to the average LOC per function is –16.2 ln(LOC), and because the average LOC values were in the range of 8.5 to 13.5, relative to *e*, the base of natural logarithms ($e \approx 2.718$), and amplified by the constant of 16.2, this term dominates the computation. For the average number of LOC per function to range between only 8.5 and 13.5, there must be a huge number of functions with very few LOC pulling the average down. This would seem to be a weakness of MI – a change in the average LOC per function from 13.5 to 8.5 (a drop of 5 LOC per function) has the same 7.5-unit impact on MI as would a 100-LOC decrease from 270 to 170 LOC per function.

We have been unable to locate any studies on Oman's Maintainability Index in the presence of refactoring, specifically when the refactoring is targeted at reducing the average function size. It is unclear, as the average function size decreases and the total number of functions increases, which has a larger impact on maintainability. Put another way, are 1,000 ten-line functions categorically easier to maintain than 10 thousand-line functions? Somewhere along the way, it seems that managing the relationships between the extra functions must outweigh the resulting simplicity of having smaller functions.

Moreover, the questions posed by this hypothetical refactoring do not account for coupling between the modules. It is not clear whether the average size of the Linux kernel functions decreased from series to series as a result of intentional redesign of the code, or if this is an artifact of some other code-related phenomenon, such as the `extern inline` function issue discussed in Section 8.1.3.1.

The most egregious problem that this exercise points out is that MI is not at all sensitive to the number of functions. According to the MI formula, the entire Linux kernel is as "maintainable" as would be a single program containing one function of somewhere between 8 and 13 non-blank LOC, provided the Volume of that function is between about 140 and 320, and there are comments on about 40% of the non-blank lines.

### 8.3.6 Common Coupling

Figure 8.11 shows the total common coupling for all kernels in all three series.

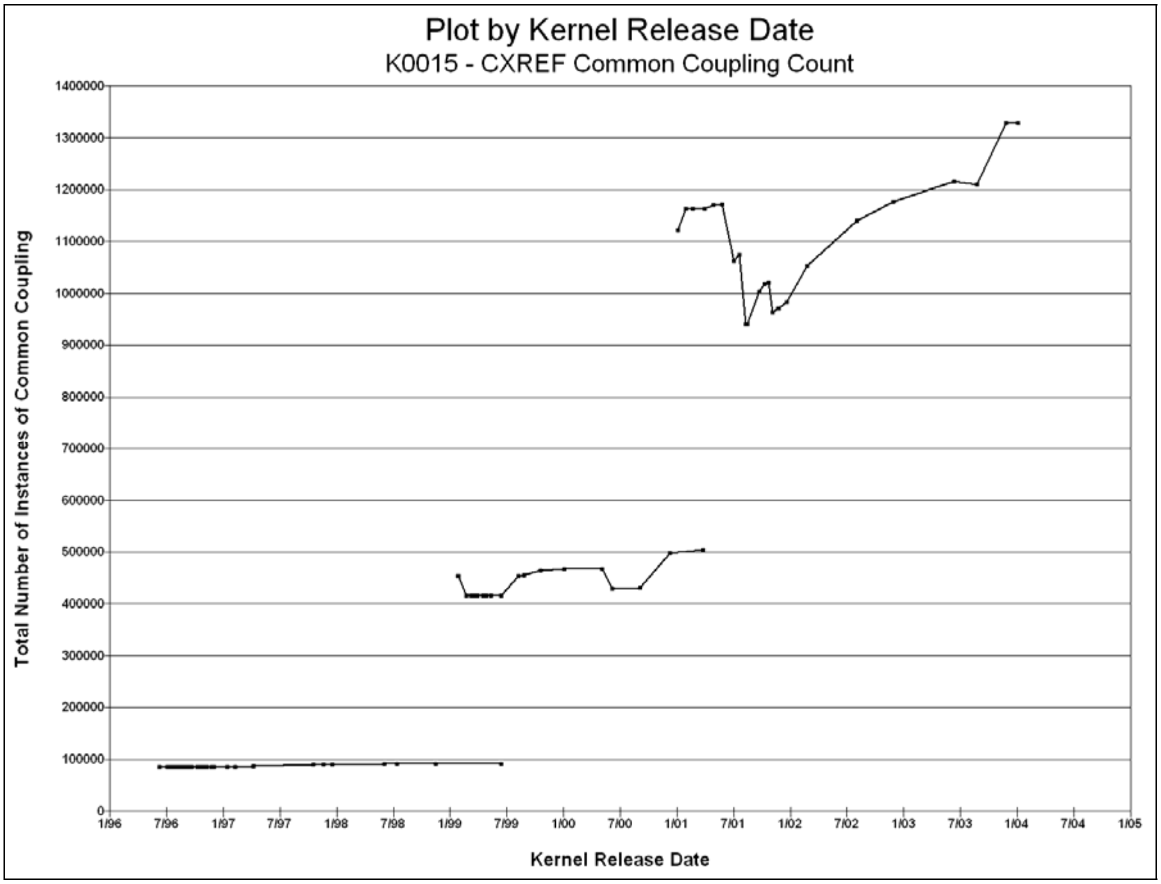


**Figure 8.11 – Sum of common coupling in each kernel, plotted by series.**

Figure 8.11 shows that the inter-file coupling rose between kernel releases, and that the volatility within each series also rose. We will examine this volatility, and its relationships with the other metrics, in Section 8.6.

## 8.4 Validation of Metrics against Common Coupling

In this section we consider each of our kernel-level metrics with respect to common coupling.

For each of the three kernel series, we computed the Spearman rank correlation between each of the metrics in Section 8.2 and common coupling (the "gold standard"), as well as the corresponding t-statistic and P-value. Table 8.2 summarizes these results. Entries in Table 8.2 corresponding to significant correlations (P < 0.05) appear in boldface italics, with a heavy grid line around the entry.

**Table 8.2 – Correlations of Individual Metrics against Common Coupling by Kernel Series**

| Correlation | 2.00.x (*N = 38*) | 2.02.x (*N = 20*) | 2.04.x (*N = 25*) |
|---|---|---|---|
| MCC vs Common Coupling | ***$\rho_s$ = 0.9821***<br>***t = 31.271***<br>***P < 0.001*** | ***$\rho_s$ = 0.8641***<br>***t = 7.2849***<br>***P < 0.001*** | $\rho_s$ = 0.2386<br>t = 1.1781<br>P < 0.3 |
| NCNB vs Common Coupling | ***$\rho_s$ = 0.9011***<br>***t = 12.467***<br>***P < 0.001*** | ***$\rho_s$ = 0.6173***<br>***t = 3.3287***<br>***P < 0.001*** | ***$\rho_s$ = 0.5020***<br>***t = 2.7838***<br>***P < 0.01*** |
| Volume vs Common Coupling | ***$\rho_s$ = 0.8882***<br>***t = 11.596***<br>***P < 0.001*** | ***$\rho_s$ = 0.8129***<br>***t = 5.9228***<br>***P < 0.001*** | $\rho_s$ = 0.2185<br>t = 1.0741<br>P < 0.3 |
| Effort vs Common Coupling | ***$\rho_s$ = 0.9025***<br>***t = 12.570***<br>***P < 0.001*** | ***$\rho_s$ = 0.8393***<br>***t = 6.5495***<br>***P < 0.001*** | $\rho_s$ = 0.2101<br>t = 1.0305<br>P < 0.4 |
| MI vs Common Coupling | ***$\rho_s$ = 0.7696***<br>***t = 7.2319***<br>***P < 0.001*** | $\rho_s$ = -0.2356<br>t = 1.0285<br>P < 0.4 | $\rho_s$ = 0.1962<br>t = 0.9597<br>P < 0.4 |

Table 8.2 shows that, for the 2.00.x series, all five of our metrics are very highly significantly correlated with common coupling (P < 0.001). In all cases, the correlation

coefficient is positive. In the case of MI, this reveals a serious problem; namely, that previous validation studies (see Section 1.3) have shown that as common coupling (the "gold standard") rises, maintainability *decreases*. However, we have obtained a *positive* correlation between common coupling and Oman's maintainability index. If maintainability, as predicted by common coupling, should decrease as common coupling rises, then we should obtain a *negative* correlation with Oman's maintainability index. Oman's maintainability index is very highly significantly correlated, but in the *wrong* direction. We have, therefore, invalidated Oman's maintainability index.

When we move to the 2.02.x kernels, we observe that four of the five metrics have correlations with common coupling that are very highly significant, but the t-values in each case are substantially lower than they were for the 2.00.x series. Accordingly, all four are again good measures of the maintainability of this data set. Oman's Maintainability Index (MI) does not have a statistically significant correlation with common coupling in the 2.02.x series of kernels.

Finally, we consider the 2.04.x series of kernels, in which the only metric showing a statistically significant correlation is NCNB ($P < 0.01$).

Why would the entire set of metrics correlate so strongly with common coupling in the 2.00.x series, while only four of the five do so in the 2.02.x series, and only one of the five in 2.04.x? There seem to be two primary reasons for this result:

1. The five metrics are all somewhat interrelated. Therefore, much of the correlation we observe is indirect.

   All things being equal, a larger code base is more likely to have more `if`, `for`, `while`, and `switch` statements than a smaller code base. This gives us a general proportionality between NCNB and MCC. Similarly, because Halstead's Volume (and by extension, Effort) is dependent on the Length of a program, more NCNB lines would naturally coincide with more Length, and therefore more Volume (and Effort). Because the formula for Oman's MI includes terms for MCC, Volume, and NCNB, we would expect a certain degree of correlation between MI and these other metrics. Therefore, there is a certain degree of correlation to be expected among this set of metrics. Accordingly, we expect all five metrics to behave in a similar fashion.

2. Oman's Maintainability Index (MI) was developed under questionable assumptions, and its validity is vulnerable to attack for several reasons:
   1) The formula for MI was developed specifically to obtain a fit to only eight data points.
   2) Oman developed "approximately 50 regression models," testing twenty different metrics. Three of those models were presented in [Oman, Hagemeister 1994], and one of those three was subsequently modified. It may have been purely coincidental that the eight points Oman used happened to fit the polynomials he published.

3) The data to which Oman tuned his formula was obtained from several (N unknown) individuals' responses to a subjective questionnaire.

4) The eight data points corresponded to eight "suites of programs," but the source code for these programs is not available for further examination. Oman does not give the application domain(s) for these eight suites. It is unclear if code from fundamentally different domains (financial transaction processing vs. operating systems, for example) should have different MI values.

5) Four of the eight "suites of programs" were written in Pascal, and four in C. It is unclear how the choice of programming language will affect MI.

6) The sizes of the eight suites ranged from 1,011 to 5,533 LOC for the Pascal code, and from 1,810 to 9,625 lines for the C code. It is unclear how well Oman's MI model scales as the size of the code base grows.

7) Oman "validated" his MI metric against only six other suites (three in Pascal, ranging from 3,859 to 5,459 LOC, and three in C, ranging from 1,113 to 7,988 LOC) of unspecified application domains. There were simply too few data points to construct and validate a reliable metric.

8) Oman's validation was performed using a different (though still subjective) questionnaire from the one used to develop the MI polynomials.

9) MI is based on averages. In a program the size of the Linux kernel, there are thousands of individual functions, and a large number of nearly trivial

functions can mask the impact of the inherent difficulty in maintaining larger functions.

10) MI is not sensitive to coupling or to the relationships between functions. Common coupling (an extremely strong kind of relationship between functions or files) has been shown in a number of studies to have a serious impact on various aspects of maintainability.

These metrics appear to become increasingly unreliable when applied to consecutive versions of the Linux kernel and the volatility of successive series rises.  Before we address the issue of volatility, we first examine the pairwise correlations between our individual metrics in each of the three kernel series.

## 8.5 Pairwise Correlations between Metrics

Tables 8.3, 8.4, and 8.5 show the pairwise correlations between all six metrics in the 2.00.x, 2.02.x, and 2.04.x kernel series, respectively.

**Table 8.3 – Correlations between metrics in the 2.00.x series kernels (N=38)**

| | NCNB | Volume | Effort | MI | Common Coupling |
|---|---|---|---|---|---|
| MCC | $\rho_s$ = 0.9032<br>$t$ = 12.626<br>$P$ < 0.001 | $\rho_s$ = .9083<br>$t$ = 13.027<br>$P$ < 0.001 | $\rho_s$ = 0.9083<br>$t$ = 13.027<br>$P$ < 0.001 | $\rho_s$ = 0.7632<br>$t$ = 7.0865<br>$P$ < 0.001 | $\rho_s$ =0.9821<br>$t$ = 31.271<br>$P$ < 0.001 |
| NCNB | N/A | $\rho_s$ = 0.9871<br>$t$ = 37.039<br>$P$ < 0.001 | $\rho_s$ = 0.9894<br>$t$ = 40.959<br>$P$ < 0.001 | $\rho_s$ = 0.6758<br>$t$ = 5.5015<br>$P$ < 0.001 | $\rho_s$ =0.9011<br>$t$ = 12.467<br>$P$ < 0.001 |
| Volume | N/A | N/A | $\rho_s$ = 0.9792<br>$t$ = 28.962<br>$P$ < 0.001 | $\rho_s$ = 0.6651<br>$t$ = 5.3444<br>$P$ < 0.001 | $\rho_s$ = 0.8882<br>$t$ = 11.596<br>$P$ < 0.001 |
| Effort | N/A | N/A | N/A | $\rho_s$ = 0.6848<br>$t$ = 5.6388<br>$P$ < 0.001 | $\rho_s$ = 0.9025<br>$t$ = 12.570<br>$P$ < 0.001 |
| MI | N/A | N/A | N/A | N/A | $\rho_s$ = 0.7696<br>$T$ = 7.2319<br>$P$ < 0.001 |

Table 8.2 showed that, for the 2.00.x kernel series, all of the other five metrics are very highly significantly correlated with common coupling. (The data of Table 8.2 relating to versions 2.00.x is reproduced in the rightmost column of Table 8.3.) Table 8.3 indicates that there is also a very highly significant pairwise correlation between each of these five metrics. In particular, the magnitudes of the pairwise correlations between MI and each of the other four metrics are all very highly significant, but the values of the correlation are positive, rather than negative, indicating that MI is measuring something diametrically different to that of the other metrics, notwithstanding the fact that they are components of MI. In other words, MI indicates that maintainability is increasing when common coupling, MCC, NCNB, Volume, and Effort all indicate that maintainability is decreasing.

Table 8.4 shows the corresponding pairwise correlations of the metrics for the 2.02.x series of kernels.

**Table 8.4 – Correlations between metrics in the 2.02.x series kernels (N=20)**

| | MCC | NCNB | Volume | Effort | MI | Common Coupling |
|---|---|---|---|---|---|---|
| MCC | | ***$\rho_s = 0.8045$***<br>***$t = 5.7468$***<br>***$P < 0.001$*** | ***$\rho_s = 0.8962$***<br>***$t = 8.5724$***<br>***$P < 0.001$*** | ***$\rho_s = 0.8962$***<br>***$t = 8.5724$***<br>***$P < 0.001$*** | $\rho_s = -0.3624$<br>$t = 1.6497$<br>$P < 0.1$ | ***$\rho_s = 0.8641$***<br>***$t = 7.2849$***<br>***$P < 0.001$*** |
| NCNB | | | ***$\rho_s = 0.5763$***<br>***$t = 2.9916$***<br>***$P < 0.01$*** | ***$\rho_s = 0.8376$***<br>***$t = 6.5050$***<br>***$P < 0.001$*** | $\rho_s = -0.1805$<br>$t = 0.7784$<br>$P < 0.5$ | ***$\rho_s = 0.6173$***<br>***$t = 3.3287$***<br>***$P < 0.001$*** |
| Volume | | | | ***$\rho_s = 0.8932$***<br>***$t = 8.4291$***<br>***$P < 0.001$*** | $\rho_s = -0.3474$<br>$t = 1.5716$<br>$P < 0.2$ | ***$\rho_s = 0.8129$***<br>***$t = 5.9228$***<br>***$P < 0.001$*** |
| Effort | | | | | $\rho_s = -0.3564$<br>$t = 1.6183$<br>$P < 0.2$ | ***$\rho_s = 0.8393$***<br>***$t = 6.5495$***<br>***$P < 0.001$*** |
| MI | | | | | | $\rho_s = -0.2356$<br>$t = 1.0285$<br>$P < 0.4$ |

Table 8.4 indicates that all of the correlations involving MI are now negative, but they are too small to be statistically significant.

Table 8.5 shows the results of the same pairwise metric correlations for the 2.04.x series of kernels.

The magnitudes of the correlations for MI are once again very highly significant, but again the values of the correlations are positive, rather than negative. The drastic changes in the correlations involving MI between 2.00.x, 2.02.x, and 2.04.x are a further indication of the general unreliability of MI.

**Table 8.5 – Correlations between metrics in the 2.04.x series kernels (N=25)**

| | MCC | NCNB | Volume | Effort | MI | Common Coupling |
|---|---|---|---|---|---|---|
| MCC | | ***$\rho_s = 0.8708$***<br>***$t = 8.4933$***<br>***$P < 0.001$*** | ***$\rho_s = 0.9954$***<br>***$t = 49.473$***<br>***$P < 0.001$*** | ***$\rho_s = 0.9931$***<br>***$t = 40.544$***<br>***$P < 0.001$*** | ***$\rho_s = 0.9192$***<br>***$t = 11.197$***<br>***$P < 0.001$*** | $\rho_s = 0.2386$<br>$t = 1.1781$<br>$P < 0.3$ |
| NCNB | | | ***$\rho_s = 0.8422$***<br>***$t = 7.4909$***<br>***$P < 0.001$*** | ***$\rho_s = 0.8462$***<br>***$t = 7.6144$***<br>***$P < 0.001$*** | ***$\rho_s = 0.7031$***<br>***$t = 4.7416$***<br>***$P < 0.001$*** | ***$\rho_s = 0.5020$***<br>***$t = 2.7838$***<br>***$P < 0.01$*** |
| Volume | | | | ***$\rho_s = 0.9977$***<br>***$t = 70.470$***<br>***$P < 0.001$*** | ***$\rho_s = 0.9223$***<br>***$t = 11.445$***<br>***$P < 0.001$*** | $\rho_s = 0.2185$<br>$t = 1.0741$<br>$P < 0.3$ |
| Effort | | | | | ***$\rho_s = 0.9262$***<br>***$t = 11.777$***<br>***$P < 0.001$*** | $\rho_s = 0.2101$<br>$t = 1.0305$<br>$P < 0.4$ |
| MI | | | | | | $\rho_s = 0.1962$<br>$t = 0.9597$<br>$P < 0.4$ |

## 8.6 Volatility and Its Impact on Validity of Metrics

Now that we have shown MI to be unreliable within the context of the three stable series of the Linux kernel, we return to Table 8.2 to examine why fewer and fewer metrics remained statistically significant as we went from the 2.00.x to 2.02.x to 2.04.x series.

Because "correlation" is not synonymous with "causality," the presence of a correlation does not necessarily mean that we have identified the *source* of the correlation. We have shown that NCNB lines are very highly significantly correlated with common coupling (in Table 8.2). It is possible that there is some other as-yet unidentified effect that causes the correlation, and that the NCNB line count simply happens to be a manifestation of this other effect.

If such an effect exists, but we either cannot identify it or we cannot measure it, then in the case of considerable volatility, the volatility may overshadow the other metrics we *do* gather. The next question becomes one of measuring volatility.

In Section 7.10.3, we showed that the particular maintenance performed on the code in the 2.04.006, 2.04.008, and 2.04.014 versions caused the common coupling counts to decrease. In one case, this coincided with the deletion of code, and in the other two cases, the size of the code base did not decrease with the common coupling. The maintenance introduced volatility in the otherwise generally linear trend in the growth of common coupling.

Figure 8.11 shows the common coupling values for the three kernel series. Intuitively, we perceive increasing levels of variation (volatility) as we progress from the 2.00.x to the 2.02.x and then 2.04.x series, but how do we objectively quantify this variation?

On a set of data ranging from 90 to 110, a one-point variation would be much more significant than a one-point variation would be on another set of data that ranges from 90,000 to 110,000. Clearly, our measure of volatility must be sensitive to both the range of the variation and the mean of the data set. We therefore measure volatility in the common coupling as the standard deviation of the series divided by the mean. Alternatively, we could use the range divided by the mean. These values are shown for the 2.00.x., 2.02.x, and 2.04.x series in Table 8.6.

**Table 8.6 – Common Coupling Volatility Statistics by Series.**

| Series | Min | Mean | Max | Range | Std Dev | Std/Mean | Range/Mean |
|---|---|---|---|---|---|---|---|
| 2.00.x | 85,732 | 86,970 | 92,168 | 6,436 | 2,141.1 | 2.46% | 7.40% |
| 2.02.x | 415,812 | 441,252 | 503,508 | 87,696 | 28,862.5 | 6.54% | 19.87% |
| 2.04.x | 940,511 | 1,095,378 | 1,329,546 | 389,035 | 113,718.7 | 10.38% | 35.52% |

Table 8.5 shows that the mean coupling counts in the 2.02.x and 2.04.x kernel series are approximately 5 and 12 times as large (respectively) as those for the 2.00.x series, yet the standard deviation values are approximately 13.5 and 53 times as large, respectively. If we use the range, rather than the standard deviation, we get similar results – the 2.02.x and 2.04.x ranges are approximately 13.6 and 60.4 times their 2.00.x counterpart, respectively.

We conclude that this increase in volatility corresponds to the decreasing viability of the metrics in Table 8.2. Because NCNB lines is the only metric that has been validated with respect to common coupling in all three series, it is the only metric that remains highly statistically significant, even in the face of the increased volatility of the common coupling count.

## 8.7 Summary and Conclusions

In this chapter we have shown that Oman's Maintainability Index (MI) is generally unreliable as a predictor of maintainability. In the respective kernels when MI *does* happen to be highly significantly correlated with the other metrics, the MI values rose, suggesting increased maintainability (i.e., greater ease and reduced time and/or cost of maintenance), contrary to the common coupling metric, the "gold standard."

We have also shown that, as volatility increased, all the other metrics we examined except NCNB lines (namely, MCC, Halstead Volume, and Halstead Effort) became less reliable. The only metric to be validated against common coupling for all three kernel series was NCNB lines.

We had hoped that, even though these other metrics are of questionable validity in general, when applied to successive versions of the same product, they would indeed be valid. For all metrics except NCNB, this was not the case.

These other metrics do seem to be valid when the volatility is low, but under such circumstances there is little to measure. As volatility rose, more and more of the metrics failed to maintain statistically significant correlations with common coupling at the kernel level.

# CHAPTER IX

# CONTRIBUTIONS AND FUTURE WORK

## 9.1 Contributions

In Section 1.5, we listed the six contributions of this work. We begin this chapter by restating those contributions, commenting briefly on each in turn.

Contribution 1 (Chapter 7): The authors of [Schach et al., 2002] used the Linux kernel version number as their independent variable, and found that the common coupling in the Linux kernel was growing exponentially. When we re-evaluated Schach et al.'s data with respect to release date, rather than version number, we obtained a linear growth rate. Repeating this study with our standardized configuration, when applied to the preprocessed source code for three separate kernel series, we again found that the growth of common coupling in the Linux kernel is linear with respect to time.

Comments: Evolution of open-source software has historically been measured from version to version. Unlike commercial, closed-source software, open-source software is not typically associated with business cycles, and, as such has no regular release schedule. We have shown that the independent variable used in measuring growth should be release date, rather than version number. The most widely-cited paper in the field of maintainability and the Linux kernel [Schach et al., 2002] shows an exponential rate in the growth of common coupling with respect to version number, but when viewed with respect to release date, we have shown that the rate of growth was actually linear, not only by using the data from the previous study, but also with the data we obtained from a completely different method. This completely changes the generally-recognized pattern of growth in common coupling in the Linux kernel.

Contribution 2 (Chapter 5): Previous studies have concentrated on examination of the source code in only the `/linux/` subdirectory. That approach captures only a tiny portion of the source code base. We have examined the entire source code base and constructed complete, fully-configured kernels, actually capable of running on real hardware.

Comments: In the 2.04.020 kernel, the entire downloadable source contains 126.6 MB in 8,421 `.c` and `.h` header files. The `/kernel/` directory contains only 26 C files, totaling 0.35 MB. Clearly, those 26 files comprise a tiny minority of the entire kernel's code base, yet previous studies have concentrated on only those files, because nearly all of them are required in every kernel, and their code implements some of the core low-level functionality of the kernel. Even so, this selection excludes other required functionality, including file system, memory management, and networking support. We acknowledge that networking support is not strictly required in order to have a complete kernel, but virtually all modern systems are connected to some network.

The other end of the spectrum would be to consider the *entire* code base, under the assumption that the entire code base must be maintained, so any maintainability issues could surely impact the entire code base. Just as considering the `/linux/` subdirectory is an impractically narrow choice, considering the entire code base is an impractically wide choice. The entire code base for the 2.04.020 kernel contains support code for seventeen different architectures, and thousands of network, sound, video, SCSI, and PCMCIA devices. Clearly, no kernel would ever use all of the code in the entire code base, and could not possibly contain code to support more than one architecture.

We found a practical middle ground by configuring a "real-world" kernel containing at least one file system, Ethernet, SCSI card, sound card, and standard keyboard, mouse, and VGA support for the IA32 platform. This approach solves the problem of missing pertinent, mandatory subsystems without arbitrarily including code to support every obscure option possible.

Contribution 3 (Chapter 5): Because the Linux kernel is so highly configurable, the set of configuration options has a profound impact on the code base selected for compilation. We have developed a framework for selecting a consistent configuration across multiple versions of the Linux kernel, which makes longitudinal studies more readily comparable. This method can be extended to other operating systems, and to software product lines that are hardware-dependent.

Comments: In developing a workflow for building the Linux kernel for our study, we quickly determined that the specific configuration chosen at build time would have a profound effect on the code base to be examined. In order to address this issue, we developed a systematic way of specifying a consistent configuration that we could use to build sequential versions of the kernel, such that we still captured the natural evolutionary growth of the kernel, and included mainstream, real-world functionality, without resorting to making capricious arbitrary selections. The relatively few deviations from our configuration method were necessary, and, we believe, minor in their overall impact to the code base being chosen for compilation.

<u>Contribution 4 (Chapter 8)</u>: A number of “classical” metrics have been argued to be of questionable value as general predictors of software quality and maintainability. We performed a validation study using a longitudinal analysis of the Linux kernel to determine if metrics of questionable general validity might have increased merit when applied to sequential versions of a single product. We concluded that only LOC (lines of code) was useful as a

predictor of common coupling, a metric that has previously been validated with respect to various quality and maintainability attributes. McCabe's Cyclomatic Complexity, Halstead's Volume and Effort, and Oman's Maintainability Index all failed to consistently have a statistically significant correlation with common coupling, the "gold standard."

Comments: The oldest code metric, dating back to the first line of code ever written, is Lines of Code (LOC). McCabe's Cyclomatic Complexity and Halstead's Software Science metrics (Volume and Effort) date back to the late 1970's. Oman's Maintainability Index was first published in the early 1990's. The validity of all of these metrics has been the subject of considerable controversy for some time. Some studies have shown these metrics to be broadly applicable to any code, and other studies have attacked them on several fronts. We performed a longitudinal study of the Linux kernel to see if, within the confines of sequential versions of a single product, these metrics could be correlated with common coupling, because coupling in any of several forms has been validated in a number of studies, and common coupling is a particularly strong form of coupling. We were able to obtain statistically significant correlations between only LOC and common coupling across all three series we examined. When the volatility of the code base was low, all of these metrics had high correlations with common coupling, but as volatility in the code base grew, the only metric robust enough to maintain its correlation with common coupling was LOC. We therefore join the camps of the detractors of these metrics. If we were unable to obtain correlations with common coupling in successive versions of a single product, we do not see how they could possibly be broadly applicable to large code bases such as the Linux kernel.

Contribution 5 (Section 8.2): Although we used many GPL open-source programs and one commercial closed-source programs in our analysis, we also created a suite of programs to act as "glue" between the other tools, and to provide a GUI-based interface to the database to facilitate selection, graphing, and computation of statistics on all metrics in the database.

Comments: We used readily available GNU tools (`gcc`, `make`, `vim`, `ld`, `rpl`, etc.), and other tools available under the GPL (including Debian Linux, KDE, MySQL, and `cxref`) for much of our research. We also used the commercial closed-source CASE tool Klocwork K7 Development Suite, provided courtesy of Klocwork, Inc. This set of tools was not sufficient, however, to provide an end-to-end solution for the collection, management, and extraction of the metrics, and resulting information. We then developed a suite of CASE tools to bridge the gaps between these various packages. In some cases, the tools were needed to parse the output of one existing tool and convert it into a format the next tool in the chain could use as input. In the case of `Grapher`, we developed a flexible, point-and-click interface to the database that greatly facilitated extraction of the metrics, their trends, and the corresponding statistics.

Contribution 6 (Appendix A): Because many of the older kernels had code that depends on compiler features that have since been deprecated, the older kernels cannot be built using newer versions of `gcc`. We have identified a set of tools that allows the kernels to be built and the changes that must be made to the source code to achieve this.

Comments: The information in the "`COMPILING the kernel`" section of the `README` file distributed with the older Linux kernels is not correct with respect to the tools required to actually build the kernel, largely because the older kernels used coding standards and shortcuts for which support in `gcc` has since been deprecated. Because of this, the newer tools will not compile the older kernels, despite the fact that the `README` files identify a specific `gcc` version and then append "`or newer`." We have determined a readily-available set of tools (packages for the Debian Linux distribution) that can be used to successfully compile all of the 2.00.x, 2.02.x, and 2.04.x series kernels. We have also identified instances in the source code itself that exercised deprecated features of these tools, and carefully documented workarounds that allow the code to compile. This detailed information, unavailable elsewhere, will serve as an aid to future researchers.

## 9.2 Tools We Developed Specifically for this Dissertation

In Sections 5.6, A.4, and 7.10.3, we mentioned in passing many of the tools we wrote as part of developing the workflow for the acquisition and analysis of the code metrics. In this section, we explicitly list and briefly describe all the tools we created specifically for this dissertation.

### 9.2.1 `ConfigurationChecker` (Source File: 1,050 lines total)

`ConfigurationChecker` reads the captured screen output from all versions of the `make config` step of the kernel compilation process, and builds a table listing the various configuration items, their default values, and the values we selected during the manual configuration step. For any configuration options for which our manual response differs from the default response, the corresponding cell in the table is highlighted in yellow, indicating an entry to double-check. As discussed in great detail in Chapter 5, there were some instances where the default changed from version to version, and we might have (correctly) responded "Y" in some versions to items with an "N" default, because in later versions, the default *was* "Y." Table entries corresponding to those instances where our responses were inconsistent from version to version are highlighted in red. Because our goal was to develop a configuration method that would assure us a consistent configuration, such instances were likely to be errors to be corrected. There were two such instances in which this was not actually indicative of an error, as discussed in Chapter 5. Finally, `ConfigurationChecker` also flags configuration option items for which the default response changed from version to version. This is not indicative of a problem; this is an informational warning. The use of `ConfigurationChecker` is discussed in Section 5.6.

### 9.2.2 `NIQAD` (Source File: 929 lines total)

Just as `ConfigurationChecker` reads the captured screen output of the `make config` step of the build process, `NIQAD` reads the captured screen output of the `make` step of the build process, except that `NIQAD` operates on only a single version of the kernel at a time. `NIQAD` reads the commands that `make` issued, and extracts those commands that were calls to `gcc`. Those commands are then processed, stripping options related to code generation and compilation, and converting the remainder of the command into one that will only *preprocess* the same file the original `gcc` invocation sought to *compile*. The entire set of commands is output to the shell script `make.sh` so that the user can then invoke the script file to preprocess exactly the same set of files that `make` just compiled. NIQAD is discussed in Section 7.4.6.

### 9.2.3 `DirectoryProcessor` (Source File: 405 lines total)

After the user has preprocessed the same files that `make` compiled, we need to delete everything except the remaining files containing the preprocessed code. `DirectoryProcessor` does precisely that. In addition, it builds the script that runs `cxref`. The command line for `cxref` requires us to explicitly list every source file over which to build the cross reference. While `DirectoryProcessor` is traversing the directory tree, is builds the `cxref` script from the list of files it keeps. Finally, DirectoryProcessor removes any directories that are empty. DirectoryProcessor is discussed in Section 7.4.6.

### 9.2.4 KWReportParser-DB_Output (Source File: 1,677 lines total)

After Klocwork runs over the preprocessed source tree, KWReportParser-DB_Output processes the report.txt file generated by Klocwork, converting it to a format that can be read by another program for uploading into the database. This program parses the report.txt file, extracting the relevant process-level and file-level metrics, building a consolidated sparse matrix, and then outputting the consolidated values in a text file. KWReportParser-DB_Output is discussed in Section 7.4.11

### 9.2.5 DatabaseLoader (Source File: 822 lines total)

DatabaseLoader takes the output from KWReportParser-DB_Output and actually populates the MySQL database with the resulting information. In order to save space, file and function names exist in their own tables. For file-level records, DatabaseLoader first checks to see that the file name already exists in the FileNames table, and, if not, it adds it to the list. Next, it checks to see if the file-level metric record already exists. If so, it updates the record, and if not, it creates it. The same approach applies to the process-level metrics. DatabaseLoader is discussed in Section 7.4.15.

### 9.2.6 CXREFProcessor (Source File: 1,666 lines total)

After cxref has been run on the preprocessed source code, the output is a file called CXREF.results. This file is processed by CXREFProcessor. For every preprocessed source file, it determines which GVs are both declared in and actually used within a function in the source file. The result is a file called CXREF.dat, which contains (1) a list of the files in the kernel build, (2) a list of the GVs that appear somewhere in the kernel build, (3) and a matrix. The rows of the matrix correspond to the files in the kernel, and the columns correspond to the GVs. At the intersection of each row and column is a "1" if that GV is visible (declared with extern, or defined) in that file, and a "2" if that GV is actually used within a function in that file. CXREFProcessor can also be used as an analysis tool. The matrix is not only saved to CXREF.dat; it is also displayed. Also, the user may click on a file name to see the GVs both visible and used in the file, and the user may click on a GV to see in which files that GV is both visible and used. CXREFProcessor is discussed in Section 7.4.12.

### 9.2.7 DatabaseLoader-FileSize (Source File: 335 lines total)

The source file size (in bytes) is not reported by Klocwork, although it is obviously available from a directory listing. If the user creates a directory listing of the preprocessed source tree using the command DIR /s *.c > DirectoryList.txt from the root of the source tree, then DatabaseLoader-FileSize will scan through the directory listing, locating each file's record in the file-level metrics table, and update the field containing its file size in bytes (F0047).

DatabaseLoader-FileSize is not explicitly discussed in any other section; it was created to do nothing other than to populate F0047 with the file sizes.

### 9.2.8 CXREF_CodeSweeper (Source Files: 4,547 lines total)

CXREF_CodeSweeper reads the CXREF.dat files, along with the preprocessed source code for a given version of the kernel, and attempts to automatically parse the code, seeking all instances of global variables. The instances it cannot automatically parse are presented to the user for classification. The output of this program is a list of manually-parsed lines and a file called GV_Data.txt, containing a list of every instance of a GV that occurs in a function within each file. CXREF_CodeSweeper is discussed in Section 6.3.

### 9.2.9 GV_Coupling_Processor (Source File: 781 lines total)

The output of CXREF_CodeSweeper is a file containing a list of every instance of a GV occurring executable code in all functions in all files of a given version of the kernel. GV_Coupling_Processor reads that list, coalesces the number of instances by kind, and optionally updates the database, populating fields K0000 through K0011.

### 9.2.10 Grapher (Source File: 7,981 lines total)

Grapher is discussed in detail in Section 7.4.16. Grapher has two main functions:

1) SQL query builder. The user can use the point-and-click interface to specify each of the components of a SQL query:

a. The desired type of graph (histogram, correlation, trend by kernel version number, or trend by kernel release date);
b. The kernel version(s) for which to select data; and
c. The metric(s) to analyze.

The user then clicks on "Build Query," and Grapher constructs a SQL SELECT statement containing the user-selected field list, as well as the appropriate FROM, WHERE, ORDER BY, and GROUP BY clauses for the selected graph parameters. The program then uses the type of graph and the selected parameters to set the graph title, subtitle, and the X-and Y-axis labels.

If the user requires a more complicated query than the query builder can handle automatically, the user has the option of manually entering a SELECT statement in a provided text box, or using the automatically-generated SELECT statement as a starting point, and then editing it manually.

After the user clicks "Execute Query," the SQL statement is passed to MySQL for processing. The resulting recordset is then returned to Grapher.

2) Graphics engine. The record set resulting from the execution of the user's query forms the input for the graph to draw. Grapher automatically computes the appropriate ranges for the X and Y axes (or the user can specify them manually), draws and labels the plot area, and plots the data. In the case of correlation plots,

> Grapher also draws a least-squares regression line through the data, computes and displays the Spearman rank correlation coefficient ($\rho_s$), as well as the corresponding t and P values. After the graph is displayed, it is automatically placed on the clipboard for pasting into other applications.

The user may also copy to the clipboard or persist to a disk file the data used to draw the graph for further processing in some other application. _As stated at the beginning of this section, Grapher is discussed in Section 7.4.16.

### 9.2.11 CXREF_Comparator (Source File: 708 lines total)

CXREF_Comparator is used to show the source of differences in the coupling counts between two (presumably, but not necessarily, sequential) versions of the Linux kernel and graphically displays the differences between the two files in terms of coupling. This involves comparing the lists of file names and global variable names, as well as which GVs are either visible or used in the two versions being examined. CXREF_Comparator is discussed in Section 7.10.3.

## 9.3 Threats to Validity

As in [Dinh-Trong and Bieman, 2005], we examine threats to validity in four broad areas – construct validity, content validity, internal validity, and external validity. We consider each of these in turn.

1) *Construct validity* refers to whether the metrics actually quantify what we want them to. We used common coupling as our "gold standard." We examined coupling in general, and common coupling (a particularly strong form of coupling) specifically, in Sections 1.3 and 2.3.1. The sources cited in these sections clearly support the use of common coupling as our reference metric. We therefore find no construct validity issues.

2) *Content validity* refers to the "sampling adequacy of the content… of a measuring instrument." On the one hand, by examining every line of every source file, we examined 100 percent of the population (where "the population" means "the entire body of C source code for a given version of the kernel"); from that viewpoint, there were no content validity issues. On the other hand, our configuration method required us to select an architecture, and we chose the popular i386 architecture. Although it seems unlikely, it is possible that the i386 architecture has unique features that make this a suboptimal choice (i.e., the choice of i386 as our architecture may have caused the preprocessor to `#include` code that is radically different in nature from the rest of the kernel, or from that of other architectures). As the most pervasive hardware platform in use, it seemed a fitting choice, and is representative of the majority of systems running Linux. Nevertheless, this is a possible content validity issue.

3) *Internal validity* refers to "cause-and-effect relationships." The goal of a validation study such as we performed is not to determine cause and effect, but to perform

correlations between metrics. We know that, even in the presence of a statistically significant correlation, we cannot make the leap to "cause and effect." Rather, some other variable(s) may be the cause, and we are merely observing and measuring the effect. Because we are not attempting to identify any cause-and-effect relationships, we find no internal validity issues.

4) *External validity* refers to "how well the study results can be generalized beyond the study data." The short answer here is that we simply do not know. Our goal was to intentionally reduce the number of variables in play by restricting our analysis to time-sequential versions of the same program, in an attempt to find correlations between various popular metrics and common coupling. Because we were unable to find such correlations, we conclude that these other metrics are invalid predictors of maintainability in the Linux kernels we examined. However, we are unable to extrapolate these findings to other series of the kernel (such as 2.6.x), or other products, without performing such a study on these other programs. We therefore do find external validity issues.

Beyond these four broad categories, we also find the following three specific threats to the validity of our research:

1. By preprocessing the source code, we were able to conclusively resolve any conditional compilation issues, and expand all macros. Therefore, we analyzed precisely the same code that would have been passed to the parser to be converted

into executable code. In Section 8.1.3.1, we discussed the issue of `extern inline` functions, in which code that appears to be a “regular” function behaves more as a macro or as a “code template.” It is possible that these functions were part of the preprocessed code only as an artifact of the `#include` tree, and not because they were actually used in a given file. In other words, it is possible that our assumption that “all code that survives preprocessing ultimately becomes executable code” is not strictly correct. However, even if our assumption is not true, this does not change any of our results or our conclusions, because maintenance is performed on source code, not executable code.

2. Although we conducted thorough tests on code samples to verify that Klocwork did, in fact, compute the various metrics in the same way that we interpreted the metrics’ definition, it is possible that there were instances in which Klockwork’s interpretation on some aspect of the code differed from ours. We identified one such a difference with respect to the “globalness” of a GV (see Section 6.1). It is possible there are other such issues of which we were not aware, despite our careful testing to mitigate this threat.

3. Some of the metrics we examined were defined in the late 1970’s, when smaller programs using different languages were the norm. These metrics were defined for “a module,” but it is not clear precisely what constitutes a module within the context of modern programming languages. It could be argued that either a function or a file

could be considered a module. Even the IEEE glossary [IEEE, 1990] leaves the precise definition of a module open:

1) A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, our output from, and assembler, compiler, linkage editor, or executable routine.
2) A logically separable part of a program. Note: The terms "module," "component," and "unit" are often used interchangeably or defined to be sub-elements of one another in different ways depending upon the context. The relationship of these terms is not yet standardized.

This explicit ambiguity implicitly makes the aggregation of "module-level" metrics into higher-level values ambiguous as well. It is possible that these metrics are inherently inapplicable in the form for which we used them. If that is indeed true, then, in the new millennium, no such "old" metric is valid in the context of the large-scale programs such as Linux, Apache, MySQL, FireFox, among others, and new metrics will have to be sought to better relate to aspects of quality factors in such programs.

## 9.4 Future Work

As with any significant endeavor, we have identified a number of items that we believe are worthy of follow-up study in the future:

1. In this dissertation, we examined the stable Linux kernel series 2.00.x, 2.02.x, and 2.04.x. Because of the merging of the development and stable releases in the 2.06.x series, we did not incorporate the 2.06.x kernels, because that would have commingled dissimilar data, and thereby would have made it impossible for us to draw reliable conclusions. However, extending this work to the 2.06.x kernels could be of interest as a way of measuring the impact of combining the two types of releases.

2. In [Schach et al. 2002], the authors did not distinguish between development and stable releases. We would extend their work into the 2.05.x and 2.06.x kernels to see if the trends from their previous study had continued.

3. We analyzed common coupling at only the kernel level (inter-file common coupling). It is possible that aggregating our metrics to the kernel level (by summing the LOC, Volume and Effort metrics; and averaging the MI values) may have obscured trends that are apparent at only lower levels of granularity (file-level or function-level). Accordingly, a study of Linux at the file- and function-level should be performed. We remark, however, that there are hundreds of files and tens of thousands of functions in *each* kernel. Accordingly, the work involved in such a lower-level study is likely to be vast.

4. It would be interesting to examine some less popular and less well-known metrics than those considered in Chapter 8, and validate them against common coupling.

5. We analyzed as our code base the preprocessed C source code for the respective Linux kernel versions. Given the issues with `extern inline` functions (Section 8.1.3.1), it might be reasonable to extend the preprocessing of our source code one step further, and apply the `-S` `gcc` compiler option to generate assembly source code from our C files. We note, however, that no previous research into metrics at the assembler level has apparently been undertaken, so this proposed analysis, though extremely promising, would require considerable theoretical research before empirical research could commence.

6. Our categorization of all variable instances into eight categories provides a fine-grained dissection of the actual occurrence of variable usage. One related area in which further research should be done is the passing of global variables as parameters to functions. In our research, we did not differentiate between passing a variable by value and passing a variable by reference. When passing a variable by reference, the function being called can only read the value of the variable (unless, of course, the variable is a pointer). Only when a variable is passed by reference (or when we explicitly pass a pointer) can the called function modify global data. In the future, we should like to perform an analysis comparing the call-by-value and call-by-reference instances of passing a global variable as a parameter.

# APPENDIX A

# EXTRACTING LINUX KERNEL SOURCE CODE METRICS

## Contributions in This Appendix

5. Although we used many GPL open-source programs and one commercial closed-source programs in our analysis, we also created a suite of programs to act as "glue" between the other tools, and to provide a GUI-based interface to the database to facilitate selection, graphing, and computation of statistics on all metrics in the database (see Appendix A).

6. Because many of the older kernels had code that depends on compiler features that have since been deprecated, the older kernels cannot be built using newer versions of `gcc`. We have identified a set of tools that allows the kernels to be built and the changes that must be made to the source code to achieve this (see Appendix A).

## A.1 Introduction

This appendix details the process of building the Linux kernels and extracting the source code metrics. Section A.2 discusses the specific steps taken in the processing, and Section A.5 examines the lessons learned from earlier unsuccessful attempts at developing what ultimately became the workflow in Section A.4. Section A.3 describes the operating environment provided by VMWare, which allows us to run Linux in a VM (Virtual Machine) under Windows.

## A.2 The Workflow

The original high-level plan for this dissertation was quite simple. For each version of the kernel:

1. Build (compile and link) it and capture the `gcc` invocations used to compile the individual files.
2. For each `gcc` invocation, change the `gcc` options to merely preprocess the source code, rather than producing object code.
3. Run `gcc` again on all the source code, this time producing preprocessed code.
4. Run Klocwork on the preprocessed code to harvest the code metrics.

Unfortunately, as with most non-trivial projects, the four-step process above turned out to be merely a *very* high level framework that still had numerous holes to fill, or involved some manual steps that clearly would have become most tedious if not automated.

A slightly more detailed plan would be:

1. Modify the source code and/or `Makefile` (if necessary) to make the kernel buildable.
2. Run `make mrproper` to provide a pristine build environment.
3. Run `make config` to specify the desired configuration.

4. Verify that the configuration is correct (specifying the configuration requires answering a large number of configuration questions, and unintended responses are quite possible).
5. Run `make dep` to resolve the file dependencies.
6. Run `make` to actually build the kernel.
7. Capture the output of Step 5, and remove from each invocation of `gcc` the compiler switches related to code generation, and replace them with options to preprocess the code and then stop. The result of this step is a shell script.
8. Run the shell script created in Step 6 to create preprocessed code.
9. Remove all files from the source code tree except the preprocessed code.
10. Run Klocwork and `CXREF` on the resulting (preprocessed) source tree.

### A.3 The Operating Environment (Debian Linux under VMWare)

Although the Klocwork CASE tool and MySQL (which ultimately houses the metrics) both run under Windows for their respective parts of this project, building the kernels requires doing so under Linux. The Debian Linux distribution was ultimately selected, because Debian maintains packages (ready-to-install binary subsystems) for the old compiler tools (including `gcc 2.72`, `gcc 2.95,` and `CXREF`) required to build and analyze the older Linux kernels.

As the early stages of this project progressed, it became clear that there would be a large number of times that something would need to be done under Linux, the result of which would be processed under Windows, and then sent back to Linux for further processing.

It would have been most cumbersome to create a dual-boot environment and "cross the Windows-Linux wall" by rebooting at every step. Moreover, there would be times where more than one process could potentially be running at once (such as building one kernel while preparing the next).

All of these concerns were addressed by installing VMWare Workstation (`www.VMWare.com`). VMWare allows us to create and run *simultaneously* up to four virtual machines (VMs) under Windows XP Pro. Three VMs were created, each with its own private virtual disk – one to build the 2.0.x kernels, one for 2.2.x, and one for 2.4.x. The private virtual disks for the 2.0.x and 2.2.x VMs were 8 GB in size, and the private virtual disk for the 2.4.x VM was 20 GB. Each VM was loaded with Debian Etch, and the appropriate utilities – `KDE 3.5`, `vim`, and the `GNU` tools required to build the respective kernels. The `synaptic` package manager was used to handle the download and installation of the packages.

Beyond the obvious ability to literally run Linux under Windows, one of the features of VMWare that makes it so desirable in this situation is that it supports what it calls *shared folders*. Because VMWare not only provides and manages the resources of the virtual machine (the "guest"), it also has access to resources on the "host" machine, and, as such, VMWare can handle the mapping of resources between the two environments. Using shared folders, the files located on the Linux VM in the directory `/dev/mnt/hgfs/Share1/<kernel version>` under Linux also appear under Windows in the subdirectory `C:\SharedFolders\<kernel version>`. This makes transferring data across “the O/S wall” trivial – we simply move the necessary files into `/dev/mnt/hgfs/Share1/<kernel version>` and then ALT+TAB to Windows, and the data is immediately available under Windows, while the Linux VM continues to run simultaneously. The transfer works in the other direction as well.

On each VM, we used the same directory structure. We created a subdirectory in the format `2.xx.yyy` (zero padded) for each version of the kernel within `C:\SharedFolders` under Windows; under Linux, our user ID was “`lgt,`” so we created the same directories under `/home/lgt`. This facilitated moving files between environments, particularly when moving files associated with more than one kernel at a time.

## A.4 The Workflow In Detail

This section presents a fully-detailed walkthrough of the workflow.

### A.4.1 Modify the Source Code and/or `Makefile`

There are some instances in which the source code (and/or `Makefile`) has to be modified slightly in order to get the kernels to compile at all. There are three primary reasons why such modifications are needed:

1. There was a fault in the original code.
2. The code (often assembler code) attempted to exercise some aspect of `gcc` that had been changed or deprecated. The 2.0.0 kernel’s `README` file calls for “`gcc`

2.6.3 or newer," but "or newer" does not always work. According to [gcc.gnu.org/releases.html], gcc 2.6.3 was released November 30, 1994, but the 2.7.2.3 version available from Debian was released August 22, 1997. The three years between the two compilers were times of significant change within gcc, and some features and coding practices acceptable at that time were later deprecated. Put simply, the new tools will not compile the old kernels.

3. A change was needed to accommodate the available tools (such as changing "gcc" to "gcc272" in the Makefile).

Some of the changes required to be able to compile the kernel resulted in no change to the code metrics, while others required deleting a line or changing code slightly. When the alternative was to not be able to compile the kernel at all, the introduction of the slight noise in the data caused by single lines of code was deemed an acceptable compromise.

Some of the changes require using vi (or some other editor) to make the necessary changes; in other instances rpl works. Because it can be invoked from the command line without any other user intervention, using rpl allows us to script some of the kernel-building process. The format for the rpl command is straightforward (for more details see the man page for rpl):

```
rpl[switches]<old string> <new string> <file>
```

Table A.1 gives the detailed, version-by-version changes required to be able to build the kernels:

**Table A.1 – Version-by-version Changes Required to be Able to Build the Kernels.**

| # | Version(s) | Change(s) |
|---|---|---|
| 1 | 2.00.000 through 2.00.040 | Enable SMP support:<br>for Z in 00 01 02 03 04 05 06 07 08 09 10 11 12 \<br>     13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 \<br>     28 29 30 31 32 33 34 35 36 37 38 39 40; \<br>rpl –b ‘# SMP = 1’ ‘SMP = 1’ ~/2.00.0$Z/linux/Makefile; \<br>done<br>One replacement should be made for every Makefile. |
| 2 | 2.00.000 through 2.00.040 | Change gcc to gcc272 in the Makefile:<br>for Z in 00 01 02 03 04 05 06 07 08 09 10 11 12 \<br>     13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 \<br>     28 29 30 31 32 33 34 35 36 37 38 39 40; \<br>  rpl –b ‘gcc’ ‘gcc272’ ~/2.00.0$Z/linux/Makefile; \<br>done<br>There are two places where this needs to be fixed in each Makefile |
| 3 | 2.00.000 through 2.00.033 | Edit arch/i386/kernel/entry.S. Search for ENTER_KERNEL. Scroll down about 10 lines. Delete the *two* lines following label “2:” that contain “btl” or (3 lines later) “btrl” instructions and “SYMBOL_NAME(smp_invalidate_needed).” The should say:<br>   btl %al, SYMBOL_NAME(smp_invalidate_needed); \<br>     and<br>   btrl %al, SYMBOL_NAME(smp_invalidate_needed); \<br>This can also be automated for all of the affected 2.0.x versions:<br>for Z in 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 \<br>     17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33; do\<br>  rpl -b ‘btl %al, SYMBOL_NAME(smp_invalidate_needed);’ ‘ ’ \<br>     ~/2.00.0$Z/linux/arch/i386/kernel/entry.S;   done<br><br>for Z in 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 \<br>     17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33; do \<br>  rpl -b ‘btrl %al, SYMBOL_NAME(smp_invalidate_needed);’ ‘ ’ \<br>     ~/2.00.0$Z/linux/arch/i386/kernel/entry.S; done<br>Each of the replacements should change only one line of code. |

**Table A.1, Continued**

| | | |
|---|---|---|
| 4 | 2.00.000 through 2.00.033 | Repeat the previous step for the file `include/asm-i386/irq.h`. The formatting is a bit different, but look first for “`ENTER_KERNEL`,” and the line containing “`2:`”. Shortly thereafter, delete the next two lines that contain “`btl`” or “`btrl`” and the string (including quotes) “`SYMBOL_NAME_STR(smp_invalidate_needed)`”<br>This can also be automated for all of the affected 2.0.x versions:<br>`for Z in 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18    \`<br>`        19 20 21 22 23 24 25 26 27 28 29 30 31 32 33; do echo $Z;   \`<br>` rpl -b ‘btl %al, “SYMBOL_NAME_STR(smp_invalidate_needed)”\n\t’ ‘ ‘  \`<br>`        ~/2.00.0$Z/linux/include/asm-i386/irq.h; done`<br>`for Z in 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18    \`<br>`        19 20 21 22 23 24 25 26 27 28 29 30 31 32 33; do echo $Z;   \`<br>` rpl -b ‘btrl %al, “SYMBOL_NAME_STR(smp_invalidate_needed)”\n\t’ ‘ ‘ \`<br>`        ~/2.00.0$Z/linux/include/asm-i386/irq.h; done`<br>Each of the replacements should change only one line of code. |
| 5 | 2.00.004 | In `drivers/sound/sb_common.c`, in `probe_sbmpu`, a semicolon is missing. About halfway through this function, in this block:<br>`hw_config->name = “Sound Blaster 16”;`<br>`hw_config->irq = -devc->irq;`<br>`sb16_set_mpu_port(devc, hw_config)   <--** missing ;`<br>`break;`<br>Append a semicolon to the line that begins with `sb16`. |
| 6 | 2.00.005 | In `net/ipv4/ip_sockglue.c`, the code will not compile as-is. A fix was included in 2.00.006 that solved this problem, so we will force the four lines here. This does not change the metrics. Search and replace (two places each):<br>`dev=rt->u.dst.dev;           -->  dev=rt->rt_dev;`<br>`atomic_dec(&rt->u.dst.use);  -->  atomic_dec(&rt->rt_use);` |
| 7 | 2.00.009 | In `drivers/block/ide.c`, in `try_to_identify`, change the first few lines from this:<br>`   int irqs, rc;           <-- change (add hd_status)`<br>`   unsigned long timeout;`<br>`#ifdef CONFIG_BLK_DEV_CMD640`<br>`   int retry=0;`<br>`   int hd_status;          <-- delete this line`<br>to this:<br>`   int hd_status, rc, irqs=0;`<br>`   unsigned long timeout;`<br>`#ifdef CONFIG_BLK_DEV_CMD640`<br>`         int retry=0;` |

**Table A.1, Continued**

| | | |
|---|---|---|
| 8 | 2.00.034 through 2.00.040 | The 2.0.34 kernel will not compile properly as-is using our chosen configuration. During the “`make dep`” step, it stops with “`[fastep] error 135`” while in the file system (`/fs` ) directory. It is unclear what is causing this problem, but based on a web posting related to similar problems another user was having relative to compiling the 2.0.38 kernel, we tried the build using the 2.0.40 version of `scripts/mkdep.c`, and the build was successful. The program `mkdep.c` (used during the `make` process, but not part of the kernel itself) controls how the dependency information is generated for `make`, and should not affect the contents of the generated code (or the subsequent metrics).<br>In 2.00.035 through 2.00.040, the same error occurs, but it occurs in the `/drivers/char` directory, rather than in `/fs`. The same fix solves the problem, however.<br>Also, beginning with 2.0.34 (and running through 2.0.40), it is necessary to include Native Language Support (NLS) in order to be *able* to include the ISO 9660 file system. Including NLS requires the choice of a code page and an ISO Character set. We arbitrarily chose Code Page 437 (US) and ISO Character Set 8859-1 (Western Europe, US). |
| 9 | 2.00.040 | When we decompress the archive as downloaded from `www.LinuxHQ.com`, the “`/linux/`” directory will become the `linux-2.0.40` directory. In order to keep our version numbering and file naming conventions consistent, just rename it with:<br>`mv ~/2.00.040/linux-2.0.40 ~/2.00.040.linux` |
| 10 | 2.02.000 through 2.02.026 and 2.04.000 through 2.04.031 | There are problems with `binutils`, `gcc`, or the newer processors and the old code. In any event, attempting to compile `process.c` produces a number of warnings, followed by a number of errors related to `mov` / `movl` assembly language instructions. This may be a heavy-handed way to fix the problem, but it DOES allow the code to compile:<br>1) Run the `make mrproper`, `make config`, and `make dep` steps<br>2) Edit `include/asm/system.h`. In the fourth line of the definition of the macro `loadsegment`, change the `movl` instruction to a `mov` instruction (drop the “L”). Just use `vim.`<br>3) Edit `arch/i386/kernel/process.c`, converting all instances of `movl` to `mov`:<br>`rpl -b ‘movl’ ‘mov’ arch/i386/kernel/process.c`<br>4) Run the `make` step, and the kernel should build without incident. |

**Table A.1, Continued**

| | | |
|---|---|---|
| 11 | 2.02.000 through 2.02.026 | The 2.2.x series of kernels all require “`gcc` 2.7.2 or newer,” but there is code in the `Makefile` of versions 2.02.018 – 2.02.026 to check for Debian `gcc-2.72`, and use “`gcc272`” to invoke `gcc`. There *is* still one place to be changed, however, so use the following commands:<br>`for Z in 00 01 02 03 04 05 06 07 08 09  \`<br>`         10 11 12 13 14 15 16 17;do     \`<br>`   rpl -b ‘gcc’ ‘gcc272’   ~/2.00.0$Z/linux/Makefile; done`<br>`for Z in 18 19 20 21 22 23 24 25 26; do \`<br>`   rpl -b ‘=gcc’ ‘=gcc272’ ~/2.00.0$Z/linux/Makefile; done` |
| 12 | 2.02.000 | The file `drivers/sound/sb_card.c` must be modified. The definition of the integer variable `esstype` resides within an `#ifdef` block that depends on the file’s being compiled as a module. We are running *without* loadable module support, so this declaration is not visible when other files that need that variable are compiled. So, edit `drivers/sound/sb_card.c`, and move the declaration of `esstype` up about 15-20 lines to the line immediately preceding “`#ifdef MODULE`.” This was apparently fixed in 2.02.001 (see http://www.linuxhq.com/kernel/v2.2/1/drivers/sound/sb_card.c for more information regarding this file). |
| 13 | 2.02.017 | Change the definition of the `CC` line in the `Makefile` to “`gcc272`,” rather than “`cc`” |
| 14 | 2.04.000 through 2.04.035 | The `README` file for 2.04.000 says that `gcc 2.7.2.3` was no longer supported, and that we need `gcc 2.91.66`, and that “`2.95.2` may also work but is not as safe.” The version of `gcc 2.95` that we can load as a Debian package is `2.95.4`, so we use that. By the 2.04.020 version of the kernel, those two were reversed in the `README` (“have `2.95.3` available. `gcc 2.91.66` may also work but is not as safe”). In 2.04.034, it still says to use `gcc 2.95.3`, and that `gcc 4` was not yet supported. So, change all of the `Makefiles` from `“gcc”` to `“gcc-2.95.”` See #11 (for 2.02.xxx) above, and use the same approach. |
| 15 | 2.04.019 through 2.04.035 | The extraction of the `linux` directory in versions 2.04.000 - 2.04.018 is straightforward -- the `.tar.bz2` file contains the `linux/` directory. Starting with version 2.04.019, however, the directory is named `linux-2.4.x/`, where `x` is two digits. Hence, once the `.tar.bz` files are all extracted, we must rename these directories to preserve our naming convention:<br>`for Z in 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35;\`<br>`    do mv ~/2.04.0$Z/linux-2.4.$Z ~2.04.0$Z/linux;          \`<br>`done` |

**Table A.1, Continued**

| | | |
|---|---|---|
| 16 | 2.2.000 through 2.02.026 and 2.04.000 through 2.04.020 | In the file `drivers/ide/ide-cd.h`, there is a declaration of type "`__u8 short`." The problem is that "`short`" is 16 bits, which overrides the 8-bit declaration of "`__u8`." The `gcc 2.95` compiler did not have a problem with this code, and compiled it as an unsigned 16-bit value. When we run `CXREF` on this file, however, it will not be USING the `cpp 2.95` preprocessor, and it will throw a syntax error on this file. This was fixed in 2.04.021, by changing "`__u8 short`" to "`__u16`" (see `http://www.linuxhq.com/kernel/v2.4/21/drivers/ide/ide-cd.h`):<br>`for Z in 00 01 02 03 04 05 06 07 08 09 10  \`<br>`            11 12 13 14 15 16 17 18 19 20; \`<br>`    rpl -b '__u8 short' '__u16'             \`<br>`    ~/2.04.0$Z/linux/drivers/ide/ide-cd.h; \`<br>`done`<br>If this is not fixed at compile time, then it is not a problem; it can be fixed before running `CXREF`; just change the name of the file to fix from `ide-cd.h` to `ide-cd_C.c` in the command above. Also, if we are changing the "`_C.c`" file, we *must not* use the `-b` option on `rpl` – we do not want an extra file for CXREF to be tempted to process! |

## A.4.2 `make mrproper`

This step is required to "clean up" from any previous fully- or partially-completed build of the kernel. It ensures that `make` has a pristine environment from which to proceed.

The output of "`make mrproper`," which is run in a terminal window under KDE, can be sent simultaneously to the screen (so we can see the output) and to a text file for later processing, by piping the output to `tee`, with "`make proper | tee ../make-mrproper.scr`" (there is no particular significance to the extension "`scr`" other than it stands for "screen scrape"). Note that we execute "`make mrproper`" from within the `/linux/` directory in the source tree, but we want the resulting `make-mrproper.scr` file to reside in the root of the source tree (hence the "`../`" following `tee`).

The alternative to using tee is to capture the output from the terminal window (using the mouse to highlight the relevant lines), and then copy and paste that information into a text editor, and save that file in the source root as "make-mrproper.scr." That file is not used again anywhere in the workflow; it is saved only for the sake of completeness.

An excerpt from the 2.0.0 make-mrproper.scr file is shown in Figure A.1:

```
make[1]: Entering directory `/home/lgt/2.00.000/linux/arch/i386/boot'
rm -f bootsect setup
rm -f bbootsect
rm -f zImage tools/build compressed/vmlinux.out
rm -f bzImage tools/bbuild compressed/bvmlinux.out
make[2]:                           Entering                           directory
`/home/lgt/2.00.000/linux/arch/i386/boot/compressed'
rm -f xtract piggyback vmlinux bvmlinux
make[2]: Leaving directory `/home/lgt/2.00.000/linux/arch/i386/boot/compressed'
make[1]: Leaving directory `/home/lgt/2.00.000/linux/arch/i386/boot'
make -C arch/i386/kernel clean
make[1]: Entering directory `/home/lgt/2.00.000/linux/arch/i386/kernel'
rm -f trampoline hexify
make[1]: Leaving directory `/home/lgt/2.00.000/linux/arch/i386/kernel'
rm -f kernel/ksyms.lst include/linux/compile.h
rm -f core `find . -name '*.[oas]' ! -regex '.*lxdialog/.*' -print`
rm -f core `find . -type f -name 'core' -print`
rm -f vmlinux System.map
rm -f .tmp* drivers/sound/configure
```

**Figure A.1 – Excerpt from the make-mrproper.scr output for the 2.00.000 kernel.**

### A.4.3 make config

This step is discussed in great detail in Chapter 5. This step sets the configuration variables, which, in turn, ultimately determines precisely which lines of code (and, in some cases, which source files) are to be used in the compilation. Unlike the output from make mrproper, the dialog resulting from make config *must* be screen-scraped (we cannot "tee-route" the output to a source file, because we need to respond to the configuration prompts), and is saved in `make-config.scr` in the source root. The collection of `make-config.scr` files (one from each kernel version) forms the input for the VB application “`Configuration Checker`,” which was discussed in Section 5.6.

### A.4.4 `make dep`

This step is similar to `make mrproper` in that it is essential for the `make` process itself, but we do not use the resulting output in this dissertation. This step builds source file dependency information for the upcoming `make` step. We can “`tee`-route” (or use the mouse to screen-scrape) the output of this step and save the result in the source root as `make-dep.scr`, for the sake of completeness.

An excerpt from the 2.00.000 `make-dep.scr` file is shown in Figure A.2:

### A.4.5 `make`

This step actually builds the kernel. We can either “`tee`-route” the output of this step to `make.scr` in the source root (with `make | tee ../make.scr`), or we can use the mouse to screen-scrape and save the result in the source root as `make.scr`. The length of this file depends on the number of files actually compiled in the process of building the kernel (plus some lines that correspond to things other than the invocation of `gcc`).

```
make[1]: Entering directory `/home/lgt/2.00.000/linux/arch/i386/boot’
make[1]: Nothing to be done for `dep’.
make[1]: Leaving directory `/home/lgt/2.00.000/linux/arch/i386/boot’
rm -f .hdepend
gawk -f scripts/depend.awk `find /home/lgt/2.00.000/linux/include -name \*.h !
–name
      modversions.h -print` > ..hdepend
mv ..hdepend .hdepend
gawk -f scripts/depend.awk init/*.c > .tmpdepend
set -e; for i in kernel drivers mm fs net ipc lib arch/i386/kernel arch/i386/mm
    arch/i386/lib; do make -C $i fastdep; done
make[1]: Entering directory `/home/lgt/2.00.000/linux/kernel’
if [ -n “dma.c exec_domain.c exit.c fork.c info.c itimer.c ksyms.c module.c
panic.c
   printk.c resource.c sched.c signal.c softirq.c sys.c sysctl.c time.c” ];
then \
        gawk -f /home/lgt/2.00.000/linux/scripts/depend.awk *.[chS] > .depend;
fi
make[1]: Leaving directory `/home/lgt/2.00.000/linux/kernel’
make[1]: Entering directory `/home/lgt/2.00.000/linux/drivers’
if [ -n “” ]; then \
        gawk -f /home/lgt/2.00.000/linux/scripts/depend.awk *.[chS] > .depend;
fi
set -e; for i in block char net  pci sbus scsi sound cdrom isdn; do make -C $i
    fastdep; done
make[2]: Entering directory `/home/lgt/2.00.000/linux/drivers/block’
if [ -n “ali14xx.c amiflop.c ataflop.c cmd640.c dtc2278.c floppy.c genhd.c hd.c
    ht6560b.c ide.c ide-cd.c ide.h ide_modes.h ide-tape.c ide-tape.h linear.c
    ll_rw_blk.c loop.c md.c promise.c promise.h qd6580.c raid0.c rd.c rz1000.c
    triton.c umc8672.c xd.c” ]; then \
        gawk -f /home/lgt/2.00.000/linux/scripts/depend.awk *.[chS] > .depend;
fi
make[2]: Leaving directory `/home/lgt/2.00.000/linux/drivers/block’
make[2]: Entering directory `/home/lgt/2.00.000/linux/drivers/char’
gcc272 -I/home/lgt/2.00.000/linux/include -o conmakehash conmakehash.c
./conmakehash cp437.uni > uni_hash.tbl
if [ -n “amigamouse.c apm_bios.c atarimouse.c atixlmouse.c baycom.c busmouse.c
    conmakehash.c console.c consolemap.c consolemap.h console_struct.h
cyclades.c
    defkeymap.c diacr.h digi_bios.h digi_fep.h digi.h fbmem.c fep.h istallion.c
    kbd_kern.h keyb_m68k.c keyboard.c lp.c lp_intern.c lp_m68k.c mem.c misc.c
    msbusmouse.c n_tty.c pcxx.c pcxx.h psaux.c pty.c random.c riscom8.c
riscom8.h
    riscom8_reg.h rtc.c scc.c selection.c selection.h serial.c softdog.c
stallion.c
    tga.c tpqic02.c tty_io.c tty_ioctl.c vc_screen.c vesa_blank.c vga.c vt.c
vt_kern.h
    wd501p.h wdt.c” ]; then \
        gawk -f /home/lgt/2.00.000/linux/scripts/depend.awk *.[chS] > .depend;
fi
set -e; for i in  ftape; do make -C $i fastdep; done
```

**Figure A.2 – Excerpt from the make-dep.scr output for the 2.00.000 kernel.**

### A.4.6 Convert `make.scr` to `make.sh` and execute `make.sh`

A typical `gcc` invocation from the 2.00.000 kernel's `make.scr` file is shown below:

```
gcc272 -D__KERNEL__ -I/home/lgt/2.00.000/linux/include -Wall
-Wstrict-prototypes -O2 -fomit-frame-pointer -fno-strength-
reduce     -D__SMP__ -pipe -m486 -malign-loops=2 -malign-
functions=2 -DCPU=686 -malign-jumps=2 -D__SMP__   -c  -o
init/main.o init/main.c
```

There are several things to note on this line:

2) The `Makefile` has been edited so that `CC` evaluates to “`gcc272`,” rather than “`gcc`.”

3) The “`–D`” compiler switches `#define` constants visible within the source file. These switches, along with those specified during the `make config` step, determine the source lines and files actually compiled.

4) The “`-c`” and “`-o`” compiler switches indicate (respectively) that the compiler is to compile (but not link) the source code, and that the output (the object code) is to be written to “`init/main.o.`” The “`o`” in “`-o`” indicates “`output,`” rather than “`object code.`”

5) All of the “`-W,`” “`-f,`” and “`-m`” compiler switches relate to warnings during parsing and code generation options, none of which we care about relative to preprocessing the code, so they will be removed. For a complete list of the `gcc` compiler switches, see the `gcc man` page.

6) The source file actually being compiled is the last argument on the line – “`init/main.c`”

In order to convert this `gcc` invocation into one that will only preprocess the code (and not attempt to generate object code), we need to:

1) Keep all “-D” switches; they set flags we need to maintain. If the same “-D” switch appears more than once, we can keep the extra instances, or we can remove them.

2) Change the “-c” compiler switch to “–E –P –C.” These three switches tell gcc:

    a. “-E” – preprocess and stop (do not parse or generate object code).

    b. “-P” – do not include #lines to show where files have been #included.

    c. “-C” – do not discard comments (we want to count comment lines later, and a different tool will eventually remove all comments).

3) Because we do not intend to parse the code or generate object code, all of the compiler switches listed in #4 above can be deleted.

4) Keep the “-o” compiler switch, which tells us where to put the output, but we do not want to write preprocessed code to a “.o” (object code, waiting to be linked) file. Rather, we replace the “.o” suffix with “_C.c.” We will henceforth use the notation that files whose names end in “_C.c” contain preprocessed code (for example, the preprocessed code for this file will go into “init/main_C.c.”

The resulting converted `gcc` invocation for this file is therefore:

```
gcc272 -D__KERNEL__ -I/home/lgt/2.00.000/linux/include -O2 –
pipe              -D__SMP__  -m486  -DCPU=686  -E –P  -C  -o
init/main_C.c init/main.c
```

There are 225 such `gcc` invocations to convert in Linux 2.00.000 (using our configuration), and 412 in 2.04.035. A program was written in Visual Basic 6.0 to read a `make.scr` file, strip all lines that are not `gcc` invocations, make the necessary changes to the remaining `gcc` lines, and write the output to a shell script file, `make.sh,` in the same directory as the `make.scr` file. This program was originally a "quick-and-dirty" little script file, but it kept evolving, and eventually became the "*n*ew and *i*mproved *Q*uick-*a*nd-*D*irty" processor, which was named NIQAD.

We can take advantage of the VMWare operating environment and its Linux-Windows shared folders in this way:

1) Build the kernel with `make`, either `tee`-routing or screen-scraping the output.
2) Copy `make.scr` from `<source root>/<kernel version>/make.scr` to `/mnt/hgfs/Share1/<kernel version>/make.scr.`
3) Use ALT+TAB to switch to Windows.
4) Run NIQAD, and convert `C:\SharedFolders\<kernel version>\make.scr` to `C:\SharedFolders\<kernel version>\make.sh`.
5) Use ALT+TAB to switch back to Linux (VMWare).

6) Copy /dev/mnt/hgfs/Share1/<kernel version>/make.sh to <source root>/<kernel version>/make.sh.

7) Open a terminal window, use cd <source root>/<kernel version> and…

8) …bash make.sh to preprocess the code for this kernel version.

### A.4.7 Remove all files except preprocessed code from the source tree

Once the make.sh shell script has executed, the source tree will contain:

- The original ".c" source files
- The compiled ".o" object files
- The preprocessed "_C.c" files
- Miscellaneous other files.

At this point, all we want are the preprocessed "_C.c" files.

As a backup, before proceeding, we create a compressed archive of this kernel version's entire source tree, which is stored in the directory "Kernel-As-Built-Archives," with file names in the format "linux-2.xx.yyy-As-Built.tar.bz2." With bzip2 compression, the 104 kernel archives require approximately 3.8 GB of storage space. As verified by the Configuration Checker, the individual kernels were properly configured, and because make completed successfully, we should never have to build the kernels again.

These compressed archives are moved from the Linux VM to Windows, and expanded into a directory for each kernel version, with the source root for each kernel version residing in C:\SharedFolders\2.xx.yyy. We wrote a program using the Windows File Scripting object to traverse the resulting directory tree and examine each file in each directory. All files whose file names do not end in "_C.c" are deleted, and if the processing of a directory leaves that directory empty, the directory itself is also deleted. Upon completion, the resulting source tree contains only the preprocessed code.

This program is called the Directory Processor. This program assumes that the source root lies in C:\SharedFolders\2.xx.yyy\Linux\, and not only does this program delete all files whose names do not end in "_C.c," it also uses the names of the surviving files to build a script to run CXREF to extract the global variable information from the preprocessed code base. This script is called CXREF.sh, and is written to C:\SharedFolders\2.xx.yyy\linux.

### A.4.8 Run CXREF Over the Entire Preprocessed Code Base

With the source root residing in C:\SharedFolders\Linux\2.xx.yyy, we switch back to the Linux VM, and execute the CXREF.sh script with the command bash CXREF.sh. CXREF requires approximately 15 minutes to process the source for each kernel (a bit longer for the larger 2.04.xxx kernels). The result of the CXREF run is a group of files whose file names are either cxref or CXREF. We move these files up one directory level to separate them from the remainder of the preprocessed source code with the commands:

```
cd /mnt/hgfs/Share1/2.xx.yyy/linux
mv CXREF.* ../
mv cxref.* ../
```

The CXREF files are rather large. The main file for each kernel is `CXREF.results`. For 2.00.000, this file was 24.8 MB in size, and for 2.04.035, it was 114.6 MB.

### A.4.9 File Cleanup and Archiving

Now that the preprocessed source and the CXREF results are separated, we take another compressed archive snapshot of the source root as a backup, this time saving the files with names using the pattern `linux-2.xx.yyy-Preprocessed.tar.bz2,` using the commands:

```
cd /mnt/hgfs/Share1/2.xx.yyy
tar –cjf linux-2.xx.yyy-Preprocessed.tar.bz2 linux
```

The entire collection of 104 archives requires approximately 2.2 GB of storage space. The `/mnt/hgfs/Share1/2.xx.yyy/linux/` subdirectory can now be deleted.

We also move the remaining CXREF files to other directories on the Windows drive for further processing.

At this point, we have completed the kernel build, and have extracted all of the information we need from the Linux VM.  Debian can be shut down, and the VM stopped.

### A.4.10 Run Klocwork on the Preprocessed Source

The Klocwork CASE tool is then run on the preprocessed source.  This process is described in detail in the following subsections.

#### A.4.10.1 Setting Up Klocwork to Gather All Metrics

Klocwork gathers some metrics at the file level, and some at the function level.  We will refer to the file-level metrics as FLMs, and to avoid confusion with "file" and "function" both starting with the letter "f," we will refer to the function-level metrics as PLMs (*procedure*-level metrics).  Although Klocwork does not directly gather any metrics at the kernel level, such as the number of source files, we will compute some metrics at the kernel level, and will refer to the resulting kernel-level metrics as KLMs.

Klocwork numbers its FLMs 7 to 45, and the PLMs from 129 to 162.  Table A.2 shows the FLMs by number.

**Table A.2 – File-Level Metrics Available in Klocwork**

| File-Fevel Metric Name / Description | # |
|---|---|
| Total number of lines in this file; calculated as (last line #) – (first line #) + 1 | 7 |
| Number of constant declarations that are declared in this file at a global level | 9 |
| Number of data items that are declared in this file at a global level | 10 |
| Number of comment sections in this file | 11 |
| Number of bytes of comments in this file | 12 |
| Number of lines of code with comments. If the line contains both code and comments, then it is included in this metric. Comments using the "//" syntax count as one line. Each visibly distinct line of multi-line comments using the "/*...*/" syntax is counted as one line. | 13 |
| Number of functions within this file | 19 |
| Sum of the number of calls to other routines for all routines in this file. See #138 below. | 20 |
| Sum of the number of conditional arcs for all routines in this file. See #150 below. | 21 |
| Maximum conditional span for all routines in this file. See #149 below. | 22 |
| Maximal value of control nesting for all routines in this file. See #146 below. | 23 |
| Sum of the logarithms of the numbers of independent paths for all routines in this file. See #136 below. | 24 |
| Total number of operands for all routines in this file. See #130 below. | 25 |
| Total number of operators for all routines in this file. See #132 below. | 26 |
| Sum of McCabe Cyclomatic Complexity metric for all routines in this file. See #135 below. | 27 |
| Sum of the number of control statements for all routines in this file. See #151 below. | 28 |
| Total number of operands (data items) defined within each routine for all routines in this file. See #148 below. | 29 |
| Sum of the number of executable statements for all routines in this file. See #141 below. | 30 |
| Sum of the LOC numbers for all routines of this file. See #129 below. | 31 |
| Sum of the number of declarative statements for all routines in this file. See #152 below. | 32 |
| Total number of statements for all routines in this file. See #142 below. | 33 |
| Complexity risk for this file: a measure of the file using key complexity characteristics that predict the likelihood of fault insertion during modification or being inherent when created | 34 |
| Sum of the number of occurrences of global variable usages for all routines in this file. See #160 and #161 below. | 35 |
| The Halstead program volume metric for this file | 38 |
| Number of Blank Lines in this file | 39 |
| Bytes of global variables declared in this file | 41 |
| Non-comment, non-blank lines of code: the number of lines of code in this file, not including comment lines and blank lines | 45 |

Table A.3 shows the PLMs by number.

**Table A.3 – Procedure-Level Metrics Available in Klocwork**

| Procedure-Level Metric Name / Description | # |
|---|---|
| Lines of code in the function; calculated as (last line number) – (first line number) + 1. Summing this metric over all functions in a file yields metric #31. | 129 |
| Number of operands (identifiers or constants) used in the function. Note that the function name is also an identifier. For this metric, a function call is considered an operator. Summing this metric across all functions in a file yields metric #25 above. | 130 |
| Number of distinct operands (variables and constants) used in the current function. Variables are distinguished by name, so usages of overridden variables do not contribute to this metric. Constants are distinguished by value, so all strings are assumed to be unique. For this metric, a function call is considered an operator. | 131 |
| Number of operators used in the function. Operators include accesses to variables, unary, binary, ternary, field access, index access, call, new instance of, expr-class, expression-this, and expression-super. All function calls are considered as operators. Summing this metric over all functions in a file yields metric #26 above. | 132 |
| Number of distinct operators used in a function. Like metric #132, but function calls are considered as one unique operator. For example, `Msg` and `printf` function calls are counted only once. | 133 |
| Number of returns – the number of `return` statements in the function (not return points) | 134 |
| McCabe Cyclomatic Complexity metric – shows the number of areas plane is divided by control flow graph. Summing this metric over all functions in a file yields metric #27 above. | 135 |
| Number of independent paths in a function, calculated as: ln(number of paths in function). All loops and if statements count as two paths. `Goto` statements are not counted in this metric. Each catch clause is an independent path. Summing this metric for all functions in a file produces metric #24 above. | 136 |
| Number of parameters (arguments) in the function. In the example (f(x, y+z, foo(g)), the parameters passed to other functions are x, y+z, and g, so the value of this metric would be 3. | 137 |
| Number of calls to unique functions. Multiple instances of calls to the same function from within a given function are counted as one instance. Summing this metric over all functions in a file yields metric #20 above. | 138 |
| Number of calls outside the class. For a function that is not a method, this metric is equal to the total number of calls | 139 |
| Number of parameters passed to other function. | 140 |
| Number of executable statements (operators such as +=, or function calls). Summing this metric over all functions in a file yields metric #30 above. | 141 |
| Number of statements (expression and control). This metric is the sum of metrics #151, #141, and #152 for the function. Summing this metric over all functions in a file yields metric #33 above. | 142 |
| Number of loops (`for`, `while`, `do`/`while`) in the current function | 143 |
| Number of conditional statements (`if`, `switch`) in the current function | 144 |
| Number of `else` and `case` statements in the current function. `Default` statements are not included. | 145 |
| Maximum level of nested control statements (`if`, `while`, `switch`, `for`, and `do`/`while`). The initial value of this metric (for example, a function with no operators) is 1. Taking the maximum of this metric over all functions in a file yields metric #23 above. | 146 |

**Table A.3, Comntnued**

| | |
|---|---|
| Average level of control nesting, calculated as the sum of the level of each executable statement for all statements in the function divided by the number of executable statements. If the number of executable statements is 0, this metric is 0. | 147 |
| Number of data items declared locally within the function. Summing this metric over all functions yields metric #29 above. | 148 |
| Maximum number of executable statements located within the span of a branch of a conditional arc. Taking the largest value of this metric over all functions in a file yields metric #22 above. | 149 |
| Number of conditional branches in the control graph of the function (`if`, `switch`, `do`, `while`, or `for` statements). Summing this metric over all functions in a file yields metric #21 above. | 150 |
| Number of control statements in a function (`if`, `switch`, `do`, `while`, `for`, `return`, `break`, `continue`, `goto`, and `try-catch`). Summing this metric over all functions in a file yields metric #28 above. | 151 |
| Number of declarative statements in the function. This differs from metric #148 in that one declaration statement can declare several local variables. Summing this metric over all functions in a file yields metric #32 above. | 152 |
| Number of accesses (read/write) to variables defined outside the function | 153 |
| Bytes of local variables declared in the function | 154 |
| Bytes of parameters for the function | 155 |
| Bytes of parameters passed to other functions | 156 |
| Number of IPC (Inter-process communication) calls | 157 |
| Number of system calls | 158 |
| Number of calls to non-prototyped functions. This metric depends on the completeness of the code. If code is compiled without includes, the metric would be the number of calls (because no functions would be prototyped). | 159 |
| Number of reads from global variables. Summing metrics 160 and 161 over all functions in a file yields metric #35 above. | 160 |
| Number of writes to global variables. Summing metrics 160 and 161 over all functions in a file yields metric #35 above. | 161 |
| The number of lines of code in a function, not including comment lines and blank lines. | 162 |

Klocwork does have metrics numbered 65-84, and 205-2116, but they are related to object-oriented code and, as such, do not apply here; Linux is written in C and Assembler. We therefore ignore these metrics completely.

Klocwork does not automatically report *any* of these metrics by default. Rather, Klocwork issues warning or error statements when we tell it to watch for the various metric values to exceed some certain threshold. If we set the warning threshold for all metrics to zero (or more accurately, the threshold *condition* to “not equal to zero”) then, whenever any metric has a nonzero value, Klocwork will issue a warning. This is accomplished by editing the file `metrics_default.mconf`. In other words, by setting the warning threshold in this fashion, Klocwork will then gather the desired metric.

Immediately after installing Klocwork, and before creating any projects, edit this file, located in `C:\Program Files\Klocwork\Development Edition\config`. Copy the file, as-installed, to some other file name in the same directory as a backup. Then edit the file in order to instruct Klocwork regarding the metrics to be computed. The *Klocwork Administration Guide* has more information on this process.

We renamed the metrics in the format *Xnnnn*, where “*X*” is either “*F*” or “P,” for the FLMs or PLMs, respectively, and *nnnn* is the four-digit zero-padded metric number from Tables A.2 and A.3. The resulting `metrics_default.mconf` file is shown in Figure A.3.

```
WARNING.SEVERITY=10 WARNING.CATEGORY=“All Klocwork Metrics-WARNINGS”
#ERROR.SEVERITY=10 ERROR.CATEGORY=“All Klocwork Metrics-ERRORS”

#                               Ent                                        Warn
#             Name              Type   Metric Expression   Err Thresh.   Thresh.
#------------------------      ----- -------------------  -----------   -------
F0007 File Size In Lines      ; FILE     ;LOC_FILE        ; 100000000;      !=0
F0008 Num Classes Declared    ; FILE     ;NOCLASSDECL     ; 100000000;      !=0
F0009 Num Const Declared      ; FILE     ;NOCONSTDECL     ; 100000000;      !=0
F0010 Num Data Items Declared ; FILE     ;NODATADECL      ; 100000000;      !=0
F0011 Num Comment Sects       ; FILE     ;NOCOMMSECT      ; 100000000;      !=0
F0012 Bytes Of Comments       ; FILE     ;BYTESCOMM       ; 100000000;      !=0
F0013 Lines Of Comments       ; FILE     ;LINESCOMM       ; 100000000;      !=0
F0014 Num Macros Defined      ; FILE     ;NOMACROS        ; 100000000;      !=0
F0015 Num Local Includes      ; FILE     ;NOLOCALINC      ; 100000000;      !=0
F0016 Num System Includes     ; FILE     ;NOSYSINC        ; 100000000;      !=0
```

**Figure A.3 – The `metrics_default.mconf` file used for this study (continues below)**

```
F0017 Num 3rd Party Includes  ; FILE     ;NO3DPINC        ; 100000000;      !=0
F0018 Total Includes          ; FILE     ;INCLDIRECTIVES  ; 100000000;      !=0
F0019 Num Routines            ; FILE     ;NOROUTINES      ; 100000000;      !=0
F0020 Calls To Other Routines ; FILE     ;RNOCALLS        ; 100000000;      !=0
F0021 Num Conditional Arcs    ; FILE     ;RNOCONDARCS     ; 100000000;      !=0
F0022 Max Conditional Span    ; FILE     ;RMAXCONDSPAN    ; 100000000;      !=0
F0023 Max Control Lev Nesting ; FILE     ;RMAXLEVEL       ; 100000000;      !=0
F0024 Sum of Log of All Paths ; FILE     ;RNOINDPATHS     ; 100000000;      !=0
F0025 Number of Operands      ; FILE     ;RNOOPUSED       ; 100000000;      !=0
F0026 Number of Operators     ; FILE     ;RNOOPRUSED      ; 100000000;      !=0
F0027 Sum McCabe Cycl Complx  ; FILE     ;RCYCLOMATIC     ; 100000000;      !=0
F0028 Num Control Statements  ; FILE     ;RNOCONTROLSTAT  ; 100000000;      !=0
F0029 Num Local Data Items Dcl; FILE     ;RNOLOCDECL      ; 100000000;      !=0
F0030 No EXecutable Statements; FILE     ;RNOEXSTAT       ; 100000000;      !=0
F0031 Routines Lines Of Code  ; FILE     ;RLOC            ; 100000000;      !=0
F0032 No Declarative Statemnts; FILE     ;RNODECLSTAT     ; 100000000;      !=0
F0033 No Statements Total     ; FILE     ;RNOSTAT         ; 100000000;      !=0
F0034 Complexity Risk         ; FILE     ;RISK            ; 100000000;      !=0
F0035 Global Variable USes    ; FILE     ;RNOACCTOGLOB    ; 100000000;      !=0
F0036 Class Methods           ; FILE     ;CNOMETH         ; 100000000;      !=0
F0037 Ways to Access Class    ; FILE     ;CNOMSG          ; 100000000;      !=0
F0038 Halstead Volume         ; FILE     ;HALSTEADVOL     ; 100000000;      !=0
F0039 Number of  Blank Lines  ; FILE     ;BLANKLINES      ; 100000000;      !=0
F0040 Compiler Directives     ; FILE     ;DIRECTIVES      ; 100000000;      !=0
F0041 Bytes of Globals Declard; FILE     ;BYTESGLDATADECL ; 100000000;      !=0
F0042 Max Include file Nesting; FILE     ;MAXINCNEST      ; 100000000;      !=0
F0043 Total No Included Files ; FILE     ;TOTALINC        ; 100000000;      !=0
F0044 CRC32 Checksum          ; FILE     ;CHECKSUM        ; 100000000;      !=0
F0045 Non_Comm Non_Blank Lines; FILE     ;NCNBLOC_FILE    ; 100000000;      !=0
#
# Now the function-level metrics.  Prefix their names with “P” for “Procedure”
#
P0129 Function Lines Of Code       ;FUNCTION;LOC_METHOD   ; 100000000;      !=0
P0130 Num Operands Used            ;FUNCTION;NOOPUSED     ; 100000000;      !=0
P0131 Num Distinct Operands Used   ;FUNCTION;NODISOPUSED  ; 100000000;      !=0
P0132 Num Opertaors Used           ;FUNCTION;NOOPRUSED    ; 100000000;      !=0
P0133 Num Discinct Operators Used  ;FUNCTION;NODISOPRUSED ; 100000000;      !=0
P0134 Number of Returns            ;FUNCTION;NORET        ; 100000000;      !=0
P0135 Cyclomatic Complexity        ;FUNCTION;CYCLOMATIC   ; 100000000;      !=0
P0136 Number of Independent Paths  ;FUNCTION;NOINDPATHS   ; 100000000;      !=0
P0137 Number of Parameters         ;FUNCTION;NOPARMS      ; 100000000;      !=0
P0138 Num Calls to Unique functions;FUNCTION;NOCALLS      ; 100000000;      !=0
P0139 Num Calls Outside Class      ;FUNCTION;NOCALLSOC    ; 100000000;      !=0
P0140 Num Pameters Passed outbound ;FUNCTION;NOPAROTHER   ; 100000000;      !=0
P0141 Num of executable statements ;FUNCTION;NOEXSTAT     ; 100000000;      !=0
P0142 Number of statements         ;FUNCTION;NOSTAT       ; 100000000;      !=0
P0143 Number of Loops              ;FUNCTION;NOLOOPS      ; 100000000;      !=0
P0144 Number of conditional stmts  ;FUNCTION;NOIF         ; 100000000;      !=0
P0145 Number of Case and Else stmts;FUNCTION;NOBRANCH     ; 100000000;      !=0
P0146 Max Level of control nesting ;FUNCTION;MAXLEVEL     ; 100000000;      !=0
P0147 Avg Level of control nesting ;FUNCTION;AVERLEVEL    ; 100000000;      !=0
P0148 Number of Local Declarations ;FUNCTION;NOLOCDECL    ; 100000000;      !=0
P0149 Max exec stmts in cond arc   ;FUNCTION;MAXCONDSPAN  ; 100000000;      !=0
P0150 Num cond arcs in control grph;FUNCTION;NOCONDARCS   ; 100000000;      !=0
P0151 Num Control Statements       ;FUNCTION;NOCONTROLSTAT; 100000000;      !=0
P0152 Num Declarative Statements   ;FUNCTION;NOMDECLSTAT  ; 100000000;      !=0
P0153 Num Accesses to Global vars  ;FUNCTION;NOACCTOGLOB  ; 100000000;      !=0
P0154 Bytes declared local         ;FUNCTION;BYTESLOCDECL ; 100000000;      !=0
P0155 Bytes of parameters for func ;FUNCTION;BYTESPARMS   ; 100000000;      !=0
P0156 Bytes of parameters passed on;FUNCTION;BYTESPAROTHER; 100000000;      !=0
P0157 Num of IPC calls             ;FUNCTION;NOIPCCALLS   ; 100000000;      !=0
P0158 Num of System calls          ;FUNCTION;NOSYSCALLS   ; 100000000;      !=0
P0159 Calls to non-prototyped funcs;FUNCTION;NOCALLSNP    ; 100000000;      !=0
P0160 Num Reads of Global vars     ;FUNCTION;READFGLOBAL  ; 100000000;      !=0
P0161 Num writes to Global vars    ;FUNCTION;WRITETGLOBAL ; 100000000;      !=0
P0162 Non-comment, non-blank lines ;FUNCTION;NCNBLOC_METHOD;100000000;      !=0
```

**Figure A.3 ,Continued**

### A.4.10.2 Running Klocwork

Once properly installed and configured, Klocwork places the results of its source code analysis in a subdirectory it calls "project_root." This directory can be located anywhere on the machine, but be forewarned that it will become extremely large. In processing the kernels for this dissertation, the resulting project_root and all its associated subdirectories totaled over 124 GB. Most of the resulting files are text-based, however, and will work quite well in a compressed folder under NTFS.

For each kernel, point Klocwork at the source root (the directory immediately below /linux/). The easiest way to create the Klocwork projects is to right-click the project tree in KMC (the Klocwork Management Console) and specify "New Project." From this point, simply follow the prompts to create the project, but do not start building it yet. In order to facilitate using the same tools over all versions, we named the projects Linux_2_xx_yyy.

When Klocwork runs on the Windows platform, it assumes that we are not using gcc, which has numerous compiler extensions. In order to tell Klocwork to parse the code as gcc would, we need to change the compiler options associated with this project. To do so, open a command window, and issue the following commands, where "<PN>" is replaced with the project name:

```
cd \Program Files\Klocwork_7.5.1.10\DevelopmentEdition\bin
kwadmin get-project-properties <PN>
```

Note the existing compiler options, because we are about to have to re-type them

```
kwdamin set-project-property <PN> compiler-options <existing options> “-
e,gnu”
```

Once the compiler options are set, the project can be built. It is possible to have Klocwork build more than one project at a time, but it causes more contention for resources than it saves through multitasking, so build one project at a time. This can take as little as 15 minutes or so for the smaller kernels, and as much as 4 hours for the larger ones. For an overnight run, starting two at one time works well. Klocwork is also resource-hungry, particularly during the linking phase. It should be run with at least 2 GB of RAM.

### A.4.11 Processing the output of Klocwork

The main file that Klocwork creates that is of interest to us is called `report.txt`. This file is located at `<project root>\Linux_2_xx_yyy\builds\build_n\reports`. For the 2.00.000 kernel, it was 96 MB, and for 2.04.035, it was 971 MB in size.

A single record from a `report.txt` file is shown in Figure A.4. These semicolon-delimited records are quite long. For purposes of illustration, the fields are shown in Figure A.4 one-per-line, and the field numbers (in square brackets) have been added for clarity:

```
[01]  C:\KWSource\2.00.000\linux\drivers\net\3c59x_C.c;
[02]  1;
[03]  0;
[04]  Info;
[05]  10;
[06]  All Klocwork Metrics-WARNINGS;
[07]  METRICS.W. F0007_File_Size_In_Lines______;
[08]  Warning;
[09]  3c59x_C.c;
[10]  -294901307;
[11]  Violated metric “F0007_File_Size_In_Lines______”: 3c59x_C.c 12806!=0;
[12]  ;
[13]  New;
[14]  Analyze;
[15]  All Klocwork Metrics-WARNINGS
```

**Figure A.4 – One record from `report.txt`, broken into individual fields.**

The keys to processing this record lie in field #11. The metric being reported in this record bears the description following “Violated metric,” and comes directly from the first column of the `metrics_default.mconf` file we created above.

Because we have set up warning conditions of “not equal to zero,” the warning in this case occurred because the observed value of 12806 violates the “!=0” condition in `metrics_default.mconf.` Hence, we know that the number of lines in the file `3c59x_C.c` is 12806. Given that the size of the `report.txt` files ranges from (approximately) 100 to 1000 MB, parsing this file manually to get the actual metric values is obviously not practical. To meet this need, we created another program, `KW Report Parser`, to parse `report.txt` and consolidate its output into another format.

The first version of this program parsed `report.txt` and generated a text file intended to be imported into Excel, but the number of functions (kernel-wide) grew too large for Excel's 65,535-row limit to handle (there are metrics for 174,409 rows in the `report.txt` file for version 2.04.035 of the kernel). At that point, it was clear that a relational database would be required, and the report parser was modified to output records to a file that could subsequently be loaded to a database.

The output of this file is a tab-delimited file containing 77 fields:

1) The File Name, including the path relative to the directory immediately below `/linux/`

2) The Function Name. If the function name (Field #9 of Figure A.4) matches the end of the file name (Field #1 of Figure A.4), then this record contains FLM values; otherwise, it contains PLM values.

3) The Line Number within the file. If this is a PLM record, then this is the line number within the file at which the function starts. If this is an FLM record, this will be 1.

4) The Column Number on the line. We do not use this field in the research presented in this dissertation.

5-43) FLMs numbered 7-45, as listed in the FLM table (Table A.2). If this is a PLM record (see #2 above), these will all be blank; if this is an FLM record, these will contain the FLMs and the PLMs will be blank

44-77) PLMs numbered 129-162, as listed in the PLM table above.

The output of this program is a file called `Report-non-aggregated.CSV`, located in the same directory as `report.txt`. The size of this file is 3.9 MB for 2.00.000, and 25.9 MB for 2.04.035. Compared with 96 and 971 MB (respectively) for the raw `report.txt` files, these are certainly still large, but much more manageable than their predecessors.

The 2.00.000 version of this file was loaded into MS Excel. A screen shot of this file is displayed in Figure A.5 at 25 percent zoom. Although far too small to be readable, it provides a rough idea of the file's contents, and the sparseness of the matrix.

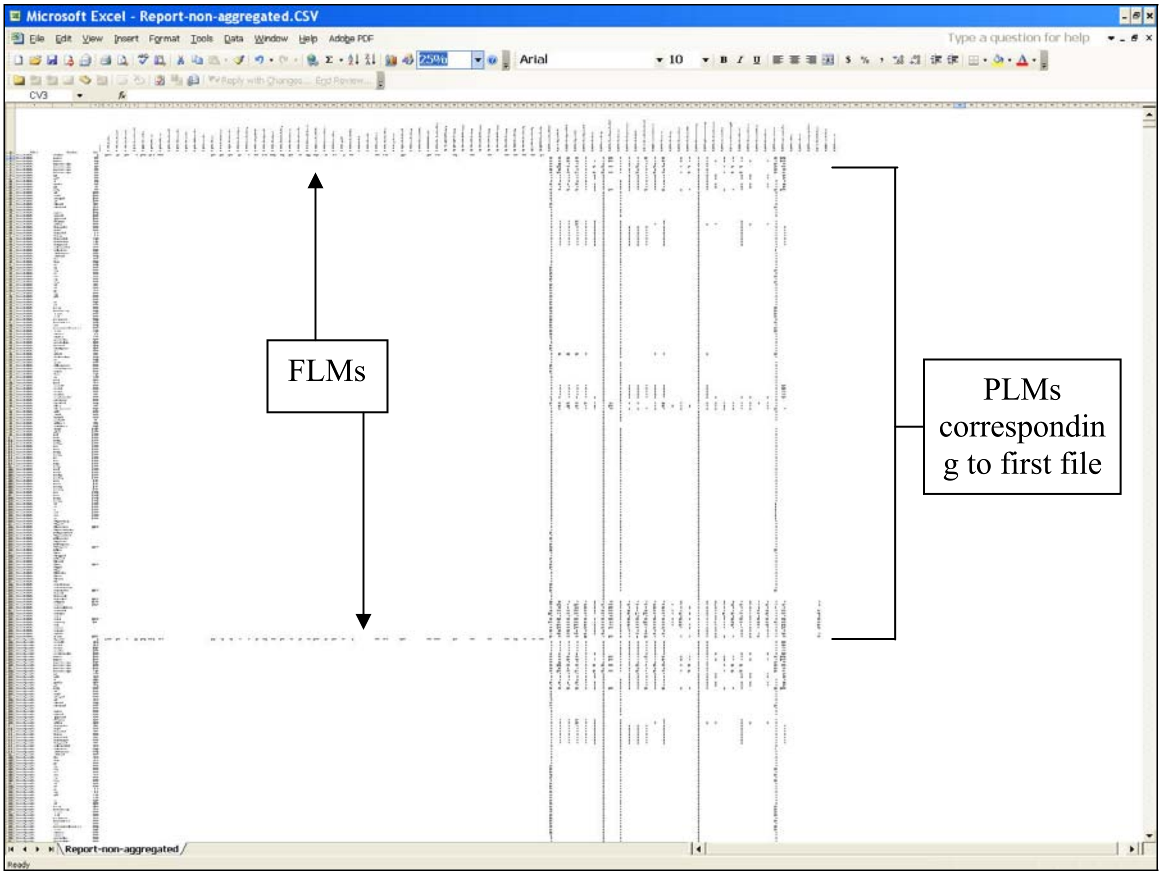

Microsoft Excel - Report-non-aggregated.CSV
File Edit View Insert Format Tools Data Window Help Adobe PDF
Type a question for help
25%
Arial
10
CV3
FLMs
PLMs correspondin g to first file
Report-non-aggregated
Ready

**Figure A.5 – Screen shot of `Report-non-aggregated.CSV`, displayed in Excel at 25%**

In addition to the FLM and PLM values, another piece of information we need later is a list of all the files and functions (kernel-wide), including the line numbers on which each function starts, and the number of lines of code in each function. This information is required to determine when a global variable appears inside a function, and, if so, in which function it appears.

If a GV appears outside a function, then it must be in a declaration (or a comment), and we do not count it as an instance of a GV, because it cannot possibly be used or its value modified in a declaration.

Consider also the possibility of a global variable X, declared at the module level, made global by the use of `extern`, and then re-declared at the local level within some function Y. In such a case, the rules of variable scope and visibility apply, and *no* instance of X within function Y is a reference to the *same* X as was declared globally. As such, the instances of X within Y are not truly instances of a GV at all. In order to be able to tell which GVs appear in which functions, and are not re-declared at a more local scope, we need to know where every function starts and ends. This information is eventually combined with the output of `CXREF` (see below).

The first step in extracting this information is to simply parse `report.txt`, looking for "P0129" (PLM #129 – the number of lines of code within a function). Because every record in `report.txt` contains the file name, the function name, and the line number, the complete set of P0129 records gives us all of the file/function/line number information we ultimately need.

The simplest way to comb through `report.txt`, keeping only the records containing “P0129” was to use the command prompt within Windows, and pipe the raw data through the `find` filter, and then redirect the output to a new file:

```
type report.txt | find “P0129” >Report_Func_Line_Counts.txt
```

### A.4.12 Processing the output of `CXREF`

As mentioned in Section A.4.8, the output of `CXREF`, `CXREF.results`, is a text file ranging in size from about 25 MB in 2.00.000 to approximately 115 MB in 2.04.035, and containing information about files, functions, types, `typedef`s, and global variables.

Because we are not concerned with any output from `CXREF` other than the `FILE` lines and `VARIABLE` blocks (and a kernel-wide listing of the global variables at the end of the file), the `CXREF` output needs to be filtered to remove everything except these two items. The raw output from `CXREF` is quite voluminous. As such, manually filtering this output is not practical, and a software tool is required to automate the filtering of this output for examination of the global variables. Figure A.6 is an excerpt from the `CXREF` output for `kernel/irq.c`.

```
VARIABLE : bh_active [Global and External]
Defined: kernel/irq.c:3044
Type: unsigned long bh_active
Declared global in ‘kernel/irq.c’
Visible in drivers/char/console.c
Visible in drivers/char/keyboard.c
Visible in drivers/char/pty.c
Visible in drivers/char/serial.c
Visible in drivers/char/tty_io.c
Visible in drivers/char/vt.c
Visible in drivers/net/8390.c
Visible in kernel/irq.c
Used in do_bottom_half : kernel/irq.c
Used in init_IRQ : kernel/irq.c
Used in mark_bh : kernel/irq.c
```

**Figure A.6 – Excerpt from CXREF output showing visibility and usage.**

Variable bh_active is declared as global in irq.c, and is Visible in irq.c (naturally, because that is where its definition lies), as well as in seven other files, because they include an extern bh_active declaration. Although this variable is Visible in a total of seven files, it is Used in only one – the file in which it was declared, irq.c. The VARIABLE line in this case also indicates that this variable is [Global and External]. By examining the attributes on the VARIABLE lines in the CXREF output, we can distinguish between local and global variables declared at the global level.

Yet another program, CXREF Parser, was written to process the CXREF.results file, and extract only the information we need into a file called CXREF.dat. The format of this file is rather straightforward:

- The first section is a list of all files compiled in the process of the kernel build, with path names relative to the first directory below `/linux/`.
- The second section is a list of all global variable names used *somewhere* in the kernel build. As discussed at the end of Section A.4.11, not every instance of these variable names is necessarily an instance of a global variable.
- The third section is a matrix with a row for every file, and a column for every global variable (GV). The matrix entries are:
  - 0 if the GV does not appear in the file at all;
  - 1 if the GV is *visible* in the GV (declared `extern`) but not *used* in the file (i.e., the GV does not appear in executable code somewhere in a function); or
  - 2 if the GV is used in at least one function within the file.

A sample of the 2.00.000 `CXREF.dat` file is shown in Figure A.7.

```
Files
 225
arch/i386/kernel/bios32_C.c
arch/i386/kernel/hexify_C.c
arch/i386/kernel/ioport_C.c
   <219 file names omitted>
net/unix/af_unix_C.c
net/unix/garbage_C.c
net/unix/sysctl_net_unix_C.c

Globals
 431
AudioDrive
C_A_D
EISA_bus
Jazz16_detected
  <425 GV names omitted>
x86_mask
x86_model
x86_vendor_id

000000000000000000000000000000000000000000000000000000000000000000...
000000000000000000000000000000000000000000000000000000000000000000...
00100101000000100010000010000000000010100100001000100001111100211010...
00100101000000200010000010000011110010100100001000100001111100111010...
0000000000000010002000000000000000000000100000000000000001211100000000...
00100101000000100010000010000000000010100100001000100001111100211010...
00100101000000100010000010000011110010100100001000100001211100211010...
00100101000000100010000010000000000010100100001000100001111100211010...
00100201000000100010000210000000000010100100001000100001211200111010...
00100101000000100010000010000000000010100100001000100001111100211010...
00100101000000200221200010000000000010200100001000100002222200111010...
00100101000000100010000010000000000010100100001000100001111100211010...
00100101000000100010000010000021110010100100001000100001111100111010...
```

**Figure A.7 – Sample of `cxref.dat` for the 2.00.000 kernel**

The program also displays the matrix, with the rows and columns labeled for more detailed manual inspection, as shown in Figure A.8. The user may click on a file name in the leftmost column in the grid shown in Figure A.8 to see which GVs are both declared in and used in that file, or the user may click on a GV name in the top row to see in which files that GV is declared and used.

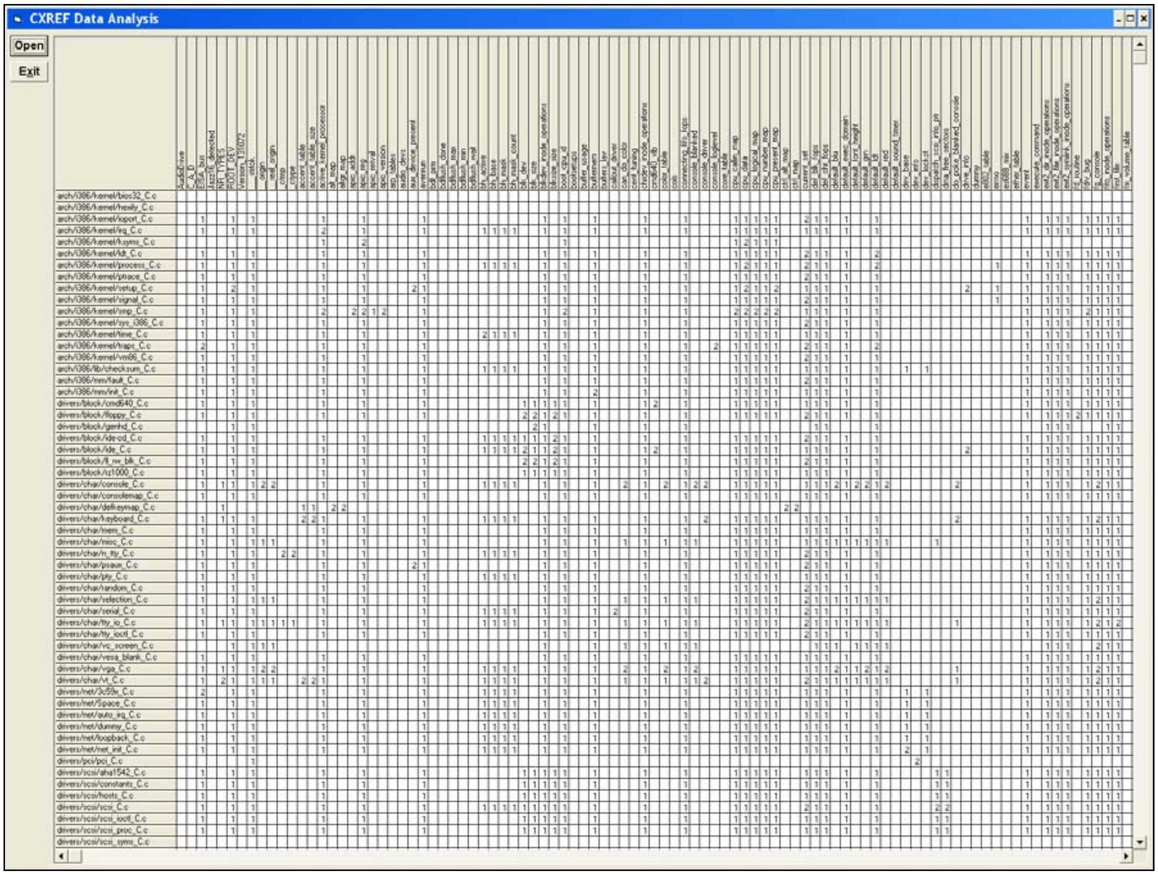


**Figure A.8 – Screen Shot from `New CXREF Processor`**

### A.4.13 Putting It All Together and Verifying the Results

We now have the following pieces (one set per kernel version) in various files:

1) The FLM and PLM values supplied by Klocwork, and parsed and distilled into a format we can manage and use (`Report-non-aggregated.CSV`).
2) A list of the GV names and files in which they are used (`CXREF.dat`).
3) A list of the functions within each file, the line number on which they start, and the length of each function in lines (`Report_Func_Line_Counts.txt)`.
4) The preprocessed source code itself (in the `.tar.bz2` archives created as discussed in Section A.4.9).

We now have the information we need in order to be able to scan through the source code (4), looking for occurrences of GVs (2), and because we can tell when we are inside a function (and which one) or outside a function (3), we can tell which instances of the GVs actually occur in executable code, as opposed to declarations outside of code.

### A.4.14 The Database

After realizing that Excel would not be able to handle the large number of records produced from the Klocwork output, it was clear that a relational database would be required. MySQL Community Server 5.0.51 was ultimately selected to house the data for two reasons:

1) Preliminary estimates indicated that the total data size would exceed 4 GB. Microsoft Access has a 2 GB file size limit. The free version of Microsoft SQL Server 2005 (SQL Server Express Edition) has a 4 GB file limit, as does the free

version of Oracle. SQL Server 2005 Standard edition has no size limit, but a five-user license would have cost $1,849, and would not have provided a solution other researchers could readily use. MySQL Community Server is available at no cost under the GPL (GNU General Public License).

2) The tools we were using easily interface with data sources that comply with the ODBC (Open DataBase Connectivity) standard. MySQL is ODBC-compliant when used with the ODBC Driver (Connector/ODBC).

The database was designed with five tables - one each for the FLM, PLM, and KLM metrics, and two lookup tables to hold the file and function names (to keep the data size down by eliminating duplicated strings).

The database structure is shown in Figure A.9. Most of the fields in the file- and procedure-level database tables (FLM and PLM, respectively) are Klocwork-supplied metrics, which we described in Tables A.2 and A.3. We created a few extra fields, to hold computed, intermediate, or supplemental metrics. We describe those, as well as the fields in the KLM (kernel-level metrics) table, in Table A.4. Fields with “ID” or “PTR” in their names are used to join the tables, and the fields “Version” and “ReleaseDate” are self-explanatory.

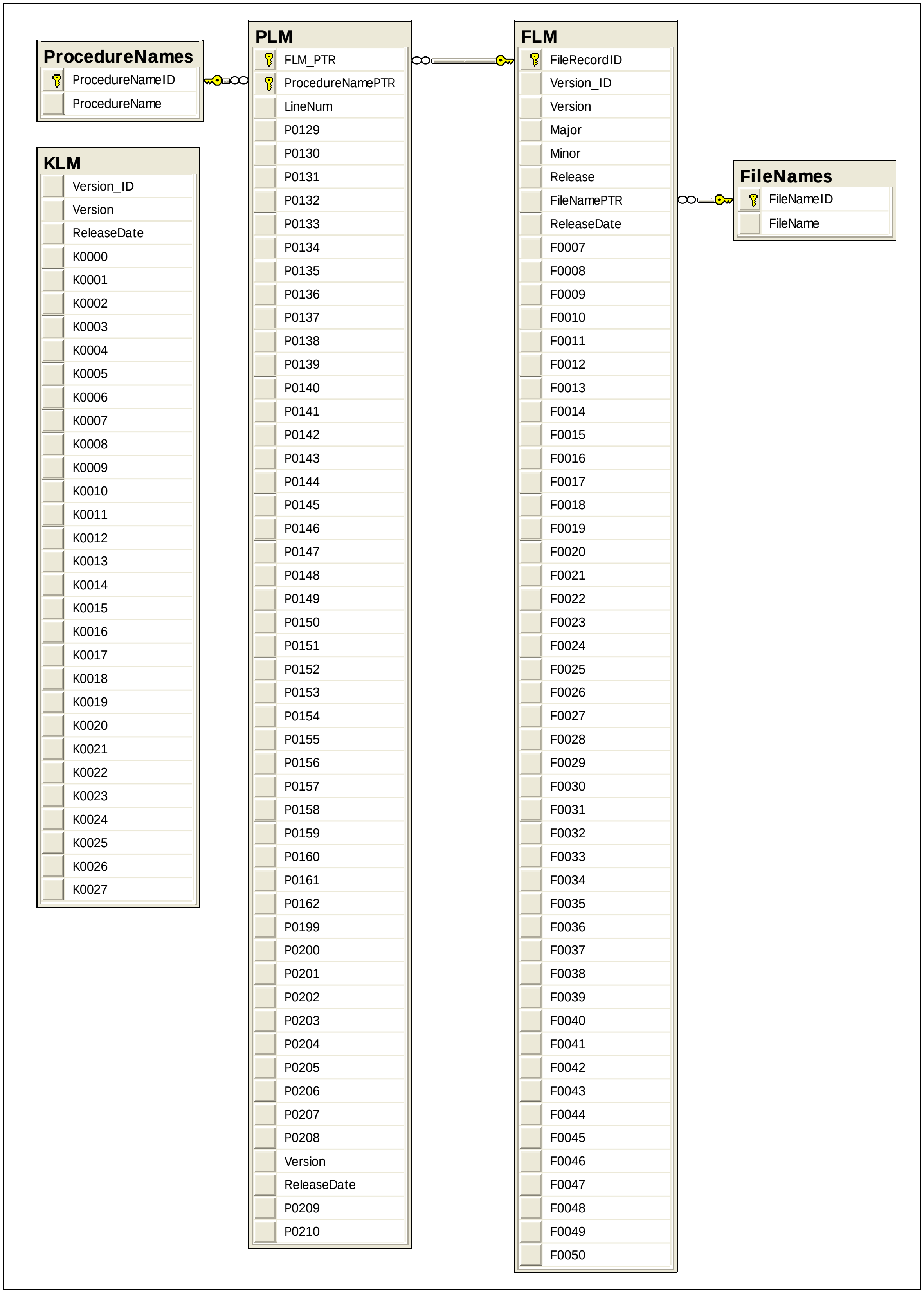


**Figure A.9 – Structure of the Database**

**Table A.4 – The Remaining Fields in the Database**

| Table | Field Name | Description |
|---|---|---|
| FLM | F0046 | Number of NCNB Lines in functions (sum of P0162 for all functions in this file) |
| | F0047 | File size, in bytes |
| | F0048 | Halstead Volume for this file |
| | F0049 | Halstead Effort for this file |
| | F0050 | Oman’s Maintainability Index for this file |
| | | |
| PLM | P0199 | Halstead Volume for this Procedure |
| | P0200 | Number of Kind-0 GV Instances in this procedure |
| | P0201 | Number of Kind-1 GV Instances in this procedure |
| | P0202 | Number of Kind-2 GV Instances in this procedure |
| | P0203 | Number of Kind-3 GV Instances in this procedure |
| | P0204 | Number of Kind-4 GV Instances in this procedure |
| | P0205 | Number of Kind-5 GV Instances in this procedure |
| | P0206 | Number of Kind-6 GV Instances in this procedure |
| | P0207 | Number of Kind-7 GV Instances in this procedure |
| | P0208 | Number of Kind-8 GV Instances in this procedure |
| | P0209 | Halstead Difficulty for this procedure |
| | P0210 | Halstead Effort for this procedure |
| | | |
| KLM | K0000 | Total Common Coupling count in this version of the kernel |
| | K0001 | Total number of Kind-1 instances in this version of the kernel |
| | K0002 | Total number of Kind-2 instances in this version of the kernel |
| | K0003 | Total number of Kind-3 instances in this version of the kernel |
| | K0004 | Total number of Kind-4 instances in this version of the kernel |
| | K0005 | Total number of Kind-5 instances in this version of the kernel |
| | K0006 | Total number of Kind-6 instances in this version of the kernel |
| | K0007 | Total number of Kind-7 instances in this version of the kernel |
| | K0008 | Total number of Kind-8 instances in this version of the kernel |
| | K0009 | Total of *all* Kinds of GV instances (K0001 – K0008) |
| | K0010 | GV Instances, only counting one instance per file |
| | K0011 | Number of distinct GV names across all files in this version |
| | K0012 | Number of source files in this version of the kernel |
| | K0013 | Number of Global Variables according to CXREF |
| | K0014 | Visibility-based (extern) pseudo-coupling count |
| | K0015 | Actual coupling count, based on CXREF data |
| | K0016 | Number of entries in CXREF.dat matrix containing “1” |
| | K0017 | Number of entries in CXREF.dat matrix containing “2” |
| | K0018 | Code churn per day |
| | K0019 | Days since last release |
| | K0020 | Average Halstead Volume |
| | K0021 | Sum of Halstead Volume |
| | K0022 | Average Halstead Effort |
| | K0023 | Sum of Halstead Effort |
| | K0024 | Total NCNB Lines in all files in this kernel |
| | K0025 | Average number of NCNB lines per file |
| | K0026 | Total MCC |
| | K0027 | Average MCC |

### A.4.15 Loading the Database

The database loading was accomplished in three phases. Our program `Database Loader` reads the file `report-non-aggregated.CSV` created as discussed in Section A.4.11, and handles loading the FLM and PLM records for all metrics originating from Klocwork. In the process of creating these records, `Database Loader` also populates the `FileNames` and `ProcedureNames` tables.

The second phase of the database loading process, loading the GV Kind data, is handled by our program `GV_Data_Processor_&_DB_Loader`, as discussed in Section A.4.13.2.

The third phase is an ad-hoc clean-up phase. For example, the initial database design did not include the `ReleaseDate` field in the FLM and PLM tables. The data for this field was obtained from `www.LinuxHQ.com`, and copied and pasted into a series of one-line UPDATE queries, examples of which are shown below:

```
UPDATE PLM SET ReleaseDate=‘1996-06-09’ WHERE Version=‘2.00.000’
UPDATE PLM SET ReleaseDate=‘1996-07-03’ WHERE Version=‘2.00.001’
UPDATE PLM SET ReleaseDate=‘1996-07-05’ WHERE Version=‘2.00.002’
UPDATE PLM SET ReleaseDate=‘1996-07-06’ WHERE Version=‘2.00.003’
UPDATE PLM SET ReleaseDate=‘1996-07-08’ WHERE Version=‘2.00.004’
...
```

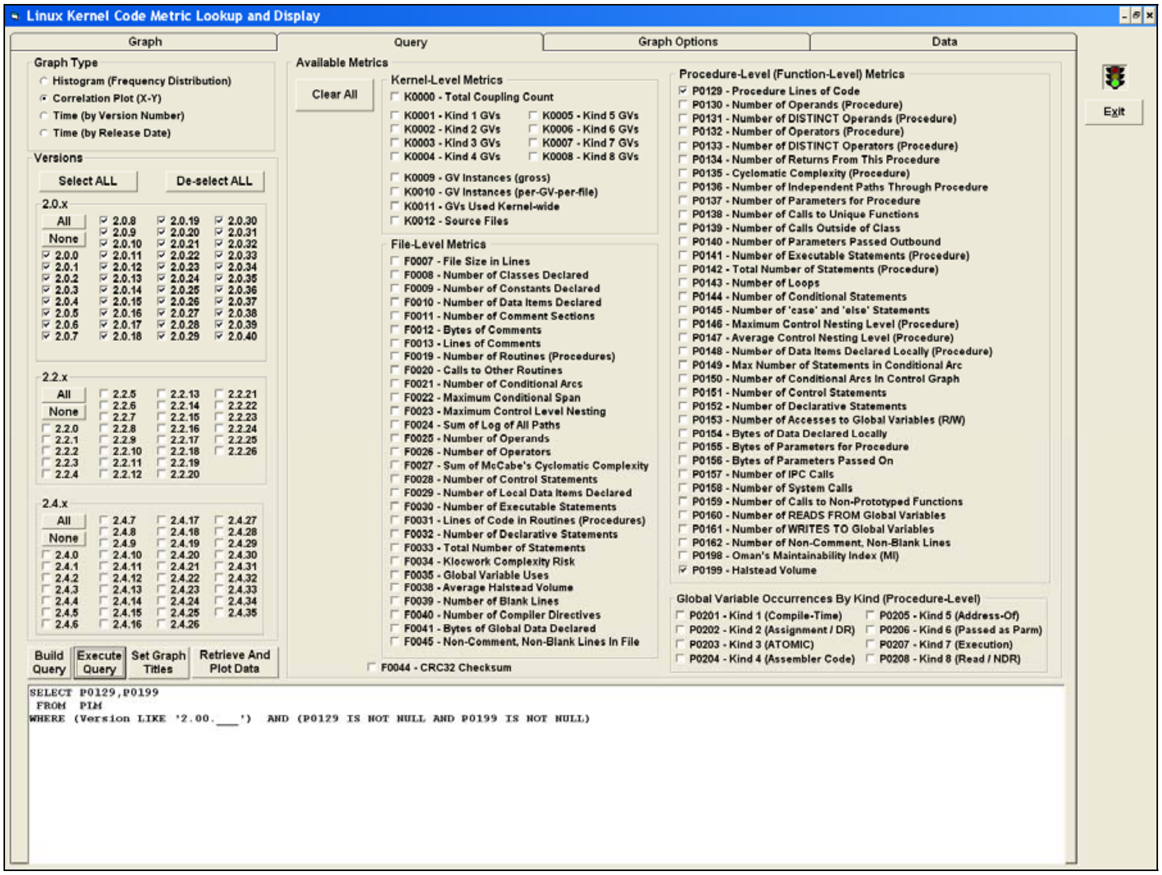
### A.4.16 Extracting Information from the Database - `Grapher`

Although a great deal of effort went into loading the data *into* the database, the data is of no value until we begin to get information *out* of the database. The final tool we created for this project is called `Grapher`. This GUI front-end to the database allows the user to select the field(s) to graph, the kernel version(s) from which to pull data, and the type of graph to draw. `Grapher` then creates the appropriate SQL query, executes it, retrieves the resulting data, and displays it in graphical form, placing the graph on the clipboard for pasting into other applications. The degree to which this facilitates extraction of information from the database cannot be understated.

`Grapher` supports four types of graphs:

1) Histograms;
2) Correlation (X-Y) plots. For correlation plots, `Grapher` displays the data as points, draws a least-squares-fit line through the data, and shows the Spearman Rank Correlation Coefficient, the corresponding *t*-value and its *P*-value, and the equation of the least-squares fit line;
3) Line graphs by kernel version number (one data point per kernel version); and
4) Line graphs by kernel release date (one data point per kernel version).

Figure A.10 shows `Grapher`'s Query tab. In this example, we have selected a correlation plot of PLMs P0129 (Lines of Code in the procedure) and P0199 (Halstead Volume) for all procedures in all files of the 2.00.xxx kernel, and clicked on the "Build Query" button. `Grapher` has correctly produced the SQL query shown in the box at the bottom of the window. For any queries not automatically created, the user can type the query of their choice directly into the box at the bottom of the window.



Graph Type
- ○ Histogram (Frequency Distribution)
- ● Correlation Plot (X-Y)
- ○ Time (by Version Number)
- ○ Time (by Release Date)

Versions

Select ALL | De-select ALL

2.0.x

All | None

☑ 2.0.0 ☑ 2.0.1 ☑ 2.0.2 ☑ 2.0.3 ☑ 2.0.4 ☑ 2.0.5 ☑ 2.0.6 ☑ 2.0.7 ☑ 2.0.8 ☑ 2.0.9 ☑ 2.0.10 ☑ 2.0.11 ☑ 2.0.12 ☑ 2.0.13 ☑ 2.0.14 ☑ 2.0.15 ☑ 2.0.16 ☑ 2.0.17 ☑ 2.0.18 ☑ 2.0.19 ☑ 2.0.20 ☑ 2.0.21 ☑ 2.0.22 ☑ 2.0.23 ☑ 2.0.24 ☑ 2.0.25 ☑ 2.0.26 ☑ 2.0.27 ☑ 2.0.28 ☑ 2.0.29 ☑ 2.0.30 ☑ 2.0.31 ☑ 2.0.32 ☑ 2.0.33 ☑ 2.0.34 ☑ 2.0.35 ☑ 2.0.36 ☑ 2.0.37 ☑ 2.0.38 ☑ 2.0.39 ☑ 2.0.40

2.2.x

All | None

□ 2.2.0 □ 2.2.1 □ 2.2.2 □ 2.2.3 □ 2.2.4 □ 2.2.5 □ 2.2.6 □ 2.2.7 □ 2.2.8 □ 2.2.9 □ 2.2.10 □ 2.2.11 □ 2.2.12 □ 2.2.13 □ 2.2.14 □ 2.2.15 □ 2.2.16 □ 2.2.17 □ 2.2.18 □ 2.2.19 □ 2.2.20 □ 2.2.21 □ 2.2.22 □ 2.2.23 □ 2.2.24 □ 2.2.25 □ 2.2.26

2.4.x

All | None

□ 2.4.0 □ 2.4.1 □ 2.4.2 □ 2.4.3 □ 2.4.4 □ 2.4.5 □ 2.4.6 □ 2.4.7 □ 2.4.8 □ 2.4.9 □ 2.4.10 □ 2.4.11 □ 2.4.12 □ 2.4.13 □ 2.4.14 □ 2.4.15 □ 2.4.16 □ 2.4.17 □ 2.4.18 □ 2.4.19 □ 2.4.20 □ 2.4.21 □ 2.4.22 □ 2.4.23 □ 2.4.24 □ 2.4.25 □ 2.4.26 □ 2.4.27 □ 2.4.28 □ 2.4.29 □ 2.4.30 □ 2.4.31 □ 2.4.32 □ 2.4.33 □ 2.4.34 □ 2.4.35

Build Query | Execute Query | Set Graph Titles | Retrieve And Plot Data

Available Metrics

Clear All

Kernel-Level Metrics
- □ K0000 - Total Coupling Count
- □ K0001 - Kind 1 GVs
- □ K0002 - Kind 2 GVs
- □ K0003 - Kind 3 GVs
- □ K0004 - Kind 4 GVs
- □ K0005 - Kind 5 GVs
- □ K0006 - Kind 6 GVs
- □ K0007 - Kind 7 GVs
- □ K0008 - Kind 8 GVs
- □ K0009 - GV Instances (gross)
- □ K0010 - GV Instances (per-GV-per-file)
- □ K0011 - GVs Used Kernel-wide
- □ K0012 - Source Files

File-Level Metrics
- □ F0007 - File Size in Lines
- □ F0008 - Number of Classes Declared
- □ F0009 - Number of Constants Declared
- □ F0010 - Number of Data Items Declared
- □ F0011 - Number of Comment Sections
- □ F0012 - Bytes of Comments
- □ F0013 - Lines of Comments
- □ F0019 - Number of Routines (Procedures)
- □ F0020 - Calls to Other Routines
- □ F0021 - Number of Conditional Arcs
- □ F0022 - Maximum Conditional Span
- □ F0023 - Maximum Control Level Nesting
- □ F0024 - Sum of Log of All Paths
- □ F0025 - Number of Operands
- □ F0026 - Number of Operators
- □ F0027 - Sum of McCabe's Cyclomatic Complexity
- □ F0028 - Number of Control Statements
- □ F0029 - Number of Local Data Items Declared
- □ F0030 - Number of Executable Statements
- □ F0031 - Lines of Code in Routines (Procedures)
- □ F0032 - Number of Declarative Statements
- □ F0033 - Total Number of Statements
- □ F0034 - Klocwork Complexity Risk
- □ F0035 - Global Variable Uses
- □ F0038 - Average Halstead Volume
- □ F0039 - Number of Blank Lines
- □ F0040 - Number of Compiler Directives
- □ F0041 - Bytes of Global Data Declared
- □ F0045 - Non-Comment, Non-Blank Lines In File
- □ F0044 - CRC32 Checksum

Procedure-Level (Function-Level) Metrics
- ☑ P0129 - Procedure Lines of Code
- □ P0130 - Number of Operands (Procedure)
- □ P0131 - Number of DISTINCT Operands (Procedure)
- □ P0132 - Number of Operators (Procedure)
- □ P0133 - Number of DISTINCT Operators (Procedure)
- □ P0134 - Number of Returns From This Procedure
- □ P0135 - Cyclomatic Complexity (Procedure)
- □ P0136 - Number of Independent Paths Through Procedure
- □ P0137 - Number of Parameters for Procedure
- □ P0138 - Number of Calls to Unique Functions
- □ P0139 - Number of Calls Outside of Class
- □ P0140 - Number of Parameters Passed Outbound
- □ P0141 - Number of Executable Statements (Procedure)
- □ P0142 - Total Number of Statements (Procedure)
- □ P0143 - Number of Loops
- □ P0144 - Number of Conditional Statements
- □ P0145 - Number of 'case' and 'else' Statements
- □ P0146 - Maximum Control Nesting Level (Procedure)
- □ P0147 - Average Control Nesting Level (Procedure)
- □ P0148 - Number of Data Items Declared Locally (Procedure)
- □ P0149 - Max Number of Statements in Conditional Arc
- □ P0150 - Number of Conditional Arcs In Control Graph
- □ P0151 - Number of Control Statements
- □ P0152 - Number of Declarative Statements
- □ P0153 - Number of Accesses to Global Variables (R/W)
- □ P0154 - Bytes of Data Declared Locally
- □ P0155 - Bytes of Parameters for Procedure
- □ P0156 - Bytes of Parameters Passed On
- □ P0157 - Number of IPC Calls
- □ P0158 - Number of System Calls
- □ P0159 - Number of Calls to Non-Prototyped Functions
- □ P0160 - Number of READS FROM Global Variables
- □ P0161 - Number of WRITES TO Global Variables
- □ P0162 - Number of Non-Comment, Non-Blank Lines
- □ P0198 - Oman's Maintainability Index (MI)
- ☑ P0199 - Halstead Volume

Global Variable Occurrences By Kind (Procedure-Level)
- □ P0201 - Kind 1 (Compile-Time)
- □ P0202 - Kind 2 (Assignment / DR)
- □ P0203 - Kind 3 (ATOMIC)
- □ P0204 - Kind 4 (Assembler Code)
- □ P0205 - Kind 5 (Address-Of)
- □ P0206 - Kind 6 (Passed as Parm)
- □ P0207 - Kind 7 (Execution)
- □ P0208 - Kind 8 (Read / NDR)

```
SELECT P0129,P0199
 FROM  PIM
WHERE (Version LIKE '2.00.___')  AND (P0129 IS NOT NULL AND P0199 IS NOT NULL)
```

**Figure A.10 – `Grapher`'s Query tab**

Clicking on "Execute Query" and then "Retrieve and Plot Data" results in the screen shown in Figure A.11.

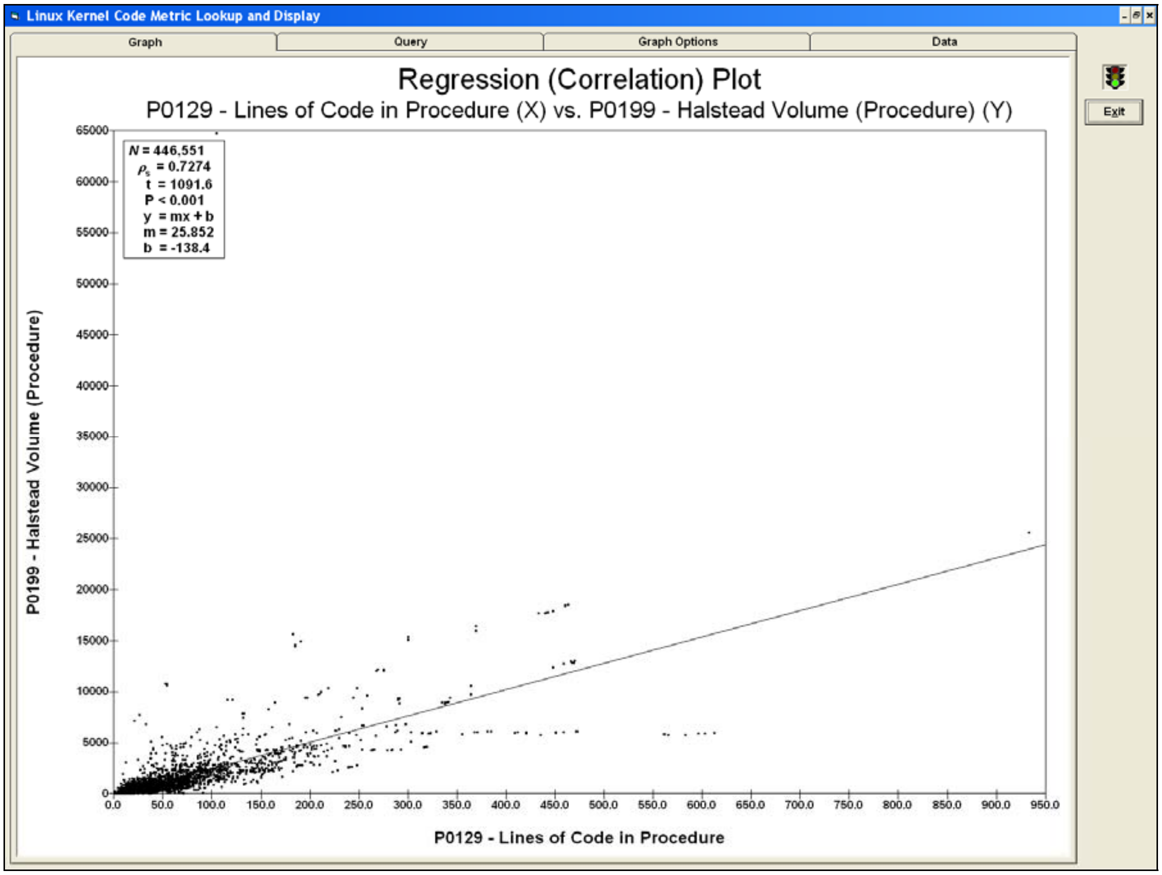


**Figure A.11 – Sample Correlation plot in `Grapher`**

Figure A.12 shows a sample histogram plot (FLM F0007 - Lines of Code in each file for all kernel versions). Figures A.13 and A.14 show line plots of total Lines of Code on a per-kernel basis, one plotted with the kernel version number as the X-axis, and the other with the release date as the X-axis. The implications of changing the independent variable from version number to release date are analyzed in depth in Chapter 8.

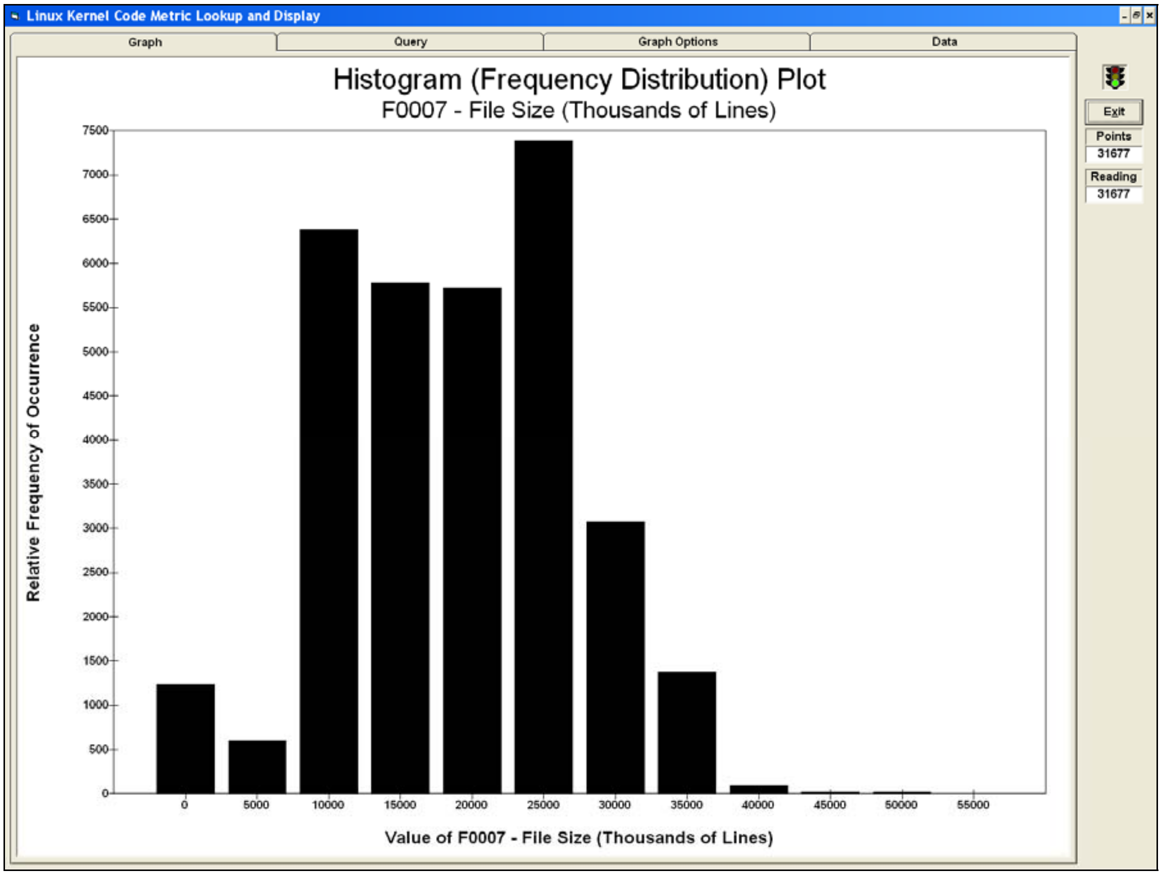

Linux Kernel Code Metric Lookup and Display
Graph
Query
Graph Options
Data
Histogram (Frequency Distribution) Plot
F0007 - File Size (Thousands of Lines)
Relative Frequency of Occurrence
7500
7000
6500
6000
5500
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000
55000
Value of F0007 - File Size (Thousands of Lines)
Exit
Points
31677
Reading
31677

**Figure A.12 – Sample Histogram plot in `Grapher`**

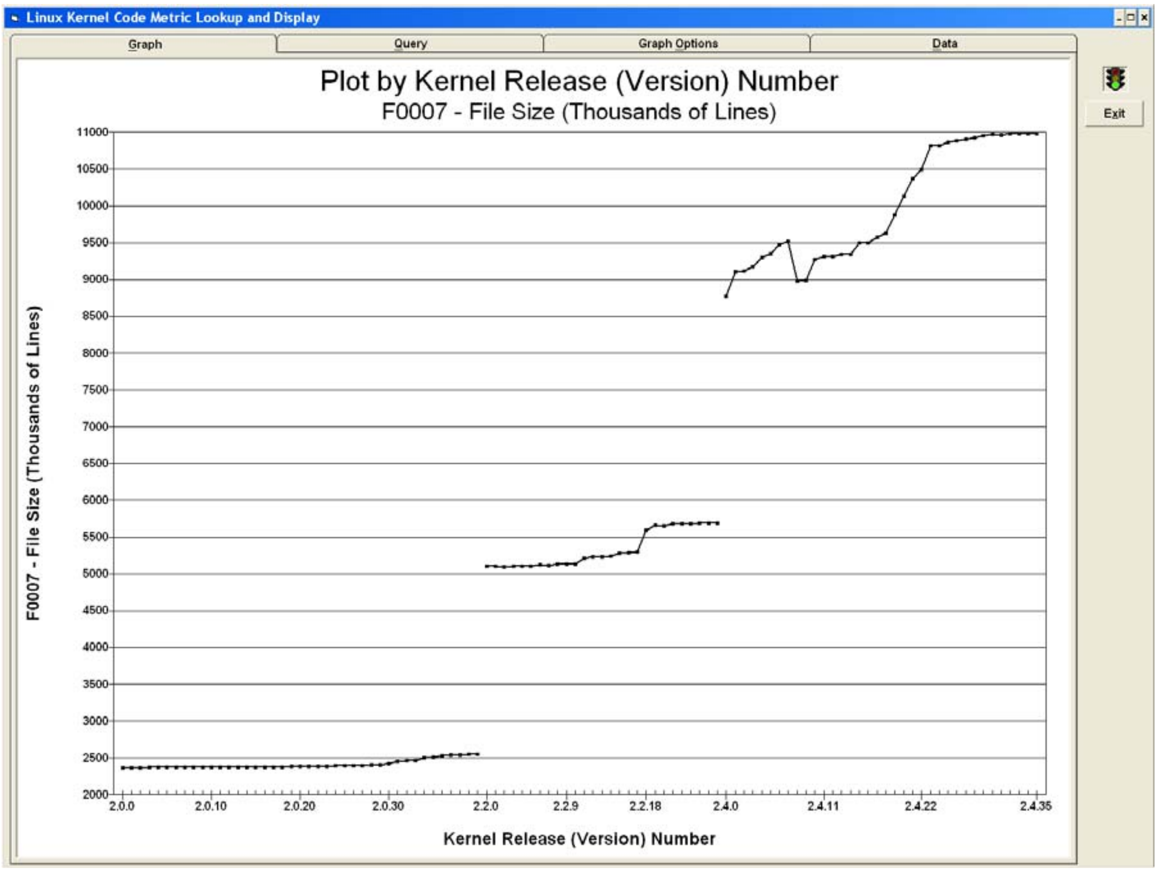

Linux Kernel Code Metric Lookup and Display
Graph
Query
Graph Options
Data
Exit
Plot by Kernel Release (Version) Number
F0007 - File Size (Thousands of Lines)
F0007 - File Size (Thousands of Lines)
11000
10500
10000
9500
9000
8500
8000
7500
7000
6500
6000
5500
5000
4500
4000
3500
3000
2500
2000
2.0.0
2.0.10
2.0.20
2.0.30
2.2.0
2.2.9
2.2.18
2.4.0
2.4.11
2.4.22
2.4.35
Kernel Release (Version) Number

**Figure A.13 – Sample plot by version number in `Grapher`**

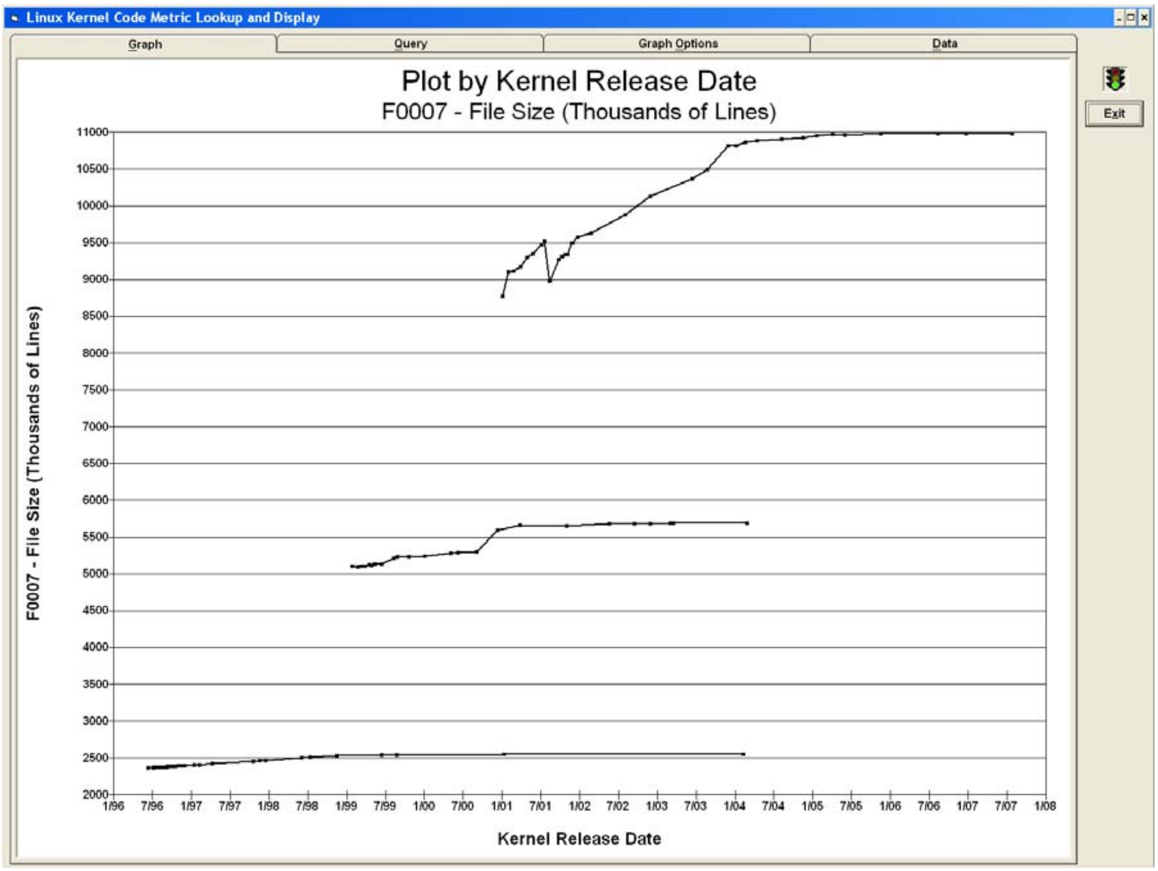


**Figure A.14 – Sample plot the same data as in Figure A.13, but plotted by release date in `Grapher`**

This concludes our account of the detailed workflow for analyzing the Linux source code, and our description of the many tools we wrote to analyze the data.

## The Final Workflow

Figures A.15, and A.16 show the entire workflow we developed to process the kernels and extract the source metrics

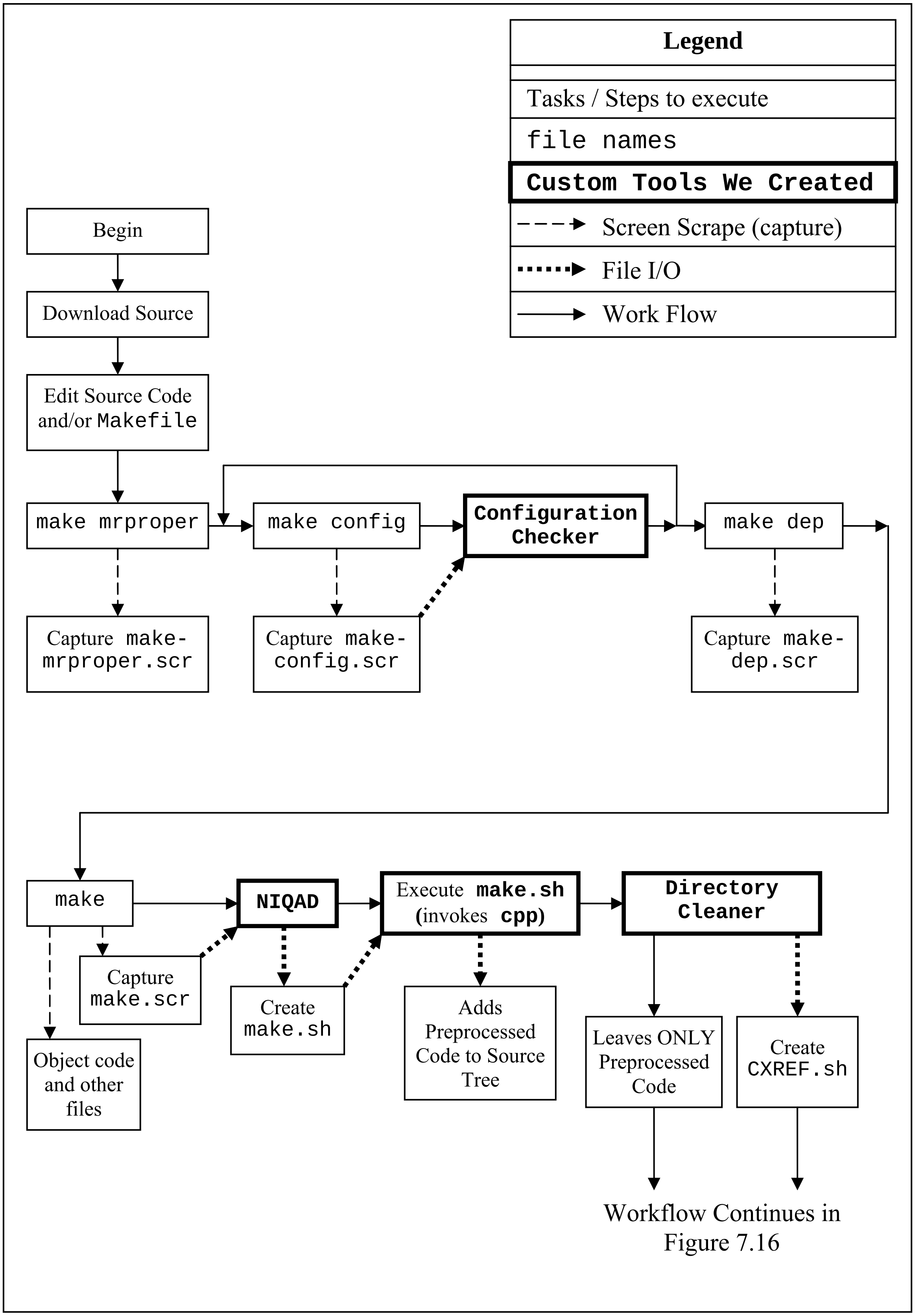


**Figure A.15 – Workflow, Part 1: Building the Kernel and Generating Raw Metrics**

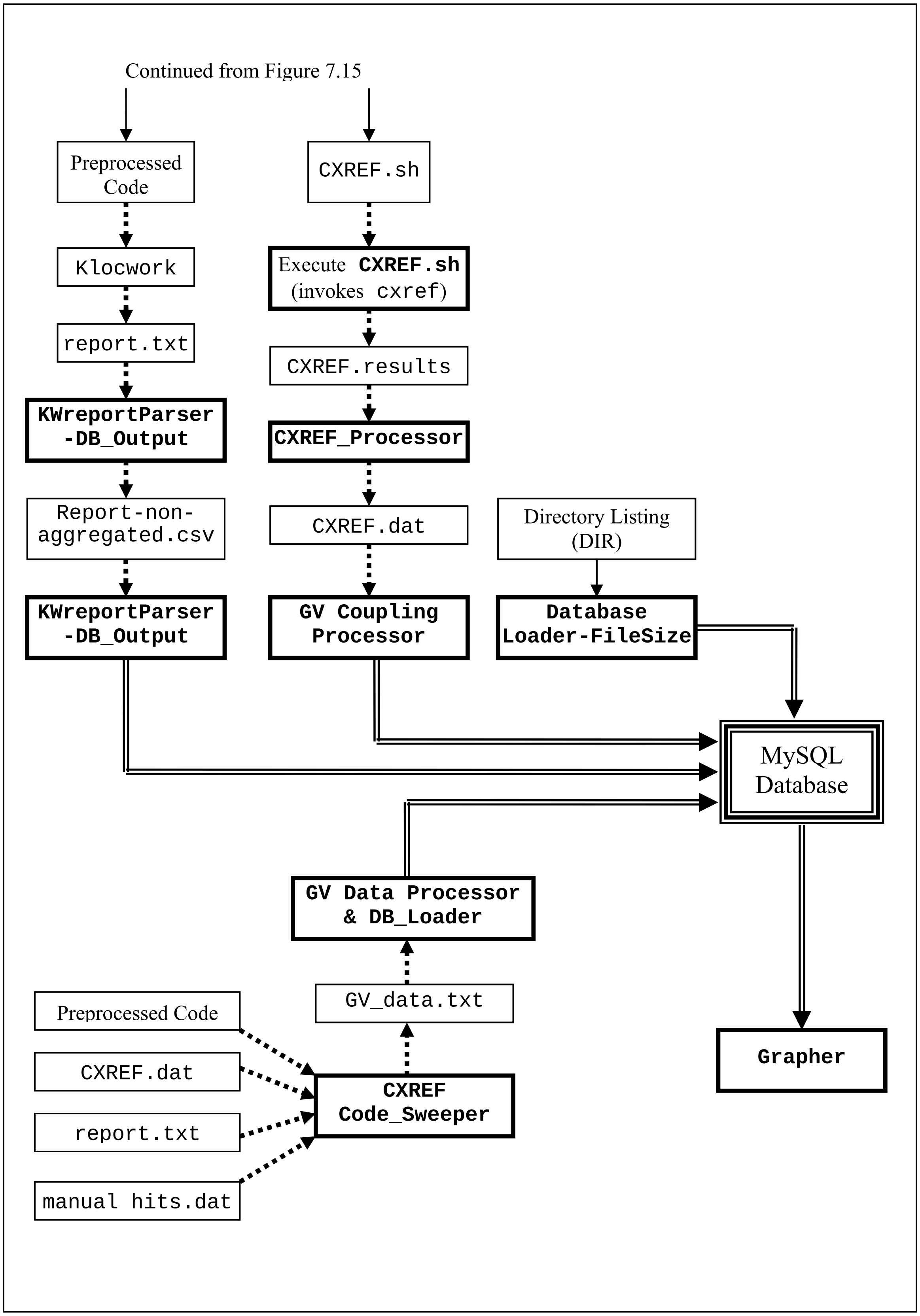


**Figure A.16 – Workflow, Part 2: Processing the Raw Metrics and Loading the Database**

## A.6 Lessons Learned from Previous Unsuccessful Attempts

### **A.5.1 Introduction**

The selection of VMWare and Debian `etch` was by no means the starting point in the selection of an environment under which to operate. This section walks through some of the unsuccessful first steps that ultimately led to the selection of the VMWare/Debian combination. The entire workflow required approximately seven months to develop, including the false starts described here.

To the best of our knowledge, none of the material presented in this section has been published either formally (in books, journals, or manuals) or on the Web. So, perhaps other researchers can benefit from our abortive attempts and not reinvent the same wheel. This dearth of available information regarding compilation of the kernel (and particularly the older ones), seems to be attributable to three primary reasons:

1) The older versions of the kernel require older versions of the build tools. A list of precisely which versions of each tool are required to build each version of the kernel (or which versions of each tool will not work) is not available. Even the documentation included with many of the kernel versions is incorrect, at least as related to the required tool versions. Without such tool information, compilation information is of little use.
2) The older tool versions (and/or feature sets within the tools) have been deprecated. In some cases, depending on the platform upon which the build is to be performed, the tools themselves must be compiled before they can be used, which leads to the

question of which versions of the old tools are required to build those old tools. There are virtually no readily available binary packages with sufficiently old versions of the necessary tools.

3) Users tend to install and run a prepackaged Linux distribution, meaning that the user generally never has to build a kernel. When the organization providing a distribution releases a new version of the kernel, the user has to do little beyond selecting the kernel version to which to upgrade, downloading a package, and initiating the upgrade. Because users typically never build kernels, there is generally no need to disseminate documentation on kernel building.

### A.5.2 Cygwin, Fedora Core, Knoppix, and Debian

Because we had Windows-based tools available, it seemed natural to start with Windows. The "path of least resistance" seemed to be to find a way to do everything under Windows.

According to the Cygwin website [www.cygwin.com]:

> "Cygwin is a Linux-like environment for Windows. It consists of two parts:
>
> - A DLL (`cygwin1.dll`), which acts as a Linux API emulation layer providing substantial Linux API functionality.
> - A collection of tools which provide the Linux look and feel."

The “collection of tools” includes text editors (`vi` and `emacs`), `make`, `gcc`, and seemingly all of the other tools required to build a Linux kernel.

From the standpoint of the tools themselves, Cygwin might be able to build the kernel. Unfortunately, the underlying file systems available under Windows (FAT32 and NTFS) are not case-sensitive. There are numerous places in the kernel where a source file with an extension of `.S` (assembly language source code) is preprocessed into a source file with an extension of `.s`, and then later assembled. Given the case-insensitivity of Windows, this clearly causes problems during the build process. There seemed to be no easy way to enable a case-sensitive feature in either file system, so it seemed that the next possibility was to get a driver that would allow Windows to use either ext2 or ext3 (native file systems for Linux). No such device driver seemed to be available. Consequently, it was time to abandon Windows as the host platform for the kernel builds.

A veritable plethora of Linux distributions exist, with a large number of versions available from a large number of distributors. As a starting point, we obtained and installed Fedora Core 5 [`fedora.redhat.com`], which was at that time the newest release.

Fedora Core 5 included version 4.11 of the gnu C compiler (`gcc`) [`gcc.gnu.org`]. The README file for kernel 1.0.0 says:

> "COMPILING the kernel:
>
> Make sure you have gcc-2.4.5 or newer available. It seems older gcc versions can have problems compiling newer versions of Linux. If you upgrade your compiler, remember to get the new binutils package too (for as/ld/nm and company)."

The README file specifically warns of trouble with the (at that time) *new kernel* and the *old tools*. Given that Fedora Core 5 (released in 2006) shipped with gcc 4.11, and kernel 1.0.0 (released in 1994), called for gcc-2.4.5 "or newer," it seemed that there should be no problems in compiling the *old kernel* with the *new tools*. Unfortunately, this was not the case. It is not possible to build the 1.0.0 kernel using gcc 4.11. Too many language features that were considered "mainstream" in 1994 have since been deprecated, and the newer compiler will not compile the code, and there are no documented options for gcc to enable "old behavior." Unlike a number of other tools, gcc is not backward-compatible, particularly when "backward" implies 12 years. Clearly, the only solution would be to find an older Linux environment, with older tools.

We happened to have an old copy of Knoppix [www.knoppix.com]. Knoppix is an interesting Linux distribution in that it can boot and run entirely from read-only media (CD-ROM or DVD). Knoppix uses a RAM drive for temporary storage (and will search for and use a Linux swap partition, if one exists on the system). Knoppix is based on the Debian core Linux distribution, with additions to enable the system to run from read-only media.

Booting the Knoppix 3.4 CD and searching the `/usr/bin` directory revealed that it included both `gcc` 3.3 as well as an "old" `gcc` of 2.95. If we could get `make` to use `gcc` 2.95, it seemed more likely that we could build the kernel, but this did not work, either. Although the Web page for `gcc` 2.95 [gcc.gnu.org/gcc-2.95] says that, "The whole suite has been extensively regression tested and package tested. It should be reliable and suitable for widespread use," a little farther down it also says, "finally, we can't in good conscience fail to mention some caveats to using GCC 2.95." Following this link [gcc.gnu.org/gcc-2.95/caveats.html] yields these disclaimers:

- `gcc` 2.95 will issue an error for invalid `asm` statements that had been silently accepted by earlier versions of the compiler. This is particularly noticeable when compiling older versions of the Linux kernel (2.0.xx). Please refer to the FAQ (as shipped with `gcc` 2.95) for more information on this issue.
- `gcc` 2.95 implements type based alias analysis to disambiguate memory references. Some programs, particularly the Linux kernel violate ANSI/ISO aliasing rules and therefore may not operate correctly when compiled with `gcc 2.95`. Please refer to the FAQ (as shipped with `gcc` 2.95) for more information on this issue.

The FAQ mentioned above can be found in the file [ftp://ftp.gnu.org/pub/gnu/gcc/gcc-2.95/gcc-core-2.95.tar.gz]. This file also clearly indicates why `gcc 2.95` will not work on the older kernels:

> "Building Linux kernels
>
> The Linux kernel violates certain aliasing rules specified in the ANSI/ISO standard. Starting with `gcc 2.95`, the `gcc` optimizer by default relies on these rules to produce more efficient code and thus will produce malfunctioning kernels. To work around this problem, the flag `-fno-strict-aliasing` must be added to the CFLAGS variable in the main kernel `Makefile`.
>
> If you try to build a 2.0.x kernel for Intel machines with any compiler other than `gcc 2.7.2`, then you are on your own. The 2.0.x kernels are to be built only with `gcc 2.7.2`. They use certain `asm` constructs which are incorrect, but (by accident) happen to work with `gcc 2.7.2`."

This warning is specific to only the 2.0.x kernels, and the 1.0.x kernels predate 2.0.x by another 2-4 years (depending on the specific versions in question). This FAQ did not give any indication as to what version of `gcc` would compile a 1.0.x kernel, but 2.7.2 certainly seemed to be a better option than any *subsequent* version.

Unfortunately, there was no `gcc` 2.7.2 package available for Fedora Core 5; nor was an older version of Knoppix available with `gcc` 2.7.2.

An extremely old Debian distribution (version 2.1, code named "slink") [`archive.debian.org/dists/Debian-2.1/`] was downloaded and tried, because it included `gcc` 2.7.2, but the tools available in such an old distribution were exceedingly primitive, and would have been difficult to work with.

Although it would have been possible to obtain the source code for `gcc` 2.7.2 and then *build* `gcc`, we did not want to go off on a tangent turning this project into one of tool-building if an acceptable off-the-shelf tool was available. If we could find the tool pre-built and ready-to-use, that would be infinitely preferable. If we could get that old tool to run in a more modern environment, so much the better.

Debian maintains lists of "packages." A Debian "package" is a binary download for a program (or group of programs), which includes all of the prerequisite information. As it happens, there is a Debian package for `gcc` 2.7.2 [packages.debian.org/stable/devel/gcc272] and the associated documentation [packages.debian.org/stable/devel/gcc272-docs]. Being able to install the Debian package for `gcc` 2.7.2 would mean that we could skip worrying about the prerequisites, setting the correct configuration, and any other build issues that would have been associated with building `gcc` from source code.

As it happens, Knoppix is built on Debian code, and Knoppix does include the Debian package manager utility, `dpkg`. After downloading the `gcc` 2.7.2 package, we booted from the Knoppix CD and attempted to run `dpkg` to install it, only to learn that `dpkg` will not run when Knoppix is booted from read-only media; it apparently does not install packages to the RAM drive.

This necessitated reformatting the Fedora Core 5 partition. A small swap partition was created, and the remainder of the drive was formatted as `ext3`, and Knoppix 3.4 was installed to it using `knx2hd`. After rebooting (to Knoppix from the hard drive, rather than from the CD-ROM), it was possible to install `gcc 2.7.2` using `gcc`. There was one missing prerequisite package, `gcc` (the C preprocessor). Downloading and installing this package allowed `gcc 2.7.2` to install without further incident. This system then had `gcc` versions `2.7.2`, `2.95`, and `3.3` all installed in `/usr/bin`.

Although Knoppix *would* boot from the hard drive, Knoppix was never intended to be installed and booted from the hard drive; rather it was a "live-boot" (boot from CD) operating system that could be used in the event of hardware failure, or to perform forensic analysis of systems without having to actually write anything to the hard drive. Knowing that Knoppix was based on the Debian distribution, we examined Debian. At that time, the "`etch`" version was still in testing (Debian's versions are named after characters in Pixar's animated film *Toy Story*), but had been "in testing" for approximately 18 months, suggesting that it had already seen extensive testing. As we would soon learn, it was only about 2 months from being christened as the latest stable release. The previous version (called "`sarge`") seemed a bit primitive, and considering the extended period that `etch` had been in testing, we elected to install `etch`.

It was just about this time that we learned that VMWare could be used to boot Linux in a window under Windows XP. We installed VMWare Workstation (5.5.3), created a VM with an 8 GB virtual disk and booted the VB from the Debian `etch` installation CD, and installed Debian Linux, and the required packages. The performance was quite acceptable, and we were able to copy files between the Linux and Windows environments with great ease.

We then cloned this 8 GB virtual disk to make another Linux VM, and then another. The three VMs were to be used to build the 2.0.x series, 2.2.x series, and 2.4.x series kernels, respectively. The sheer size of the 2.4.x kernels necessitated resizing the third VM's virtual disks (to 20 GB), which turned out to be tedious (but possible with some Web research); it should have simply been created larger from the outset.

Debian did not provide a version of `gcc` older than 2.7.2, so there did not seem to be any way of processing the kernels prior to 2.0.0.

Upon closer inspection of the dates of the old kernel releases (www.linuxhq.com), the 1.0.x kernel series consisted of 10 versions (1.0.0 – 1.0.9) released over the 35-day period of March 13 – April 16, 1994. The 1.1.x development cycle ran through 96 versions (1.1.0 – 1.1.95) over an 11-month period (April 6, 1994 – March 2, 1995).

Similarly, the 1.2.x production kernels saw only 14 releases over 5 months, between March 7, and August 2, 1995. The 1.3.x development cycle ran through 101 versions (1.3.0 – 1.3.100) over an eleven-month period from June 12, 1995 – May 10, 1996. The 2.0.0 kernel was released on June 9, 1996.

We elected to exclude the 1.0.x and 1.2.x kernels from our analysis for two primary reasons:

1) There were at least three fundamental architectural differences between 1.x and 2.x kernels that made comparing the 1.x and 2.x kernels an apples-and-oranges proposition:

    a. The 1.x kernels ran on only the i386 architecture. The underlying hardware interfaces changed considerably to support more architectures in the 2.x kernels.

    b. The KLM (Kernel Loadable Module) model was introduced in the 1.3.x series, and as such, was not present in either the 1.0.x or 1.2.x kernels.

    c. SMP (Symmetric Multi-Processing, or having multiple CPUs) was not available in the 1.x kernels.

2) Knowing that we would be examining the kernels with regard to growth over time, we had to consider the disproportionate influence that the old kernels would have on the time series. With a number of releases so closely timed and so long ago, we felt that the inclusion of the 1.x kernels would play havoc with the trend computations. With only 10 and 14 releases of 1.0.x and 1.2.x, respectively, there were not sufficient data points to analyze these series on their own.

### A.5.3 Klocwork

Despite Section A.3's orientation toward "lessons learned from things that failed," we still must include Klocwork in this section, not because it failed per se, but because it handled some things differently than we needed it to. Most specifically, Klocwork's documentation indicates that Klocwork defines a "global variable" as one "defined outside the function or method." The problem for us with this definition is that Klocwork classifies as *global* all module-level `static` variables defined outside of a function, including those that are *not* visible from any other file, and, as such, cannot participate in any instances of inter-file common coupling.

Normally, declaring a variable at the module level (i.e., outside of any function) would result in the creation of a global variable. However, because this declaration is preceded by `static`, the resulting identifier is visible to all functions within the same file, but a global variable is not created [ACM, 2007]:

> Normally, declaring a variable at file scope results in the creation of a global variable, which other translation units [files] can access by using an `extern` declaration. However, by preceding the declaration by the keyword `static`, the compiler restricts the use of that variable to the translation unit (file) in which it is declared. Furthermore, these variables do not conflict with variables in the global namespace or with statics in other translation units, even if they use the same name.

Klocwork's interpretation of the definition of "global variable" is certainly correct within the context of their framework, and indeed, if we were attempting to measure common coupling at the function level, such instances would have to be considered. However, measuring common coupling at the function level is beyond the scope of this dissertation, so we have used Klocwork for all metrics *except* those pertaining to global variables. For the global variable metrics, we have relied on CXREF.

# BIBLIOGRAPHY

[ACM, 2007] ACM Crossroads, http://www.acm.org/crossroads/xrds2-4/ovp.html. 2007

[Allen, Cocke, 1976] Allen, F.E. and Cocke, J., ”A Program Data Flow Analysis Procedure,” Communications of the ACM, Vol. 19, No. 3, pp. 137-147, 1976.

[Aoki et al., 2001] Aoki, A., Hayashi K., Kishida, K., Nakakoji, K., Nishinaka, Y., Reeves, B., Takashima, A., and Yamamoto, Y., “A Case Study of the Evolution of Jun: an Object-Oriented Open-Source 3D Multimedia Library,” Proceedings of the International Conference on Software Engineering (ICSE2001), IEEE Computer Society, Los Alamos, CA, pp. 524-533, May, 2001.

[Basili, Briand, Melo, 1996] Basili, V.R., Briand, L.C., and Melo, W.L., “A Validation of Object-Oriented Design Metrics as Quality Indicators,” IEEE Transactions on Software Engineering, Vol. 22, No. 10, pp. 751-761, 1996.

[Briand, Daly, Porter, Wust, 1998] Briand, L.C., Daly, J., Porter, V., and Wust, J. “A Comprehensive Empirical Validation of Design Measures for Object-Oriented Systems,” Proceedings of the Fifth International Software Metrics Symposium, 1998, pp. 246-257.

[Briand, Wust, Lounis, 1999] Briand, L.C., Wust, J., and Lounis, H., “Using Coupling Measurement for Impact Analysis in Object-Oriented Systems,” Proceedings of the IEEE International Conference on Software Maintenance, pp., 475-483, 1999

[Chou et al., 2001] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D., “An empirical study of operating systems errors,” ACM SIGOPS Operating Systems Review, ACM Press, New York, NY, USA, Vol. 35, No. 5, pp. 73-78, 2001.

[Coleman, Ash, Lowther, and Oman, 1994] Coleman, D., Ash, D., Lowther, B., and Oman, P., “Using Metrics to Evaluate Software System Maintainability,” Computer, Vol. 27, No. 8, pp. 44-49, 1994.

[CXREF, 2007] CXREF Web Site (http://www.dedanken.demon.co.uk/cxref), 2007

[Dempsey, Weiss, Jones, Greenberg, 2002] Dempsey, B., Weiss, D., Jones, P., and Greenberg, J., “Who Is an Open Source Software Developer?,” Communications of the ACM, Vol. 45, No. 2, pp. 67-72, 2002.

[Dinh-Trong and Bieman, 2005] Dinh-Trong, T.T., and Bieman, M., “The FreeBSD Project: A Replication Case Study of Open Source Development,” IEEE Transactions on Software Engineering, Vol. 31, No. 6, pp. 481-494, June 2005.

[Epping, Lott, 1994] Epping, A. and Lott, C.M., "Does Software Design Complexity Affect Maintenance Effort?," Proceedings of the Nineteenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt MD, pp. 291-313, 1994.

[Feitelson et al., 2007] Feitelson, D.G., Adeshiyan, T.O.S., Balasubramanian, D., Etison, Y., Madl, G., Osses, E., Singh, S., Suwanmongkol, K., Xie, M., and Schach, S.R., "Fine-Grain Analysis of Common Coupling and Its Application to a Linux Case Study," Journal of Systems and Software, Vol. 80, No. 8, pp. 1239-1255, 2007.

[Ferneley, 2000] Ferneley, E., "Coupling and Control Flow Measures in Practice," Journal of Systems and Software, Vol. 51, No. 2, pp. 99-109, 2000.

[Fitzsimmons, Love, 1978] Fitzsimmons, A., and Love, T., "A Review and Evaluation of Software Science," ACM Computer Surveys, Vol. 10, No. 1, March 1978

[gnu manifesto, 2007] The GNU Manifesto, http://www.gnu.org/gnu/manifesto.html, 2007

[Godfrey, Tu, 2000] Godfrey, M., and Tu, Q., "Evolution in Open-Source Software: A Case Study," Proceedings of the International Conference on Software Maintenance (ICSM-00), San Jose, California, October 2000.

[Grosser, Sahraoiu, Valchev, 2002] Grosser, D., Sahraoui, H.A., and Valtchev, P., "Predicting Software Stability Using Case-Based Reasoning," Proceedings of the 17th International Conference on Automated Software Engineering, pp. 295-298, 2002.

[Halstead, 1977] Halstead, M., "Elements of Software Science (Operating and Programming Systems Series)," Elsevier Science Inc., New York, NY, 1977

[IEEE, 1990] "IEEE Standard Glossary of Software Engineering Terminology," IEEE Standard 610.12-1990. Reaffirmed 2002 <http://standards.ieee.org/db/status/status.txt>, 1990, 2002.

[Ince, 1998] Ince, D., "Software Development: Fashioning the Baroque," Oxford University Press, 1988.

[Jin, 2001] Jin, B., Unpublished Data, Computer Science Department, Vanderbilt University, 2001.

[Jones, 1994] Jones, C., "Software Metrics: Good, Bad, and Missing," Computer, Vol. 27, No. 9, pp. 98-100, 1994.

[Klocwork Defects, 2006] "Detected Defects and Supported Metrics," Klocwork, Inc., Ottawa, Ontario, Canada, 2006.

[Lehman, Ramil, Sandler, 2001] Lehman, M.M., Ramil, J.F., and Sandler, U., "An Approach to Modeling Long-Term Growth Trends in Software Systems," Proceedings of the IEEE International Conference on Software Maintenance, pp. 219-228, 2001.

[MacCormack, 2006] MacCormack, A., Rusnak, J., and Baldwin, C., "Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code," Management Science, 2006.

[McCabe, 1976] McCabe, T.J, "A Complexity Measure," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976, pp 308-320.

[McCabe, 1989] McCabe, T.J, "Design Complexity Measurement and Testing," Communications of the ACM, December 1989, Vol. 32, No. 12, pp. 1415-1425.

[McCabe, Watson, 1994] McCabe, T.J. and Watson, A.H., "Software Complexity," Crosstalk, Journal of Defense Software Engineering, Vol. 7, No. 12, pp. 5-9, also available at http://www.stsc.hill.af.mil/crosstalk/1994/12/xt94d12b.asp

[Mockus, Fielding, Herbsleb, 2002] Mockus, A., Fielding, R., and Herbsleb, J, "Two Case Studies of Open Source Software Development: Apache and Mozilla," ACM Transactions on Software Engineering and Methodology, Vol. 11, No. 3, pp. 309-346, 2002.

[Myers, 1977] Myers, G.J., "An Extension to the Cyclomatic Measure of Program Complexity," ACM SIGPLAN Notices, Vol. 12, No. 10 (October 1977), pp. 61-64.

[Nakakoji et al., 2002] Nakakoji, K., Yamamoto, Y., Nishinaka, Y., Kishida, K., and Ye, Y., "Evolution Patterns of Open-Source Software Systems and Communities," Proceedings of the International Workshop on Principles of Software Evolution, ACM Press New York , NY USA, pp. 76-85, 2002.

[Oman, Hagemeister, 1994] Oman, P. and Hagemeister, J., "Construction and Testing of Polynomials Predicting Software Maintainability," The Journal of Systems and Software, Vol. 24, No. 3, pp. 251-266, September 1994.

[Paulson, Succi, Eberlein, 2004] Paulson, J.W., Succi, G., and Eberlein, A., "An Empirical Study of Open-Source and Closed-Source Software Products," IEEE Transactions on Software Engineering, Vol. 30, No. 4, pp. 246-256, 2004.

[Raymond, 2000] Raymond, E.S., "The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary," O'Reilly & Associates, Sebastopol, CA, 2000. Also available at http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/index.html

[Samoladas, Stamelos, Angelis, Oikonomou, 2004] Samoladas, I., Stamelos, I., Angelis, L., and Oikonomou, A., "Open Source Software Development Should Strive for Even Greater Code Maintainability," Communications of the ACM, Vol. 47, No. 10, October 2004, pp. 83-87

[Schach, 2007] Schach, S.R., "Object-Oriented & Classical Software Engineering," 7th Edition., McGraw-Hill, New York, NY, 2007.

[Schach et al., 2002] Schach, S.R. and Jin, B. and Wright, D.R., Heller, G.Z., and Offutt, A.J., "Maintainability of the Linux kernel," Software Engineering, IEE Proceedings, Vol. 149, No. 1, pp. 18-23, 2002.

[Schach et al., 2003] Schach, S.R, Jin, B., Wright, D.R., Heller, G., and Offutt, J., "Quality Impacts of Clandestine Coupling," Software Quality Journal, 11:211-218, 2003

[SEI Cyclomatic, 2000] Carnegie Mellon Software Engineering Institute, http://www.sei.cmu.edu/str/descriptions/cyclomatic.html, 2000.

[SEI Oman, 2002] Carnegie Mellon Software Engineering Institute, http://www.sei.cmu.edu/str/descriptions/mitmpm_body.html, 2002.

[Selby, Basili, 1991] Selby, R.W., and Basili, V.R., "Analyzing Error-Prone System Structure," IEEE Transactions on Software Engineering, Vol. 17, No. 2, pp. 141-152, 1991.

[Shepperd, 1988] Shepperd, M., "A Critique of Cyclomatic Complexity as a Software Metric," Software Engineering Journal, Vol. 3, No. 2, pp. 30-36, March 1988.

[Shepperd, 2000] Shepperd, M., "Dynamic Models of Maintenance Behaviour," Workshop on Empirical Studies of Software Maintenance, 2000.

[Shepperd, Ince, 1994] Shepperd, M., and Ince, D.C., "A Critique of Three Metrics," The Journal of Systems and Software, Vol. 26, No. 3, pp. 197-210, Eslevier Science, 1994.

[Tenenbaum, DeSilva, Langford, 2000] Tenenbaum, J., DeSilva, V., and Langford, J., "A Global Geometric Framework for Nonlinear Dimension Reduction," Science, Vol. 22, No. 12, December 2000.

[The Open Group, 2003] http://www.unix.org/what_is_unix/history_timeline.html, 2003

[Walsh, 1979] Walsh, T.J., "A Software Reliability Study Using a Complexity Measure," Proceedings of the AFIPS National Computer Conference, New York, AFIPS Press, pp. 761-768, 1979.

[Wang, Schach, Heller, 2001] Wang, S., Schach, S.R., and Heller, G.Z., "A Case Study In Repeated Maintenance," Journal of Software Maintenance and Evolution Research and Practice, Vol. 13, No. 2, pp. 127-141, 2001.

[Watson, McCabe, 1996] Watson, A.R., and McCabe, T.J., "Structured Testing: A Testing Methodology Using The Cyclomatic Complexity Metric," National Standards Institute of Standards and Technology, Computer Systems Laboratory, Special Publication #500-235, Gaithersburg, MD, September, 1996. Also available on-line at http://www.mccabe.com/iq_research_nist.htm

[Welker, 2001] Welker, K.D., "The Software Maintainability Index Revisited," Crosstalk, Journal of Defense Software Engineering, August 2001, pp. 18-21, also available at http://www.stsc.hill.af.mil/crosstalk/2001/08/welker.pdf

[Wikipedia Linux Kernel, 2007] http://en.wikipedia.org/wiki/Linux_kernel, 2007.

[Wikipedia Kernel Versions, 2007] http://en.wikipedia.org/wiki/Linux_kernel#Versions, 2007.

[Wikipedia Unix, 2007] http://en.wikipedia.org/wiki/Unix, 2007.

[Yu, Schach, Chen, Offutt, 2004] Yu, L., Schach, S.R., Chen, K., and Offutt, J., "Categorization of Common Coupling and its Application to the Maintainability of the Linux Kernel," IEEE Transactions on Software Engineering, Vol. 30, No. 10, pp. 694-706, 2004.