SUPPORTING SOFTWARE ENGINEERING VIA LIGHTWEIGHT FORWARD STATIC SLICING

A dissertation submitted

to Kent State University in partial

fulfillment of the requirements for the

degree of Doctor of Philosophy

by

Hakam W. Alomari

August, 2012

Dissertation written by

Hakam W. Alomari

Ph.D., Kent State University, 2012

M.S., Jordan University of Science and Technology, Jordan, 2006

B.S., Yarmouk University, Jordan, 2004

Approved by

Dr. Jonathan I. Maletic	Chair, Doctoral Dissertation Committee
Dr. Feodor F. Dragan	Members, Doctoral Dissertation Committee
Dr. Ruoming Jin	
Dr. Michael L. Collard	
Dr. John Portman	

Accepted by

Dr. Javed Khan	Chair, Department of Computer Science
Dr. Timothy Moerland	Dean, College of Arts and Sciences

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

DEDICATION

I dedicate this work to my family and many professors. A special feeling of appreciation to my loving parents, *Waleed* and *Nada* whose words of encouragement and push for tenacity ring in my ears. My wife *Dr. Hanan* has never left my side and is very special.

I also dedicate this dissertation to my brothers (*Dr. Mohammad* and *Dr. Abdullah*) and my sisters (*shoroq*, *Nour*, and *Dana*) who have supported me throughout the process. I will always appreciate all they have done.

I dedicate this work and give special thanks to my professors.

ACKNOWLEDGEMENTS

I thank my committee members who were more than generous with their expertise and precious time. A special thanks to *Dr. Jonathan Maletic*, my advisor and committee chairman for his countless hours of reflecting, reading, encouraging, and most of all patience throughout the entire process. I am deeply grateful for his great cooperation and support.

Thank you *Dr. Michael Collard* for your valuable contributions, and thank you *Dr. Feodor F. Dragan*, *Dr. Ruoming Jin*, and *Dr. John Portman* for agreeing to serve on my committee. As well, I express to *Dr. Eugene Gartland* my appreciation for willingness to serve as a graduate faculty representative.

I would like to acknowledge and thank my school division for allowing me to conduct my research and providing any assistance requested. Special thanks go to the graduate secretary *Marcy Curtiss* for here continued support.

Hakam W. Alomari

June 2012, Kent, Ohio

CHAPTER 1

INTRODUCTION

Program slicing is a commonly used approach for understanding and detecting the impact of changes to software. That is, given a variable and the location of that variable in a program, tell me what other parts of the program are affected by this variable. The approach has been used successfully for many years for various maintenance tasks [Gallagher, Lyle 1991; Tip 1995; Xu, Qian, Zhang, Wu, Chen 2005]. For example, slicing was used to help address the Y2K problem by identifying parts of a program that could be impacted by changes on date fields.

The concept of program slicing was originally identified by Weiser [Weiser 1979; Weiser 1981] as a debugging aid. He defined the slice as an executable program that preserved the behavior of the original program. Weiser's algorithm traces the data and control dependencies by solving data-flow equations for determining the direct and indirect relevant variables and statements. Since that time a number of different slicing techniques and tools have been proposed and implemented. These techniques are broadly distinguished according to the type of slices such as: Static vs. Dynamic [Tip 1995; Xu, Qian, Zhang, Wu, Chen 2005], Closure vs. Executable [Xu, Qian, Zhang, Wu, Chen 2005], Inter-procedural vs. Intra-procedural [Horwitz, Reps, Binkley 1988; Gallagher, Binkley 2008], and Forward vs. Backward [Kumar, Horwitz 2002; Xu, Qian, Zhang, Wu, Chen 2005].

Previous work on the detection of program slices has most often been based on the notion of a Program Dependence Graph (PDG) [Ottenstein, Ottenstein 1984] or one of its variants, e.g., a System Dependence Graph (SDG) [Liang, Harrold 1998]. Unfortunately, all these approaches typically suffer from scalability and computational issues due to the fact that building the PDG is complicated in terms of time, space, and data related operations. While there are some (costly) workarounds, generating slices for a very large system can often take days of computing time. Additionally, many tools are strictly limited to an upper bound on the size of the program they can slice.

While the lengthy time constraint is reasonable for deep analysis of systems it limits the use of program slicing to only the very critical situations. Using slicing for quick project planning and management decisions is effectively unpractical. For example, if a project team needed to assess the inclusion of a new feature in the next version release of a system, program slicing would be a great tool to predict what other modules would be impacted. These impacted modules would of course need to be thoroughly tested. If modules with a high cost to test were to be impacted the risk to include that new feature would greatly influence its inclusion in the next release. It would be very beneficial to have some results of program slicing in minutes to help with this type of decision making.

The work presented here addresses this limitation by eliminating the time and effort needed to build the entire PDG. Instead the dependence information is computed as needed (on-the-fly) while computing the slice for each variable in the program. Lightweight static analysis techniques are used in computing dependencies. As such, the increase in speed is countered by a possible decrease in accuracy. The slicing process is performed using the srcML [Collard, Decker, Maletic 2011] format for source code. Source code is first converted to srcML and then a stream-oriented approach to compute the slice is performed. The srcML format provides direct access to abstract syntactic information to support static analysis. The entire process is very efficient and scalable. For example, it takes approximately 20 minutes (using a desktop machine) to convert the Linux kernel (~14 MLOC) into srcML and compute the slice for every variable in the entire system.

We feel that our slightly less accurate but highly scalable approach for slicing will help enable researchers to more easily investigate large systems and even the entire history of a system in the context of program slicing. Moreover, practitioners will have a very practical way to estimate the impact of a change to a large within minutes (or hours) instead of days. This would be very useful to determine if a more accurate analysis of the change is necessary and cost effective.

Because of the potentially complex and nature of heavyweight program slicing, new lightweight approaches and tools must be developed to support maintainers working in this domain, and an efficient slicing approach is needed to address many unpractical problems.

1.1 Research Overview

In view of the fact that existing slicing approaches are inadequate for the purpose of maintaining large-scale systems, we believe that the work in this dissertation will be important in the future as systems grow and hence the maintenance becomes more complicated and costly.

The objective of this dissertation is twofold. First, develop a very-efficient and highly-scalable lightweight slicing approach and tool. This approach will compute program slices in a drastically more efficient (time-wise) manner. The second objective, due to the efficient nature of the approach it will be used to address a number of applications and problems that, in practice cannot be (or are extremely costly) to addressed with current heavyweight slicing approaches. The dissertation addresses the following specific problems in novel ways.

1. Demonstrate the scalability of the proposed slicing approach for large-scale systems. For example, all the slices for 17 years of versions of the Linux kernel (over 900 versions) will be performed. The slicing tool takes a lightweight approach; that is the main reason that the analysis over all these versions was even possible. Since a partial parsing of the source code is done (i.e., those source statements are in interest) and the time and space required is small and scalable. Given that, we can overcome the shortcomings of using traditional intermediate program representation models by saving the wasted time and space from building the above models.
2. Investigate how slices change over the history of software system. This aims at introducing a slice-based software metrics over versions history that reflects maintenance effort. These metrics are extracted directly from the source code without any other metadata needed, e.g., person hours, passage of time, etc. This

work is the first to uncover the maintenance changes using slicing over a large amount of data.

3. Develop an approach to estimate the maintenance effort for open-source systems using the new slice-based metrics. Three different granularities of slice sizes are analyzed (i.e., slice size is computed as the total number of line, functions, or files in the slices). Changes to the system are then modeled using the difference between the slice sizes of two versions. To the best of our knowledge, this is the first work applying a slice-based metric to build an estimation approach for maintenance effort in open-source systems. We consider the hypothesis that the historical source code changes can be used to regulate effort estimation approaches with a high sensible degree of predictive power. Existing effort estimation models are inadequate for the purpose of open-source systems, thus the need to develop new models is crucial. In this dissertation we accomplished a slice-based indirect effort-estimation model for open-source systems.

4. Characterizing the type of maintenance activities (i.e., adaptive, corrective, perfective, and preventative) being performed during the lifetime of a large software system (e.g., Linux kernel). The objective is to provide empirical support for *Lehman's laws* of software evolution [Lehman 1980; Lehman 1996; Lehman, Ramil, Wernick, Perry, Turski 1997; Lehman, Ramil 2003]. Lehman proposed a number of laws identified initially as a series of observations or behaviors formulated starting in 1974 in the evolution of software. Lehman's laws of software evolution explain the forces that driving new developments on the

software on one hand, and the forces that slow down the development progress on the other hand. Although, these laws are believed to observe all the changes during the software evolution process, however, some empirical observations of studying the development of open-source system appear to challenge some of Lehman's laws [Godfrey, Qiang 2000], since the laws are believed to apply mainly to strictly managed and closed-source code systems [Israeli, Feitelson 2010].

1.2 Contributions

The primary contribution presented in this dissertation is:

1. The development of a very-efficient and highly-scalable program slicing approach and tool to automatically produce slices for all variables in a large-scale system.

The ability to apply slicing to large systems opens up new avenues of research. The following program slicing applications are considered important in the software development process and maintenance. Unfortunately, no special algorithm in the literature has been introduced to serve these applications. This introduces the need for slicing algorithms that compute all the required slices in a more efficient way.

Specific research contributions include:

2. Introducing a slice-based software metrics over versions history that reflects maintenance effort.
3. A lightweight approach to estimate maintenance effort for open-source systems using the new slice-based metrics.
4. A characterization of software maintenance activities using Linux kernel, based on calculation of different slice-based metrics.

5. A characterization of system’s evolution using Lehman’s laws as a basis.

The first research contribution (CHAPTER 3) has been addressed and the results are submitted to the 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM’12) [Alomari, Collard, Maletic 2012]. The second and third research contributions (CHAPTER 5) are detailed and submitted to the 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM’12) [Alomari, Collard, Maletic 2012]. The fourth and fifth research contributions (CHAPTER 4) are addressed and the result is written up for submission to the 25th IEEE International Conference on Software Engineering (ICSE’13).

1.3 Organization

Background reading is presented in CHAPTER 2; this chapter gives an overview of program slicing. CHAPTER 3 details the lightweight forward static slicing approach. In addition, we empirically demonstrate the scalability of the approach over the GNU Linux kernel. CHAPTER 4 shows how slices change over incremental source code changes. It also introduces and validates a slice-based metrics used to provide a characterization of software maintenance activities using Linux kernel. CHAPTER 5 presents a novel slice-based approach to estimate maintenance effort in open-source systems. This work is a direct application for identifying slice-based metrics for large-scale systems. Conclusions and directions of future work are presented in CHAPTER 6.

APPENDIX A includes the XML translation information. APPENDIX B includes slice intersection comparison. APPENDIX C includes a listing of Linux kernel versions with their slices. APPENDIX D includes the source code scripts used to obtain the slice information of the *CodeSurfer* tool.

Please note that each chapter contains its own related work section.

CHAPTER 2

BACKGROUND MATERIAL ON PROGRAM SLICING

In this chapter we discuss the basic concepts and terminology associated to our work and that are used in later chapters.

2.1 Program Slicing

In computer programming, program slicing is the computation of the set of programs statements, the *program slice*, which may affect or affected by the values at some point of interest, referred to as a *slicing criterion*.

Mark Weiser [Weiser 1979; Weiser 1981] was the first to propose the idea of program slicing. In short, his idea is based on eliminating any code statement not affecting the values computed at a specified point in the program. Based on data flow and control flow dependences, slices are formed by computing consecutive sets of relevant statements.

Based on the original definition of Weiser, informally, a static program slice S consists of all statements in program P that may affect the value of variable v at some point p. The slice is defined for a slicing criterion $SC = (x, V)$, where x is a statement within program P, and V is a subset of variables in P. A backward static slice includes all the statements that affect variable v for a set of all possible inputs at the point of interest (i.e., at the statement x). Static slices are computed by finding consecutive sets of indirectly relevant statements, according to data and control dependencies.

For example, in Figure 2.1, the statements in the backward slice of the output statement *write* (*sum*) (in line 8) are highlighted. This new program is a valid slicing of the program with respect to the criterion *SC* = (*write* (*sum*), {*sum*}). The value of variable *sum* impacted by lines 1, 2, 4, 5, and 7 in the example code.

```
1. int i;

2. int sum = 0;

3. int product = 1;

4. for(i = 0; i < N; ++i) {

5. sum = sum + i;

6. product = product *i;

7. }

8. write(sum);

9. write(product);
```

Figure 2.1. The original definition of backward slice of `write(sum)` as proposed by Weiser [Weiser 1981].

Program slicing has motivated a large body of work for different applications in software engineering, and has been found to be of use in many aspects of the software development life cycle, including testing, debugging, re-engineering, comprehension, maintenance, and measurement [Gallagher, Lyle 1991; Tip 1995; Ishio, Kusumoto, Inoue 2003; Xu, Qian, Zhang, Wu, Chen 2005; Feng, Maletic 2006; Pan, Kim, Whitehead

2006]. These applications require different properties of slices, thus a number of different definitions to program slicing have been proposed after Weiser's. The main distinctions between these definitions are as follows:

- *Static or dynamic slicing* [Tip 1995; Xu, Qian, Zhang, Wu, Chen 2005]. Static slicing makes no assumptions regarding the input of the program. The static slice includes all statements that potentially affect/affected by the value of a variable at a particular point of interest in the program. This captures all possible executions of the program.

  On the contrary, dynamic slicing focuses on a particular input for a program, and contains all statements that affect/affected by the value of a variable at a specific point in the program assuming a given, fixed input. This approach gives a better understanding of programs and their executions for a particular input [Feng, Maletic 2006; Zhang, Gupta, Gupta 2007].

- *Direction of program traversal*. Program slicing can be either backward or forward [Kumar, Horwitz 2002; Xu, Qian, Zhang, Wu, Chen 2005]. A forward slice contains all those statements in the program which are affected by the value of the slicing variable at a given point in the program (i.e., statements affected by changing the value of the slicing variable), and removes all statements which are not affected by the slicing criterion.

  Forward slicing provides support for increasing and improving the understanding of large programs, source code debugging, code certification and inspection, and program differencing and specialization [Bent, Atkinson, Griswold 2000; Collard

2003]. In contrast, a backward slice contains all those statement in the program that may affect the value of variable, i.e., those statements that have some effect on the slicing criterion. Although backward slicing often provides more information than is needed for many program understanding situations, it can be of use in bug location [Tip 1995; Gallagher, O'Brien 2001].

There are variations from these definitions. Bergeretti et al [Bergeretti, Carre' 1985] proposed the notion of a forward static slice, where slices can be defined in terms of information flow relations that can be observed from a program in a syntax directed fashion. Horwitz [Horwitz, Reps, Binkley 1988] defines a forward slice as a set of statements affected by the value of the variable in the slicing criterion, rather than an executable subprogram. Two definitions of forward slicing are introduced and related back to traditional definitions of forward and backward slices.

- *Whether the slice is executable* [Xu, Qian, Zhang, Wu, Chen 2005]. An executable slice means that the slice can be compiled and run. Non executable slices do not constitute an executable program, and form a subset of statements of the program that affect/affected by the value of the slicing variable. The requirement that a slice be an executable may be too restrictive when the slice is to be used to comprehend a program or in the impact analysis. In those two cases, Kumar et al [Kumar, Horwitz 2002] mentioned that it might be more appropriate the slice to be a sub-statements of the program's statements, rather than an executable.
- *Scope of the slice* [Horwitz, Reps, Binkley 1988; Binkley, Gallagher 1996]. The slice can also be characterized in how it handles slicing across procedure boundaries called

inter-procedural slicing, or locally, called intra-procedural slicing. In [Weiser 1984] Weiser introduced inter-procedural program slicing, and extended his previous intra-procedural work proposed in [Weiser 1981]. In that work he determines the effect of call statements on the set of relevant statements. To do so the slicing criterion is extended with those argument variables. All of the types of slices are presented by Venkatesh [Venkatesh 1991] in terms of semantic approaches.

In contrast to any differences, the common factor between these diverse definitions is that they are based on the notion of a PDG, or one of its variants. The definition introduced by Ottenstein et al [Ottenstein, Ottenstein 1984] redefined static slicing in terms of a reachability problem on a PDG based on data flow and control flow dependencies. That is, all of these approaches depend on the calculation of a fully-computed PDG. For a large program the resulting PDG can be quite large.

The approach presented in this dissertation does not depend on the creation of a fully-computed PDG. Instead the required dependence information is retrieved as needed while computing the slice for each variable in the program. This will be explained in more detail in the next chapter.

2.2 Data-flow and Control-flow Dependencies

Simply, a data or flow dependence means that the calculations performed at one statement directly depend on the calculation from other statement. For example, as shown in Figure 2.2, consider statements *S1* and *S2* we say that data dependence exists from *S1* to *S2* via variable *A*, since its value is used in calculation *S2*.

```
S1.A = B * C

S2.D = A + E
```

Figure 2.2. Data dependence of statements S1 and S2 via variable A, (variable D depends at variable A value from S1).

A control dependence implies that the computations performed at one statement are directly depending on the result of a conditional predicate which determines whether the statement is executed. This can be seen in Figure 2.3 where statement *S1* is a conditional predicate whose result determines whether statement *S2* is executed.

```
S1.if (A) then

S2.B = C + D
```

Figure 2.3. Control dependence of statement S2 on the evaluation of the condition in S1, (execution of S2 depends at S1).

Based on the scope of the slice, there are variations of these definitions. The data and control dependencies could be either inter-procedural or intra-procedural. That is, the inter-procedural and intra-procedural dependencies are defined as follows:

- An intra-procedural data-dependence relation between two points exists if the first point may assign a value to a variable that may be used by the second point.
- An intra-procedural control-dependence relation between two points exists if the first point is a conditional predicate, and the execution at the second point directly depends on the result of the first point.

- In addition, there is an inter-procedural data-dependence relation between each function call argument and the corresponding parameter.
- Finally, there is an inter-procedural control-dependence relation from each call point of a function to its signature.

In the computation of a slice, these dependencies information is required and should be considered in order to construct the program slice.

2.3 Program Dependence Graph

The PDG graph for the program is a directed graph whose vertices represent the assignment statements and control predicates. The vertices are connected by two types of intra-procedural edges that represent either a control dependence or data dependence.

In more details, each program's simple statement and control predicate is represented by a node. Simple statements comprise assignment, write, and read statements. Compound statements comprise loop and conditional statements and they are represented by more than one node. The edges in a PDG are two types: data dependence edges and control dependence edges.

A data dependence edge between two nodes implies that the computation performed at the destination node of the edge directly depends on the value computed at the source node of edge. That is, the source node has the definition of the variable used in the destination node. This definition of data dependence ensures that each use of a variable is reached by exactly one definition, i.e., no in between any other definition of the variable. A control dependence edge between two nodes implies that the result of the predicate expression at the source node determines whether to execute the destination node.

```
1. void main( ) {
2.       cin>> n;
3.       int i = 1;
4.       int sum = 0;
5.       while(i <= n) {
6.               sum = sum + i;
7.               ++ i; }
8.       cout<< sum; }
```

Figure 2.4. Intra-procedural Data and Control dependencies source code example.

The SDG graph is a directed graph consisting of interconnected PDGs (one per procedure) by the inter-procedural control and data dependence edges. That is, control edges connect procedure call sites to the entry points of the called procedure, and data edges represent the flow of data between actual parameters and formal parameters (and return values). In SDG there are several types of vertices and edges that do not found in PDG [Tip 1995; Binkley, Harman 2004] (e.g., call-site vertex, actual-in vertex, formal-in vertex, parameter-in edge, etc.).

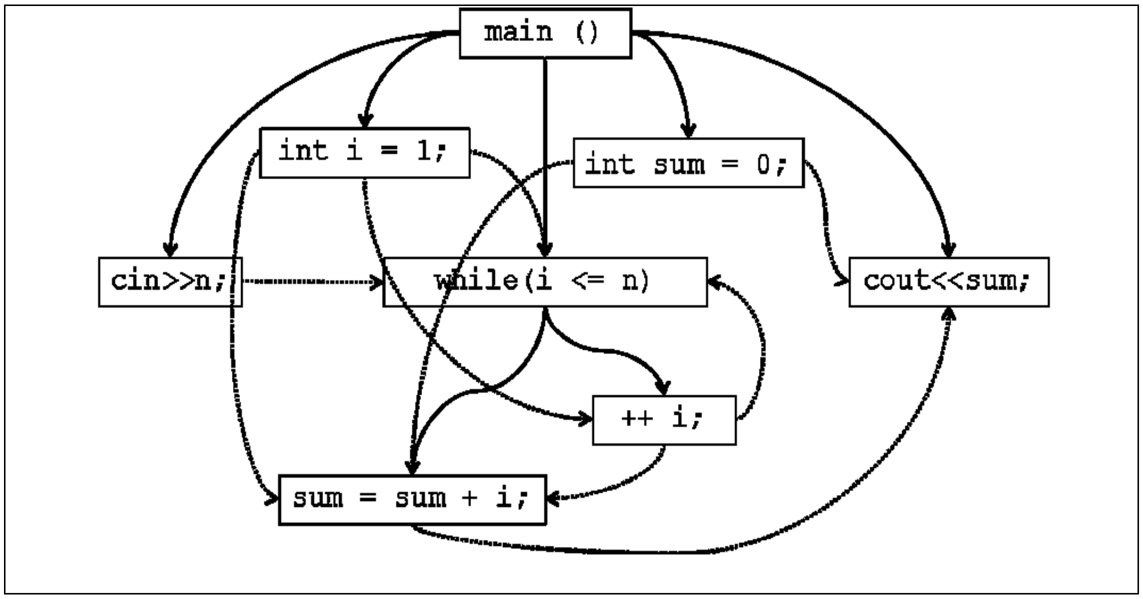


Figure 2.5. The PDG of the sample source-code in Figure 2.4.

Figure 2.5 shows a simple PDG graph for the example program of Figure 2.4. In this example, each node of the PDG represents one statement of the program (e.g., in this particular example the eight statements are represented by eight nodes). The bold directed edges correspond to intra-procedural control dependence, and the dashed directed edges represent intra-procedural data dependence.

By using the PDG in the slicing process, these dependencies are calculated in the preprocessing phase before the slicing process is started. Such as, the slice is constructed by traversing the considered data and control edges to retrieve all reachable nodes from the slicing node.

CHAPTER 3

LIGHTWEIGHT FORWARD STATIC SLICING

The work presented in this chapter introduces our lightweight slicing method and tool. The method leverages an XML representation of C source code called srcML. The program dependence graph is not computed for the entire program but instead the dependence information is computed as needed while computing the slice on a variable (in Section 3.4). The result is a list of line numbers, dependent variables, aliases, and function calls that are part of the slice for a given variable (in Section 3.2). The tool produces the slice in this manner for all variables in a given system. The tool is highly scalable and can generate the slices for all variables of the Linux kernel in less than 15 minutes. Benchmarks results are compared with the CodeSurfer slicing tool (in Section 3.5).

Forward static program slicing [Bergeretti, Carre' 1985] refers to the computation of program points that are affected by other program points. The forward slice from program point *p* includes all the program points in the forward control flow affected by the computation at *p*. Program points (or variables) are the most basic fragments of the source code. A program may contain multiple files, a file may contain multiple functions, a function may contain multiple lines, and a line may contain multiple variables. In this work, for convenience, we report the impact at the variable granularity.

A forward, static, non-executable, inter-procedural approach to program slicing is taken to build our approach to lightweight slicing. The approach varies from the traditional definitions in two ways.

1. A PDG is not computed for the entire program,
2. The slicing criterion does not require a precise reference to a location in the source.

The approach relies on an underlying XML representation of the source code, namely srcML [Maletic, Collard, Marcus 2002; Collard, Kagdi, Maletic 2003; Collard, Decker, Maletic 2011]. srcML augments source code with abstract syntactic information. This syntactic information is used to identify program dependencies as needed when computing the slice.

srcML (SouRce-Code Markup Language) is an XML format used to augment source code with syntactic information from the AST to add explicit structure to program source code. Of interest for this particular task, srcML takes an unprocessed view of the source code, i.e., before the c-preproccesor is run, and provides a relatively compact representation of the program. The srcML format is supported with a toolkit, *src2srcml* and *srcml2src*, that supports conversion between source code and the format. Multiple languages, including C, C++, and Java, are supported. Once in the srcML format, standard XML tools can be used for analysis. This format has been previously used for lightweight fact extraction[Collard, Kagdi, Maletic 2003; Collard, Decker, Maletic 2011], source-code transformation [Collard, Maletic, Robinson 2010; Collard, Decker, Maletic 2011], and pattern matching of complex code [Dragan, Collard, Maletic 2006].

Before presenting the lightweight static forward algorithms, we define our slicing criterion, and then show how a slice is computed using this approach.

3.1 Extended Slicing Criterion

We define our slicing criterion to consist of a file name, a function name, and a variable name. This slicing criterion is the triple (*f, m, v*) where *f* is a file in the system, *m* is a function/method in the file *f*, and *v* is a variable in the given function *m*. This definition of a slicing criterion does not require a precise reference to a statement number. This concept of slicing is used by Gallagher et al [Gallagher, Lyle 1991; Gallagher 2004] and is referred to as a *decomposition slice*. The definition includes all relevant computations involving a given slicing variable.

A decomposition slice can be viewed as a union of a collection of slices taken at individual statements on the given variable [Gallagher, Lyle 1991]. The example used by Gallagher does static backward slicing only. The decomposition slice on variable *v* is the union of backward slices taken at a set of statements that output variable *v* in addition to the slice taken at the last statement in the program. The last statement is included so that a variable which is not part of the program output may still be used as a decomposition criterion.

For example, from Figure 3.1 If we perform two backward slices for the program using the criterion (c, s_4) and the criterion (c, s_6), i.e., both statements that output value of the variable *c*, the resulting slices are the statements $\{s_1, s_2, s_3, s_4\}$ and $\{s_1, s_2, s_5, s_6\}$ respectively. The slicing criterion using line s_6 is not sufficient to capture all computations over the slicing variable *c*, given that statement s_3 is not retrieved in the slice, even though it alters the value of the slicing variable.

For our static forward decomposition slicing we modify Gallagher's definition by slicing at the set of statements that assign (or input) to the given variable *v*. This choice is motivated by the example given in Figure 3.1. If we perform two forward slices of the variable *c* in this program starting at statements s_3 and s_5, i.e., both statements that assign (redefine) a value to the variable *c*, then the resulting slices include the statements $\{s_3, s_4\}$ and $\{s_5, s_6\}$ respectively, that is, statements impacted by the value of variable *c*. Slicing from the assignment statement in s_3 is not sufficient to capture all the impacted statements by the value of *c*, given that statement s_6 is not retrieved in the slice, because the value of *c* assigned in statement s_3 can never reach the use of *c* on statement s_6, as there is an assignment that redefines *c* in statement s_5. Therefore, the decomposition slice obtained by a forward slicing algorithm for the example in Figure 3.1 using the variable *c* is equal to *slice* $(c, s_3) \cup$ *slice* (c, s_5).

```
s1.cin >> a;

s2.cin >> b;

s3.c = a + b;

s4.cout << c;

s5.c = a - b;

s6.cout << c;
```

Figure 3.1. Slicing motivation proposed by Gallagher [Gallagher, Lyle 1991].

From the point of view of data-flow analysis the decomposition slice could be either *backward-based* or *forward-based*. That is, the *backward-based* decomposition slice is computed iteratively by propagating information from the outputs of variables to their inputs, and from inputs to outputs in the case of *forward-based* decomposition slice.

The quality of the decomposition slice is affected by the quality of the slice, since as shown the definition of decomposition slice is independent of any underlying slicing technique. Once a slice is obtained using any slicing algorithm, a decomposition slice may be computed [Gallagher, Lyle 1991; Gallagher 2004].

3.2 Slice Profile and System Dictionary Construction

In the computation of a slice, certain dependence information is required. Unlike other slicing techniques, our algorithm does not rely fully on pre-computed data and control dependencies since they can require costly analysis, e.g., constructing the *def-use* chains in the existence of pointers. Instead, this is calculated as needed on the fly for the slicing variable while constructing the slice.

The approach computes a *slice profile* that contains all the relevant statements, from all possible slices, over a given slicing variable *v*. After the algorithm is applied, the slice profile associated with a variable *v* consists of the lines of code transitively affected by the value of *v* along control and data dependencies.

By modifying the slicing criterion to be (*f, m*), our approach can retrieve the slices for all the variables inside a given function. Moreover, the slicing criterion (*f*) can be used to find all the slices of all variables in all functions in a given file. A *system dictionary* can be built, referred to as (*F, M, V*), and includes all files in the system, all functions in each file, all variables in each function, and all the global variables in the system. Each entry of the system dictionary is a single slice profile with the following structure:

`file_name / function_name / variable_name / @index, slines, cfunctions, dvariables, pointers.`

The field *@index* is the index of the slicing variable as declared inside the function, *slines* is a finite set of the slicing statement numbers, *cfunctions* is a finite set of the called functions with the slicing variable, *dvariables* is a finite set of dependent variables affected by the value of the slicing variable, and finally *pointers* which is a finite set of the pointer aliases of the slicing variable.

For example, Figure 3.2 (B) shows the relevant slice profiles for slices taken with respect to defined variables in the source code in Figure 3.2 (A). The slice profiles are computed from Line 1 to Line 11. For example, the slice profile of variable z = (*slines* (z) ∪ *cfunctions* (z) ∪ *dvariables* (z) ∪ *pointers* (z))*;* thus, the slice profile includes the transitive closure of the data and control dependences of the variable z.

We now formally define our slicing criterion and how a slice is computed using this criterion.

- **Definition** (*Slicing Criterion*):

  A *slicing criterion* is of the form (f, m, v), (f, m), (f), and (F, M, V), where

$F = \{f_1, f_2, \ldots, f_j\}$ is the finite set of files in the system, $M = \{m_1, m_2, \ldots, m_j\}$ is the finite set of methods for each $f \in F$, and $V = \{v_1, v_2, \ldots, v_j\}$ is the finite set of variables for each $m \in M$.

- **Definition** (*Forward Decomposition Slice*):

A *forward decomposition slice* on variable v with respect to the slicing criterion (f, m, v) is of the form:

$$slice(f, m, v) = \bigcup_{n \in N} slice(v, n),$$

where N is a set of statements that assigns to the variable v. For the slicing criterions (f, m), (f), and (F, M, V) the slices consist of the union of static forward slices as follows:

- $slice(f, m) = \bigcup_{i=1}^{j} slice(f, m, v_i)$
- $slice(f) = \bigcup_{i=1}^{j} slice(f, m_i)$
- $slice(F, M, V) = \bigcup_{i=1}^{j} slice(f_i).$

(A)	`1.          f3 (int z) {`
	`2.            int *zp;`
	`3.            zp = &z;`
	`4.            zp++;`
	`5.          }`
	`6.          main () {`
	`7.            int var1 = 1;`
	`8.            int *p;`
	`9.            p = &var1;`
	`10.           f3 ( *p);`
	`11.         }`
(B)	*file/f3/z/@index (1), slines {1,3}, pointers {zp}*
	file/f3/zp/@index (2), slines {2,3,4}, dvariables {zp}
	file/main/var1/@index (1), slines {7,9}, pointers {p}
	file/main/p/@index (2), slines {8,9,10}, cfunctions {f3@ (1)}

Figure 3.2. (A) Sample source code (*Pointer* program used in Section 3.5), (B) System dictionary with four slice profiles for the source code in (A).

3.3 Algorithm Overview

The slicing process is performed by first converting the source code (*.c* and *.h*) files into srcML. Figure 3.3 shows the main components of the approach. As can be seen the source files are converted first using the *src2srcml* toolkit to srcML representation form, then the analyzer component start the slicing decisions at the variable level inside the function. The computations stored inside the slice profile (e.g., function calls, dependent variables, etc.) are identified as a result of the processing at this level.

The architecture of our program slicing method is described in Figure 3.3. The analyzer component did all the slicing decisions needed in order to output the system dictionary which includes all the slicing information. In order to increase the user interactivity the analyzer request from the user to specify the slicing criterion, that is the user can insert either (*f, m, v*), (*f, m*), (*f*), or (*F, M, V*) as a slicing criterion.

The algorithm extracts the source code artifacts that reflect all the possible slicing criterions, as follows: to locate the file *f* (i.e., slice starting point), and the function *m* inside *f*, the processing starts at the beginning of the srcML documents and reads the stream of tokens looking mainly for the srcML elements <unit>, <function>, <type>, and <name>. As they are encountered, tokens are classified using a decision tree as shown in Figure 3.4.

The slicing process considers the inter-procedural and intra-procedural behaviors over the entire system by taking into consideration the following language features:

- Expression statements,
- Control predicates,
- Declaration statements,
- Conditional statements,
- As well as the function definitions and function calls.

These are all easily identified in srcML. The main srcML elements of interest to detect the above features are: The declaration of the variable <decl_stmt>, using the slicing variable as a parameter in the called function <parameter_list>, a variable used in an expression <expr_stmt>, the calling statements <call>, an argument list inside the function <argument_list>. Finally, global variables are identified by the <decl_stmt> elements outside of functions (using static scoping).

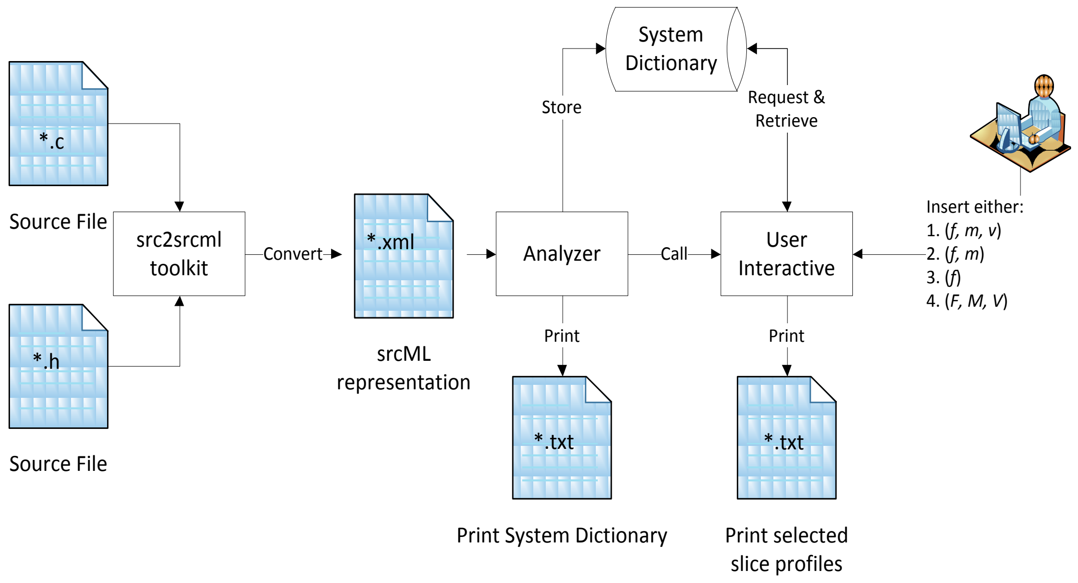


Figure 3.3. Architecture of the proposed program slicing Algorithm.

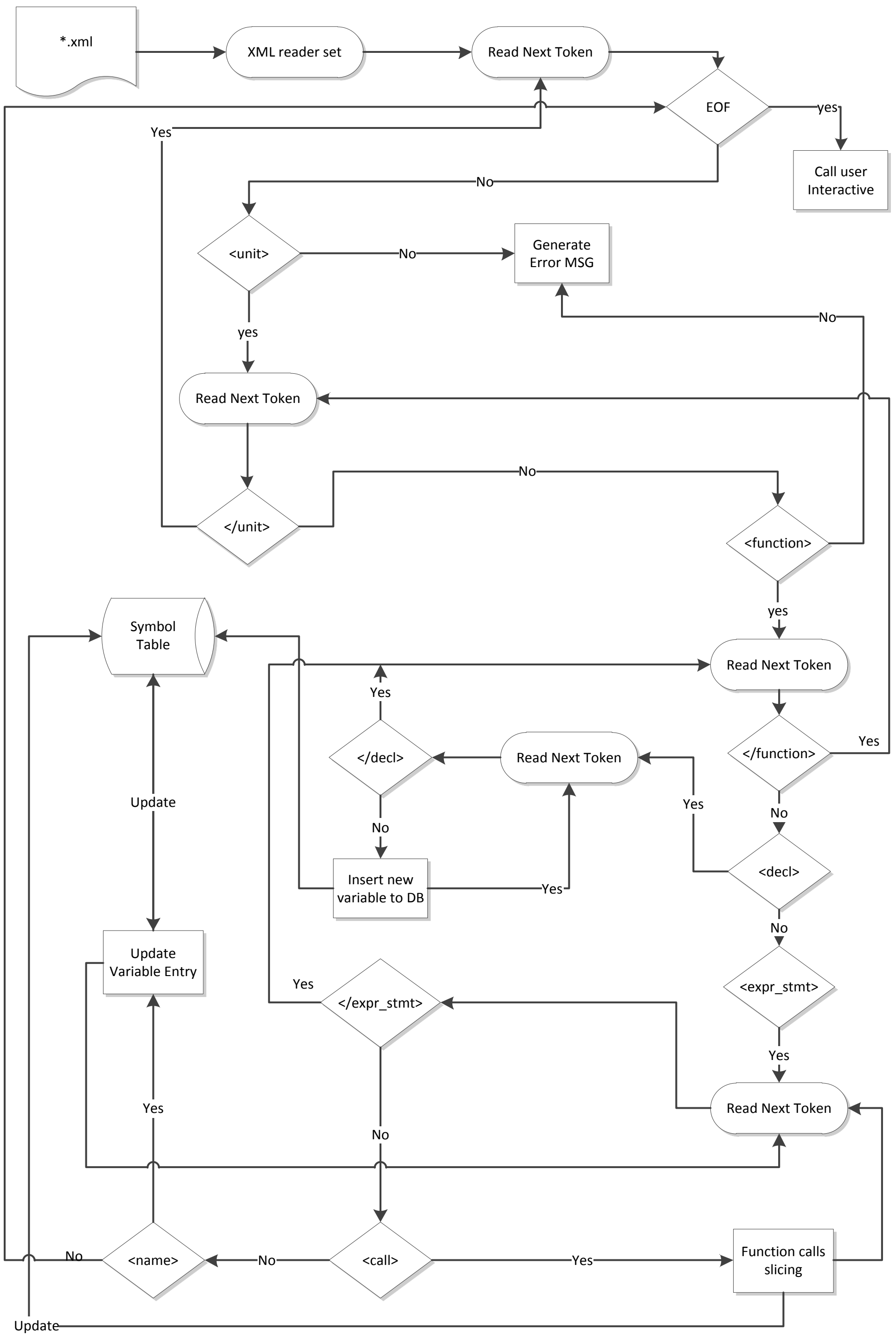


Figure 3.4. The Analyzer algorithm decision tree, drawn using flow chart symbols.

3.4 Analysis Algorithm and Implementation

Our approach is implemented by the tool *srcSlice*[1] as a forward static slicing method. Qt 4.7.0 is used because it provides a fast well-formed XML parser class called *QXmlStreamReader*[2]. In addition, Qt provides a powerful user GUI framework that allowed us to construct an interface for exploration of the different features and investigation of the various results obtained by the tool. The stream reader pulls tokens from input srcML one after another as needed. The main advantage of using the pull approach is the ability to construct a recursive parser making traversing the code quite simple. In addition, this approach is memory conservative since there is no need to store the entire srcML document tree in memory, as in a DOM approach.

The computation of the individual slices and the resulting system dictionary is computed as follows.

We first focus on the computation of data slices by identifying direct data dependencies (i.e., *def-use*). Once this problem is solved, we can compute the transitive closure over the data dependencies, and then consider the control dependencies.

The inter-procedural and intra-procedural dependencies are defined as the same definition given above in Section 2.2. An intra-procedural data-dependence relation between two points exists if the first point may assign a value to a variable that may be used by the second point. An intra-procedural control-dependence relation between two

[1] Pronounced, “Source Slice”.

[2] See http://doc.qt.nokia.com/4.7/qxmlstreamreader.html

points exists if the first point is a conditional predicate, and the execution at the second point directly depends on the result of the first point. In addition, there is an inter-procedural data-dependence relation between each function call argument and the corresponding parameter. Finally, there is an inter-procedural control-dependence relation from each call point of a function to its signature.

To extract the direct data-dependence relations between statements, we used the standard definition of *def-use* chains, except that the forward redefinition of the variable is allowed. For example from Figure 3.1, the returned slice using the criterion (c, s_3) includes the statements $\{s_3, s_4, s_6\}$. If we allow the redefinition of variable c in statement s_5 this is the decomposition slice of variable c.

Let us assume that we are interested in the slice for variable v. We start with the first definition of variable v in function/method m. Then all the expression statements where the slicing variable v is referenced are recorded including assignments, function calls, and pointer aliases. The statements that reference pointer aliases are recorded as they are impacted indirectly by the slicing variable.

The algorithm we present computes the direct data dependencies in two steps:

1. *Definition Detection*, and
2. *Use Verification*.

The output of definition detection is a set of pairs of the form (v, Sp (v)) where v is the slicing variable and Sp (v) is *v's* slice profile that initially includes the statement that defines v. A new declaration statement for a variable with the same name of the existing variable (e.g., due to scope), results in a new slice profile.

Use verification ensures that there is a *def-use* chain from the declaration statement in *Sp* (*v*) to other statements in the forward trace through which a definition of variable *v* reaches. As a result these use statements are included in *Sp* (*v*). A failure to find a *def-use* chain will result in an empty slice profile.

To compute the transitive closure over the data dependencies, all statements that include local or global variables affected by the value of the slicing variable are included. For example, the slice of an assignment statement {`a = c;`} with respect to a variable *c* will include the slice profile of variable *a*. Detecting such statements is important due to the fact that the static slicing necessitates following the slicing variable over all its possible values.

In order to locate all statements relevant to the slicing variable *v* across the boundary of the function *m*, we consider the following. Each called function in the set *cfunctions* is mapped to its function signature, i.e., the inter-procedural control-dependence relation between the call point of a function and its entry. All arguments in these function calls are mapped to the corresponding parameters in the function signature, i.e., the inter-procedural data-dependence relation between each function call argument and the corresponding parameter.

Our algorithm for computation of a forward decomposition slice is presented in Figure 3.5. After all above computations, the process of constructing the system dictionary occurs from lines 1 to 14.

The algorithm we present traces the program statements forward to determine data and control dependencies between statements. The algorithm *ComputeSlice*, shown in Figure 3.5, is the main algorithm for intra-procedural slicing. The same procedure is used when slicing over the called functions from the slice profile of the slicing variable. The *ComputeSlice* algorithm performs forward propagation of variables whose definitions are being detected. Statements and parts of statements are evaluated in the order in which they occur. This algorithm implements the definition detection by analyzing the declaration statements (*see line 28*) and parameter statements (*see line 17*), and implements the uses verification by analyzing expression statements (*see line 29*).

The algorithm *ExtractGlobal* (omitted here for brevity as much duplication exists) analyzes global declaration statements in the same way as the definition detection in the algorithm *ComputeSlice*. The *ComputeSlice* algorithm is repeatedly called for each function in file *f* to compute the transitive closure over data dependencies. The definition detection generates a set of variables. The immediate data dependencies corresponding to these variables are checked by the use verification, and the dependencies are included in the appropriate slice profiles. From the newly added statements, new sets of dependent variables are generated for the transitive closure, and the above steps are repeated until no more statements are added to the slice.

The set V (V_l or V_g) is responsible for storing the slice profiles of the variables. The elements in the set are (v, Sp (v)) pairs. Defined variables are added to the set as they are encountered. For a variable used as an l-value, a slice profile is created (if not already present) and the statement line number is added. This is done while processing declaration statements. When a variable is used as an r-value in an assignment statement, the l-value variable of the assignment statement is added to the set *dvariables* of the slice profile of the r-value variable (*see lines 34 and 35*). The set *cfunctions* for the effective variable is filled while processing function calls (*see line 44*), making it possible to compute the transitive closure across the system.

The algorithm *ComputeSlice* computes intra-procedural control dependencies as follows: Given a statement $stmt_i$ that has just been included in the slice profile of variable *v*, an immediate control predicate of $stmt_i$, say $stmt_j$, must be included in the slice profile of variable *v*. The main control predicates of interest are: *while*, *for*, *if*, *else*, *switch*, *case*, and *do*. The *return* statements are not considered, since our algorithm captures the analogous effects of a return statement appearing before the function exit through slicing over all variables. By storing those control-flow statements (loop or condition) when $stmt_i$ is included, we check to see whether it is in the body of the block of a control-flow statement. In this case it is added to the appropriate slice profile.

Inter-procedural slicing is accomplished easily by mapping the indices of the variables in the argument list (*see line 20*) to their corresponding indices detected in the calling statements (*see line 49*). From this we recover all statements that are included in the slice profile of the parameters.

For example, as shown from the set *cfunctions* in the slice profile of variable *p* in Figure 3.2 (B), the function *f3* is called using the variable *p* as index 1 in the argument list. As a result the slice profile of variable *z* in function *f3* (variable at index 1 in the parameter list) is retrieved and included in the slice profile of variable *p*.

```
Input:     $Slicing\ Criterion = (F, M, V)$

Output:  $Slice\ (F, M, V)$

$Sp$: $Slice\ Profile\ \forall v \in V$

$V_l$: $Set\ of\ local\ variables\ \forall m \in M$ - $elements\ are\ the\ pairs\ (v_i, Sp\ (v_i))$

$V_g$: $Set\ of\ global\ variables\ \forall f \in F$ - $elements\ are\ the\ pairs\ (v_i, Sp\ (v_i))$

$M$: $Set\ of\ functions\ \forall f \in F$ - $elements\ are\ the\ pairs\ (m_i, V\ (m_i))$

$F$: $Set\ of\ files\ in\ the\ system$ - $elements\ are\ the\ pairs\ (f_i, M\ (f_i))$

$Xml$: The QXmlStreamReader parser
1:  $Xml \leftarrow setDevice\ (srcML\ input)$
2:  repeat
3:  for each $f_i \in F$ do
     // for each file
4:    $ExtractGlobal\ (f_i)$
5:    for each $m_i \in M$ do
       // for each function declaration
6:      $ComputeSlice\ (m_i)$
7:      $M \leftarrow M \cup \{(m_i, V_l)\}$
8:      $V_l \leftarrow \emptyset$
9:    end for
10:   $F \leftarrow F \cup \{(f_i, M)\}$
11:   $M \leftarrow \emptyset$
12:   $V_g \leftarrow \emptyset$
13: end for
14: until the end of srcML input

algorithm ComputeSlice(m)
15: repeat
16: for each $stmt_i \in m$ in statement sequence do
17:   if $stmt_i$ is a parameter-list then
      // Detection Step
18:     for each $v \in stmt_i$ do
19:       if $(V_l = \emptyset) \vee (v \notin V_l)$ then
20:         $Sp\ (v).@index \leftarrow \{index\ of\ v\}$
21:         $Sp\ (v).slines \leftarrow \{stmt_i\ line\ number\}$
22:         $V_l \leftarrow (v, Sp\ (v))$
23:       else if $v \in V_l$ then
```

```
                // update Sp(v)
24:                 Sp (v).slines ← Sp (v).slines ∪ {$stmt_i$ line number}
25:                 $V_l$ ← (v, Sp (v))
26:             end if
27:         end for
28:     else if $stmt_i$ is a declaration statement then
            // Detection Step - repeat the lines 18-27
29:     else if $stmt_i$ is an expression statement then
           // Verification Step
30:         found ← false
31:         for each $v \in stmt_i$ , such that, $v \in V_l$ do
32:             found ← true
33:             if v is a (r-value) , such that, v (r-value) ≠ v (l-value) then
34:                 Sp (v).dvariables ← { v (l-value) }
35:                 Sp (v).slines ← Sp (v).slines ∪ {$stmt_i$ line number}
36:             else if v (l-value) is a pointer alias of v (r-value) then
37:                 Sp (v).pointers ← { v (l-value) }
38:                 Sp (v).slines ← Sp (v).slines ∪ {$stmt_i$ line number}
39:             end if
40:             $V_l$ ← (v, Sp (v))
41:         end for
42:         if !(found) then
                // for global variable, repeat lines 31 – 41 using the $V_g$ set instead of $V_l$ set
43:         end if
44:     else if $stmt_i$ is a function call then
            // extract all the arguments, with their indices
45:         for each $v \in$ argument-list do
46:             if $v \in V_l$ then
47:                 Sp (v).slines ← Sp (v).slines ∪ {$stmt_i$ line number}
48:                 Sp (v).cfunctions ← Sp (v).cfunctions ∪ {function name}
49:                 Sp (v).@index ← {index of v}
50:                 $V_l$ ← (v, Sp (v))
51:             else if $v \in V_g$ then
                    // repeat lines 47 – 50 using the set $V_g$
52:             end if
53:         end for
54:     end if
55: end for
56: until end of m
end ComputeSlice
```

Figure 3.5. Overview of the lightweight forward static slicing algorithm.

By using the system dictionary and our slice profile definition, the transitive closure can be found with a single pass through the system. The set *dvariables* (*v*) contains all the variables affected by the variable *v*. These are the variables in the transitive closure of the data and control dependencies of variable *v*. This also applies to the called functions stored in the set *cfunctions* and the pointers stored in the set *pointers*.

In the next section we will describe the design of the experiments we conducted, including benchmarks/open source programs used, and the slicing criterion and the evaluation criteria used for comparison.

3.5 Comparative Study

To assess our approach and the *srcSlice* tool we conducted a comparative study with the academic version of *CodeSurfer*[3] from GrammaTech Inc. The objective of this study is twofold. First, we want to determine if the slices produced from *srcSlice* are comparable to those produced by *CodeSurfer* in terms of the correctness and the size of the slices. That is, we compare how *srcSlice's* algorithm affects the size and the accuracy of the slices compared to a standard. The second objective is to demonstrate that our approach is highly scalable and efficient. Together these two objectives lead to three primary research questions this study tries to address:

- ***RQ1***: *Does srcSlice produce accurate slices?*
- ***RQ2***: *Is there an unacceptable level of inaccuracy?*

---

[3] CodeSurfer ® version 2.1p1, Copyright © GrammaTech Inc., See http://www.grammatech.com/products/codesurfer.

- ***RQ3*: *Is srcSlice highly scalable and efficient?* (*Will be covered in Section 3.6*).**

The question of what is a perfectly accurate slice is somewhat open to interpretation [Weiser 1979]. This is the case for many results of static analysis. For example, an empirical study of static call graph extractors by Murphy et al [Murphy, Notkin 1998] demonstrates that the call graphs extracted by several broadly distributed tools vary significantly enough to surprise many experienced software engineers. These differences are shown with nine different call graph extraction tools of C code from three software engineering systems. In particular, an evaluation and comparison of five different implementations of program slicing by Hoffner [Hoffner 1995] showed that the resulted slices differ in their size and accuracy. His study covered three inter-procedural slicing tools.

In order to evaluate our slicing approach, we compare the results obtained by our tool to the results of *CodeSurfer*. The same benchmarks are given to both tools. We feel that by comparing our results to that of a commonly used existing approach/tool; will minimally give us a baseline with respect to the accuracy of the results. That is, if our results are similar to that of *CodeSurfer*'s we feel confident that it produces reasonably correct slices.

3.5.1 Set-Up and Configurations

CodeSurfer is a commercial based slicing tool for C/C++ programs. Produced originally as a research tool, it is now available from GrammaTech Inc. It is based on the slicing work done at the University of Wisconsin surveyed in [Horwitz, Reps 1992].

This tool starts by generating a control-flow graph (CFG) for each source file in the system, and then the SDG is constructed for the entire system. *CodeSurfer* views slicing as a graph reachability problem, either backward or forward with three options for dependences: control-dependence edges only, data-dependence edges only, or both edges.

There are several features provided by *CodeSurfer* to assist in the code analysis process of slicing including the extraction of return values, passed by reference parameters, and modified global variables for a given function scope. *CodeSurfer* allows the user to control the settings of the above features with five different static-analysis parameters that affect the level of precision and consequently the build time.

For the *Super-lite* setting all expensive analyses are disabled including pointer analysis and no data or control dependencies. The *Lite* setting is the same as Super-lite except that the control-flow graph is generated. For the *Medium* setting the intra-procedural data dependencies are calculated, but no inter-procedural data dependencies, and imprecise, but more efficient, pointer analysis is performed. For the *High* setting the full functionality is supported with high precision, except that dynamic storage is not included in the pointer analysis [Bent, Atkinson, Griswold 2000]. The *Highest* setting eliminates this last limitation.

In the context of this study, *CodeSurfer* has two main limitations. The first limitation is that *CodeSurfer* does not have the ability to slice incomplete and non-compile-able source code. While this may not be a major deficiency our approach does not require the system to be compiled (or complete). The second limitation is the free academic version of *CodeSurfer* used in this study is unable to slice programs larger than 200 KLOC [Wisconsin] Thus, all of the benchmark cases used here are under 200 KLOC.

For our study, the Highest setting for *CodeSurfer* was used to provide the most precise results. We expect to obtain a *safe* or *conservative* (definitions introduced in next section) slicing results at the expense of longer build times. Since the tool can be used for other tasks than computing slices, all features except the slicing results were turned off.

3.5.2 Evaluation Criteria

Here we want to evaluate the slicing results of our tool to determine if correct slices are produced, and are produced efficiently in both time and space required. The time and cost it takes to generate the slice, including execution time and memory requirements, is of particular interest with respect to usability of the method. In addition, we want to determine if the results obtained by *srcSlice* are comparable to *CodeSurfer* in terms of accuracy. However, since the implementations of these tools have so few aspects in common, it is not meaningful to compare all of the relevant aspects of the different implementations. Therefore, we focus our attention on evaluating slices of both tools, by taking into consideration the correctness, size of the results, time and space efficiency, and the limitations of both tools. Finally, we investigate most of the language features supported, e.g., is the tool able to handle pointers, call by reference, etc.

3.5.2.1 Slice Size and Quality

The correctness of the slice relates directly to its purpose [Hoffner 1995]. A small slice that contains relevant parts of a program for a specific input could be used for locating bugs, but it might be too small for applications where we have to consider all possible inputs (e.g., overall program comprehension) [Pan, Kim, Whitehead 2006].

The slice size (denoted by *SZ*) for both tools is measured by the number of statements lines of code. The best slice for a given slicing criterion should be the smallest correct slice. The slice is *safe* if it contains every statement that is actually affected by the slicing criterion. A safe slice is *conservative* if it may be imprecise, i.e., if it also contains statements that are not affected by the value of the variable in the slicing criterion (*false positive*). The *minimal slice* is a safe slice that contains no unnecessary statements [Bent, Atkinson, Griswold 2000], i.e., no other slice for the same slicing criteria contains fewer statements. The slice efficiency factor can be measured by how close the resulting slice is to the minimal slice [Hoffner 1995].

The problem of determining the minimal slice is not in general decidable [Weiser 1979; Bent, Atkinson, Griswold 2000; Danicic, Fox, Harman, Hierons, Howroyd, Laurence 2005]. In fact, such a set is un-computable because of the un-decidability of the required static analysis. However, as mentioned earlier the definition of what is a minimal slice depends on the intended use. Therefore, even with the most precise slicer, the resulting slice is at best a conservative approximation of the minimal slice, i.e., the *resulting slice* $\supseteq$ *minimal slice*.

A preferred decrease in the slice size is limited by the ability of the resultant slice to reflect all system behavioral aspects. Binkley et al [Binkley, Gold, Harman 2007] observed that after studying 43 programs with ~1 MLOC that the most precise program slicer had an average slice size equal to 30% of the original program. He studied five factors that may influence the slice size including the expansion of structure fields, the inclusion of calling context, the level of granularity of the slice, the presence of dead code, and finally the choice of points-to analysis.

For the purpose of comparison, we use the intersection of corresponding slices returned by the both tools, called the intersected slice following the same approach used in the qualitative study of Bent et al [Bent, Atkinson, Griswold 2000]. That study and compared the dataflow-slicing approach (*Sprite*) and a PDG-slicing approach (*CodeSurfer*). Bent used the intersected slice as an approximation of the minimal slice. The relative safety margin (denoted by *SM*) of a slice (size of resultant slice divided by the size of the intersected slice) was used to provide a measure of the relative quality.

Let us assume that the corresponding slices returned from both tools are correct with different contents. In that case, the differing statements are not required to be in the slice. Because the statements they are not included would be incorrect, this is a contradiction with the assumption that both slices are correct. Therefore, a smaller correct slice must exist that does not include the differing statements, i.e., intersected slice. Hence, the *intersected slice* $\supseteq$ *minimal slice*.

However, as our performance benchmarks results demonstrate in the *Slice Intersection Comparison* section (Section 3.5.3.5); we can obtain several hints that indicate an approximation of the minimal slice using *srcSlice* tool. Our results indicate that the *srcSlice's* slice ≅ *intersected slice* fairly often, so that the *srcSlice's* slice ⊇ *minimal* slice.

In this work, the slice size represents the total slice size (denoted *TSZ*), the sum of individual slice sizes for each slicing criterion. If there are *n* criterions (denoted by the set $SC = \{sc_1, sc_2, \ldots, sc_n\}$), then the total slice size is denoted by:

$$TSZ(SC) = \sum_{i=1}^{n} SZ(sc_i)$$

3.5.2.2 Slicing Time

The building time (denoted *BT*) reports the time required to build the SDG for *CodeSurfer* and the system dictionary for *srcSlice*. *CodeSurfer* does most of its work during the build phase. The build phase pre-computes a large amount of information, storing the SDG that contains data and control dependencies, and pointer information.

Whenever *CodeSurfer* slices a program, it must reload that information from disk with the slicing performed any number of times for a particular load.

The slicing time (denoted *ST*) is the time it takes to handle a particular slicing request. If there are *n* slicing criterions (denoted by the set $SC = \{sc_1, sc_2, \ldots, sc_n\}$), then the total slice time, the sum of individual slice times for each slicing criterion, is denoted by:

$$TST(SC) = \sum_{i=1}^{n} ST(sc_i)$$

Furthermore, the total time overhead for one build for both slicers is denoted by: $T(SC) = BT + TST(SC)$.

Previous studies on program slicing focus on slices produced using one slicer, and do not consider the build time [Binkley, Harman 2003; Binkley, Gold, Harman, Li, Mahdavi 2006; Binkley, Gold, Harman 2007; Binkley, Harman, Hassoun, Islam, Li 2010]. However, Bent et al [Bent, Atkinson, Griswold 2000] verify that many slices using *CodeSurfer* take almost zero seconds once the load time is excluded, such that $ST(sc_i) \cong 0.00$. For comparison purpose with our tool, the build times are substantially larger than the total slicing time. Therefore, the time needed for retrieving the slice is ignored in our comparison since as mentioned early; both tools do their slicing while constructing the system dictionary and SDG.

We captured the build time for all of the slices using the UNIX built-in *time* command. The *wall-clock* time is reported since this represents the actual time a user is wait for her results. The time to convert to srcML is also included. It took less than a second to generate the srcML for the feature benchmarks and close to 11 seconds for the largest program, *cvs-1.12.10*, in the performance benchmarks.

3.5.3 Benchmarks Studied

We first ran both slicers on a number of small programs (aka *feature benchmarks*). These results are used to determine the correctness of our results and help explain slicing results in larger programs. This initial comparison leads us to a more comprehensive comparison.

Second, we ran both slicers on larger open-source programs (aka *performance benchmarks*) of varying size that worked with both slicers. These results are used to illustrate the first and the second research questions (*RQ1* and *RQ2*) and partially address the third research question (*RQ3*). Finally, we ran *srcSlice* on the Linux kernel to answer the third research question (*RQ3*).

Table 3.1 and Table 3.3 show a list of the feature benchmarks and the performance benchmarks, respectively, along with statistics related to the programs. These statistics include three measures of program size:

- The size of each program in LOC as reported by *wc –l* utility,
- The size of the program as file counts, and
- The size of the program as function counts.

Table 3.2 and Table 3.4 show the results of the feature benchmarks and the performance benchmarks, respectively, along with statistics related to the programs and their slices. These statistics include:

- Slices taken,
- Slicing time,
- Slice size, and
- Slice size relative to LOC.

In both tables, the number of slices taken represents the number of forward slices over all possible criterions for each program. For *CodeSurfer* this corresponds to slicing for each vertex in the SDG that represents executable code. In *srcSlice* this number represents the number of variables in the program using the (*F, M, V*) slicing criterion.

Table 3.1. Feature Benchmarks studied.

Program	Size			Slicing Criterion	
	LOC	# of Files	# of Functions	Method	Variable
Information _flow	112	1	12	main	hi
				All Possible Criterions	
Sum	21	1	2	main	sum
				All Possible Criterions	
Wc	39	1	3	line_char_count	eof_flag
				Scan_line	i
				All Possible Criterions	
Pointer	36	1	5	main	var1
				All Possible Criterions	
Testcases	114	1	14	main	var1
				All Possible Criterions	
Callofcall	24	1	3	main	var1
				All Possible Criterions	
Total	346	6	39		
Average	57.7	1	6.5		

Note that in Table 3.2 and Table 3.4, the number of slices for both tools and the number of lines of code from the corresponding tables (Table 3.1 and Table 3.3) do not match, since in the PDG based slicing approach one line of code could be represented by multiple vertices [Binkley, Harman 2003]. In contrast, our slicing approach is a variable granularity. Thus, one line of code may have several variables.

The line counts for the programs and the slice sizes are included to provide a consistent measure of sizes that facilitates comparison with our results. For instance, calculating the intersection of corresponding slices returned require knowing which source lines relevant to slicing query.

3.5.3.1 Feature Benchmarks

The first study concerns feature benchmarks chosen to evaluate various language features, with programs of different sizes and complexities. A list of these programs is given in Table 3.1.

The column *Slicing Criterion* contains the inputs used for the slicing process. For each program we used our experience as programmers to select slicing criterion that we felt expose the effects of the language features on each slicer's behavior. Additionally, in order to avoid any possible bias from our choices, we also computed the slice over all possible slicing criterions for each program. This represented in Table 3.1 as *All Possible Criterions* entry.

CodeSurfer can take different combinations of slicing criterion including the point (line number), variable name, and function name. In order to unify the results obtained by both tools and since all feature benchmarks were in one source file, we adjusted the slicing criterion for *srcSlice* to use the criteria format (*f, m, v*).

The programs *Information_flow*, *Sum*, and *Wc* were those provided with *CodeSurfer* as critical test cases. The programs *Pointer*, *Callofcall*, and *Testcases* were written by the authors to assess additional critical test cases.

The main language features of the above programs are as follows:

- *Information_flow*: pointers, pointer casting, double pointers, data and control dependencies, global variables, function indirection,

- *Sum* and *Wc*: external libraries,
- *Pointer*: pointer flow,
- *Callofcall*: nested function calls, and
- *Testcases*: functions calls, local and global variables, call by reference, call built-in functions, dependence flow.

The programs we developed attempt to cover a range of test cases in C/C++ that are critical for most slicing methods [Binkley 1993; Bent, Atkinson, Griswold 2000; Binkley, Gold, Harman 2007]. The purpose of these programs was to exercise the slicing behavior and for in-depth analysis.

In general, these language features included the following:

- Detecting function calls inside control blocks, such as while, if, for, etc. For example, function *f1* is called inside the *if-block* and *while-block* as shown:

```
if ( f1(v1, v2) );
while ( f1 (v1, v2) > 0 );
```

- Tracking multiple call depths, for instance the *f1* function calls the *f2* function with the slicing variable *v1*, and the *f2* function calls the *f3* function with the argument *v2* assigned to, and so on:

```
void f1() { f2(v1); }
void f2(int v2) { f3(v2); }
```

- Nested function calls, for instance where the function *f1* uses function *f2* as one of its parameters. We paid particularly close attention to this case, because most of the existing static slicing methods do not consider the types of parameters.

  Frank Tip [Tip 1995] shows in his study of 22 static slicing approaches that the only approach that takes parameter aliasing into account was with Binkley [Binkley 1993]. Of the other 21 approaches only 9 could support inter-procedural slicing. For example, the intra-procedural algorithm produced by Weiser does not take into account which output parameters depend on which input parameters.

  As we can see in the following example, the value returned from the function *f2* is used for the second argument of the function *f1*. The result is that the slice profiles of both functions are merged.

```
f1(v1, f2(v2), v3);
```

- Distinguish between local and global variables having the same name, and detecting the flow of the data dependence between them. In addition, there are cases that include transitive dependence (indirect dependence).
- Call by reference parameter passing. This case supports pointer aliases.

```
void f1(int &x, int y, int w);
```

- Slicing over pointer variables. As shown below the pointer *p* is defined as a reference for the variable *v*. So the slice profile of pointer *p* should be part of the slice profile of *v*, since we can refer to *v* using the pointer *p*.

```
int *p; p = &v; f(*p);
```

- Detecting the calls of library functions whose implementation may not be available: for example calling function `abs ()` from the library `#include <cmath>`.

In our approach, the code in external libraries is not analyzed as in the case of *CodeSurfer*. Specifically, we do not include any code in the analysis unless specifically provided. We try to keep the slice space at a minimum while still being useful in testing and maintenance tasks.

3.5.3.2 Feature Benchmarks Results

Table 3.2 shows the results obtained by *srcSlice* and *CodeSurfer* for the feature benchmarks. The *Program* column is the benchmarks used for comparison. The *Slices Taken* column is the number of forward slices taken by both tools. As seen in the last row of the table, for *CodeSurfer* the number of slices taken is 444. For srcSlice 79 slices were taken.

The program *Testcases* covers most of the language issues discussed in previous section. The slices obtained by running both tools using the slicing criterion (*main, var1*) were observed to be correct; however, *CodeSurfer* included some global variables that did not have any dependence on the slicing variable.

Binkely et al [Binkley, Gold, Harman 2007] reasoned that this case due to the fact that the slice size in the SDG reports the global variables that modeled as a *value-result* parameters. Thus each global variable counts as a node in the SDG added at both the caller and procedure entry. In contrast, *srcSlice* ignores those variables in the returned slice. Table 3.2 demonstrates these results, as the slicing time and the slice size of *srcSlice* are smaller using both types of the slicing criterion. According to the definition by Hoffner [Hoffner 1995], the best slice should be the smallest correct slice. Manual checking of the slices produced by both tools showed that they were 100% correct; however *srcSlice* produced a smaller slice.

From Table 3.2, we can see that the slice size of *srcSlice* is consistently smaller than the ones produced by *CodeSurfer* (the average forward slice contained 45.2% of the program source using *CodeSurfer* and 34.1% using *srcSlice*) except for the program *Pointer* using the slicing criterion (*main, var1*).

A closer investigation of this program shows that for the sample code in Figure 3.2 (A), *CodeSurfer* has limitations in detecting the flow from pointer **p* in line 10 to the receiver argument *z* in function *f3*, which is assigned in the body of the function to pointer *zp* at line 3.

Table 3.2. Feature Benchmarks results and comparison of CodeSurfer and srcSlice, time measured in seconds, slice size measured in number of statements, (%) columns are the slice size relative to LOC.

Program	CodeSurfer				srcSlice			
	Slices Taken	Slicing Time	Slice Size	%	Slices Taken	Slicing Time	Slice Size	%
Information _flow	1	1.5	32	28.6	1	1	27	24.1
	149		66	58.9	22		48	42.9
Sum	1	1	6	28.6	1	0.5	4	19.0
	26		14	66.7	2		8	38.1
Wc	1	1.2	16	41.0	1	0.3	10	25.6
	1		7	17.9	1		4	10.3
	46		24	61.5	9		19	48.7
Pointer	1	1.5	11	30.6	1	0.4	15	41.7
	37		25	69.4	8		17	47.2
Testcases	1	7.7	50	43.9	1	0.6	44	38.6
	156		79	69.3	24		56	49.1
Callofcall	1	2.9	4	16.7	1	0.4	4	16.7
	23		13	54.2	7		10	41.7
Total	444	15.7	347		79	3.2	266	
Average	34.2	2.6	26.7	45.2	6.1	0.5	20.5	34.1

Bent et al [Bent, Atkinson, Griswold 2000] describe this case as a gray area in the *CodeSurfer* algorithm, and defined it as "*handling undefined entities*". In particular a call of the form *f3* (*&var1*) will not be treated as a possible definition of *var1*. Also, an uninitialized pointer will not be a member of any *points-to* set so any effects through it are not tracked. That is, the statements **p = var1*; *z = *p*; when slicing on *var1* will not add *z* to the slicing criterion, nor is there a warning. However, this is not the case when slicing over all possible criterions, i.e., the number of slices taken is equal to 37. The slice size is equal to 25, and manual checking of the returned slice showed that *CodeSurfer* detects the lines from 1 – 4 using the slicing criterion *(f3, *p)* in line 10.

In contrast, *srcSlice*, as shown in Figure 3.2 (B), captures this case and included in the slice profile for each variable. This inability to track the chains of pointers in this particular example in *CodeSurfer* results in a slice with missing critical statements, especially when the slice includes aliases of the original variable.

The accuracy of the slices produced using *srcSlice* for the programs *Information_flow*, *Sum*, *Callofcall*, and *Wc* was identical to *CodeSurfer*. The slices produced using *srcSlice* was manually checked and found to be correct. The difference in the results obtained by *CodeSurfer* was due to retrieving unrelated statements; such as statements mentioned inside the blocks of *for* and *while* predicates and standard libraries. That is, *CodeSurfer* highlights statements that are not only semantically related to the slicing criterion but also syntactically related to the executable slice [Bent, Atkinson, Griswold 2000]. For example, *CodeSurfer* returned all relevant statements that modify or determine control flow statement in the *else* part of an *if* statement whose body was not in the slice.

As shown, for the settings chosen, *CodeSurfer* provides a correct slice with regards to data and control dependencies. The results also show that *srcSlice* produced accurate slices when compared to *CodeSurfer*. We note again that the settings used for *CodeSurfer* were to enhance accuracy and not performance.

3.5.3.3 Performance Benchmarks

The second study considers just over 700 KLOC of C code from 16 open-source programs that range in size from approximately 3 to almost 150 KLOC.

Table 3.3. Performance Benchmarks (open-source programs) studied.

Program	Version	Size		
		LOC	# of Files	# of Functions
ed-1.2	1.2	3087	10	126
ed-1.6	1.6	3260	10	128
which-2.20	2.20	3586	14	51
wdiff-0.5	0.5	3874	13	56
barcode-0.98	0.98	5205	18	74
acct-6.5	6.5	8749	27	127
enscript-1.4.0	1.4.0	18162	52	180
make-3.82	3.82	36397	58	474
enscript-1.6.5.2	1.6.5.2	56491	107	488
enscript-1.6.5	1.6.5	56494	107	488
enscript-1.6.5.1	1.6.5.1	56494	107	488
a2ps-4.10.4	4.10.4	57052	188	1104
findutils-4.4.2	4.4.2	72384	314	1141
radius-1.0	1.0	82029	196	1719
dico-2.2	2.2	119592	332	2504
cvs-1.12.10	1.12.10	144278	340	2027
Total		727134	1893	11175
Average		45446.9	118.3	698

Table 3.3 shows these 16 programs along with the three program size measures explained in Section 3.5.3. In this table, the slices taken represent the number of forward slices over all possible criterions for each program.

The performance programs were chosen to cover a wide range of programming styles (e.g., *acct* contains different related computations; *ed* has a single purpose). Eight of these programs appear in Binkley's studies [Binkley, Harman 2003; Binkley, Gold, Harman 2007] although they may be different versions.

Net we will present data comparing slices from both slicers along with an investigation of the intersected slice using the performance benchmarks.

3.5.3.4 Performance Benchmarks Results

The results obtained by *srcSlice* and *CodeSurfer* for the performance benchmarks in Table 3.3 are given in Table 3.4. Each row in the table is a benchmark we used for the comparison. As seen in the last row of the table, the average slice size using both tools over all 16 programs included between 23.0% and 29.3% of the program source code. The range of the slice size coverage in the program for *CodeSurfer* is striking with an overall range from 16.8% for *wdiff-0.5*, to 57.7% for program *ed-1.2*. *srcSlice* had a narrower overall range from 13.0% for *which-2.2* to 38.1% for *ed-1.6.*

Preliminary analysis does not indicate any trend relating program size and slice size using both slicers. Smaller LOC (*ed-1.2 with 3087*) gives high percentages (*CodeSurfer* = 57.7%, *srcSlice* = 37.1%), and the larger LOC (*wdiff-0.5 with 3874*) gives low percentages (*CodeSurfer* = 16.8%, *srcSlice* = 14.5%). The same thing occurs with programs *ed-1.6* and *which-2.2*. The program size is only one of the program attributes that potentially affects slice size, as the programming style (i.e., number of methods, global variables, pointers, etc.) also affect slice size.

Table 3.4. Performance Benchmarks results and comparison of the CodeSurfer and srcSlice, slicing time measured in seconds, slice size measured in number of statements, (%) columns are the slice size relative to LOC.

Program	CodeSurfer				srcSlice			
	Slices Taken	Slicing Time	Slice Size	%	Slices Taken	Slicing Time	Slice Size	%
ed-1.2	4438	21.8	1782	57.7	516	4.4	1146	37.1
ed-1.6	4527	19.9	1863	57.1	560	4.4	1241	38.1
which-2.20	1429	15.2	736	20.5	203	4.3	465	13.0
wdiff-0.5	1097	11.9	652	16.8	136	4.4	561	14.5
barcode-0.98	4590	29.5	2177	41.8	451	4.5	1647	31.6
acct-6.5	4983	47.5	2510	28.7	868	4.8	2193	25.1
enscript-1.4.0	9456	71.1	5916	32.6	1139	6.0	3073	16.9
make-3.82	17012	807	9446	26.0	3459	8.7	10703	29.4
enscript-1.6.5.2	20234	184	12907	22.8	3043	11.8	10119	17.9
enscript-1.6.5	20252	184	12913	22.9	3050	11.0	10126	17.9
enscript-1.6.5.1	20252	185	12913	22.9	3050	10.8	10126	17.9
a2ps-4.10.4	24493	393	14249	25.0	4119	13.9	9035	15.8
findutils-4.4.2	23641	215	13689	18.9	7229	18.3	17298	23.9
radius-1.0	38487	335	19218	23.4	7822	16.5	18287	22.3
dico-2.2	52297	1763	28639	23.9	13012	22.7	30703	25.7
cvs-1.12.10	74328	286328	40869	28.3	10116	26.5	30310	21.0
Total	321516	290584	180479		58773	173	157033	
Average	20094.8	19372	11279.9	29.3	3673.3	10.8	9814.6	23.0

One way to see this is to take a look at the number of slices taken for both tools. For example in the two programs *a2ps-4.10.4* and *findutils-4.4.2*, the slice size as a percentage are related directly to the number of slices taken by the *CodeSurfer* and *srcSlice*, as follows: in *a2ps-4.10.4* the numbers of slices taken are 24493 and 4119, with slice percentages equal to 25.0% and 15.8%, respectively. Moreover, in *findutils-4.4.*2 the numbers of slices taken are 23641 and 7229, with 18.9% and 23.9%, respectively. Even though LOC size for *findutils-4.4.2* program is larger. As explained early the number of slices taken by *CodeSurfer* is related to the number of executable vertices in the SDG, for *srcSlice* related to the number of variables in the program.

In general, the slice size produced by *srcSlice* is smaller than the one produced by *CodeSurfer*; however this is not the case in 3 out of 16 cases, i.e., *make-3.82*, *findutils-4.4.2*, and *dico-2.2*. The intersected slice results next will give us several hints why this is so.

On a per-program and overall basis, *srcSlice's* slicing time is very fast; the smaller programs took around four seconds (including the time to convert to srcML). This is an indication that the pre-computation strategy is successful at reducing slicing costs. The slicing times for *CodeSurfer* range from ~11 to ~286,000 seconds, with the highest settings for precision used and are up to 19,000 times slower than *srcSlice's*. However, *CodeSurfer* does produce a larger number of slices (~5.5 times more), which accounts for part of the slow down. By excluding the larger program (*cvs-1.12.10*), *CodeSurfer's* slicing time reduces to ~126 times that of *srcSlice*.

3.5.3.5 Slice Intersection Comparison

We use the intersected slice as a measure of the quality of calculated slice. As explained in Section 3.5.2, we feel that by intersecting our results to that of a *CodeSurfer* will minimally give us a baseline with respect to accuracy of the results. That is, if our results are closer to that of the intersected slice we feel confident that it produces reasonably correct slices.

The slices of selected files are generated using all possible slicing criterions with both tools, and then the intersection between corresponding slices is taken. The intersected slices are generated for two performance benchmarks from Table 3.3 which are *enscript-1.6.5* and *findutils-4.4.2*. Those programs were particularly chosen to demonstrate the exceptions where the slice size differed drastically between the tools.

The results of running the slicers on all possible criterions over 13 files of program *enscript-1.6.5* are presented in Table 3.5. In order to provide a better estimate of file size, the third column reports the number of non-blank non-comment lines of code as reported by *sloc-count* utility[4]. In this case the results of the intersected slice are more reasonable since both slicers should return only source code statements (no comments no blanks).

Focusing first on the slice sizes, it is apparent that for all slices *srcSlice's* results are consistently smaller. The average slice size for *CodeSurfer* and *srcSlice* is 69.0% and 32.4%, respectively. Upon closer examination, we observe that *CodeSurfer* produced a

[4] See http://www.dwheeler.com/sloccount/sloccount.html

higher safety margin (*SM*) on all slices than those produced by *srcSlice*. *CodeSurfer* produced a maximum *SM* of 8.18% on the slice of *afmlib/deffont.c* file, and a minimum (close to the intersected slice) of 1.31% on the slice of *afmlib/afm.c* file.

Table 3.5. Intersected Slice over 13 files from enscript-1.6.5, where (%) in the CodeSurfer (CS) and srcSlice (sS) columns is the slice size relative to LOC, (%) in the intersection column is the intersected slice relative to both tools slice size, (SM) is the relative safety margin for a slice, L = slice size in number of lines.

enscript-1.6.5	Size		Slice Size						Intersection		
			CodeSurfer (CS)			srcSlice (sS)					
File Name	LOC	SLOC	L	%	SM	L	%	SM	L	CS %	sS %
src\psgen.c	2860	1993	1351	67.8	1.75	863	43.3	1.12	771	57.1	89.3
src\util.c	2156	1623	1227	75.6	1.48	853	52.6	1.03	827	67.4	97.0
src\main.c	2660	1406	1178	83.8	1.59	768	54.6	1.04	739	62.7	96.2
src\mkafmmap.c	250	153	92	60.1	2.04	45	29.4	1.00	45	48.9	100.0
afmlib\strhash.c	386	268	145	54.1	1.36	145	54.1	1.36	107	73.8	73.8
afmlib\afmparse.c	1017	759	636	83.8	2.05	313	41.2	1.01	310	48.7	99.0
states\ex.c	2378	1536	813	52.9	3.35	279	18.2	1.15	243	29.9	87.1
states\gram.c	2408	1607	433	26.9	2.41	301	18.7	1.67	180	41.6	59.8
afmlib\afm.c	824	590	468	79.3	1.31	357	60.5	1.00	357	76.3	100.0
afmlib\afmtest.c	184	113	67	59.3	1.60	42	37.2	1.00	42	62.7	100.0
afmlib\deffont.c	379	323	311	96.3	8.18	38	11.8	1.00	38	12.2	100.0
afmlib\e_88594.c	284	261	190	72.8		0	0.0		0	0.0	0.0
afmlib\e_mac.c	284	261	219	83.9		0	0.0		0	0.0	0.0
Total	16070	10893	7130			4004			3659		
Average	1236	838	548	69.0	2	308	32.4	1	281	52.8	91.1
Min	184	113	67	26.9	1.31	0	11.8	1	0	12.2	59.8
Max	2860	1993	1351	96.3	8.18	863	60.5	1.67	827	76.3	100

In contrast, *srcSlice* produces a maximum *SM* of 1.67% on the slice of *states/gram.c* file, and a minimum *SM* of 1% (identical to the intersected slice) on four files. As shown, the slice size produced by *srcSlice* is consistently closer to the intersected slice. The intersected slice size relative to the *srcSlice's* and *CodeSurfer's* sizes, in average are equal to 91.1% and 52.8%, with a maximum of 100% and 76.3%, respectively.

The size of the intersected slice for the file *afmlib/deffont.c* was small (*38 lines*). In addition, the intersected slice size on files *e_88594.c* and *e_mac.c* from the same directory was zero. A closer examination of the slices, particularly the two files *e_88594.c*, and *e_mac.c* with the same size 261 SLOC, shows that both files contain 258 SLOC of array initialization values of the form {*0x00, AFM_ENC_NONE*}. This indicates that imprecision with regards to large array initialization might be an issue.

Because the *CodeSurfer* algorithm treats each element of an array as a distinct variable, [CodeSurfer], the slice sizes from *CodeSurfer* for these files were 72.8% and 83.9% respectively. This more precise approach requires complex dependence analysis, however this leads to unnecessarily large slices [Xu, Qian, Zhang, Wu, Chen 2005].

In contrast, the *srcSlice* algorithm treats the entire array as a single variable, and the declaration of the array is detected and processed the same as a scalar variable. That is, if the array is not referenced inside the file, then the slice size is zero. The same senario occurs in the *deffont.c* file which contains a 262 SLOC array declaration. The results of comparing the slices of the *srcSlice*, *CodeSurfer*, and the intersection on 13 files from *enscript-1.6.5* program are shown in Figure 3.6.

The results of comparing the slices of *srcSlice*, *CodeSurfer*, and the intersection of these slices on 10 files from the program *findutils-4.4.2* are shown in Figure 3.7. As can be seen, the *srcSlice* results are always closer to the intersected slice, except for the file *find/defs.h*. In this case, the *CodeSurfer* slice size is only 1.7% of a 348 SLOC file as shown in Table 3.6. However, we are unsure of the cause of the imprecision in *CodeSurfer*.

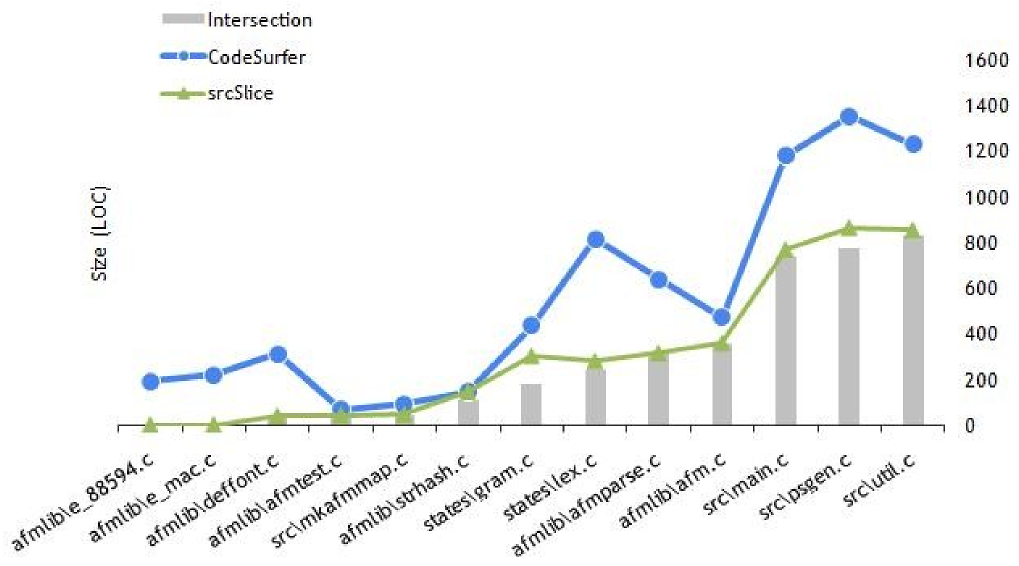


Figure 3.6. Comparison of CodeSurfer, srcSlice, and the slice intersection over 13 files from the program enscript-1.6.5 ordered by the size of slice intersection. The srcSlice was much closer to the slice intersection than CodeSurfer.

Upon closer examination, we observe that *srcSlice* produced a higher safety margin (*SM*) on only one slice than the one produced by *CodeSurfer*. For this particular file, *srcSlice* produced a maximum *SM* of 40.20% on the slice of *find/defs.h* file, and a minimum (close to the intersected slice) of 1% on the slice of *gnulib/lib/exitfail.c* file. In contrast, *CodeSurfer* produces a maximum *SM* of 2.35% on the slice of *xargs/xargs.c* file, and a minimum *SM* of 1% on *gnulib/lib/exitfail.c* file.

Table 3.6. Intersected Slice over 10 files from findutils-4.4.2, where (%) in the CodeSurfer (CS) and srcSlice (sS) columns is the slice size relative to LOC, (%) in the intersection column is the intersected slice relative to both tools slice size, (SM) is the relative safety margin for a slice, L = slice size in number of lines.

findutils-4.4.2	Size		Slice Size						Intersection		
			CodeSurfer (CS)			srcSlice (sS)					
File Name	LOC	SLOC	L	%	SM	L	%	SM	L	CS %	sS %
gnulib/lib/exitfail.c	24	4	1	25	1.00	1	25.0	1.00	1	100	100
find/defs.h	660	348	6	1.7	1.20	201	57.8	40.20	5	83	2.5
find/util.c	1040	715	419	58.6	1.68	279	39.0	1.12	249	59	89
xargs/xargs.c	1380	941	610	64.8	2.35	264	28.1	1.02	260	43	99
find/find.c	1532	959	572	59.6	1.58	393	41.0	1.09	361	63	92
gnulib/lib/getdate.c	3332	2366	1034	43.7	2.85	533	22.5	1.47	363	35	68
find/tree.c	1655	1202	813	67.6	1.64	501	41.7	1.01	497	61	99
locate/locate.c	1941	1374	914	66.5	1.69	548	39.9	1.01	541	59	99
find/pred.c	2553	1816	1176	64.8	1.62	745	41.0	1.03	724	62	97
find/parser.c	3544	2601	1746	67.1	1.57	1132	43.5	1.02	1109	64	98
Total	17661	12326	7291			4597			4110		
Average	1766	1232	729	52	1.7	459	38	4.9	411	63	84
Min	24	4	1	1.7	1	1	22.5	1	1	35	2.5
Max	3544	2601	1746	67.6	2.85	1132	57.8	40.2	1109	100	100

However, if we exclude the *find/defs.h* file, then the slice size produced by *srcSlice* is consistently closer to the intersected slice. The intersected slice size relative to the *srcSlice's* and *CodeSurfer's* sizes, in average are equal to ~84% and ~63%, respectively.

So, If we exclude the *find/defs.h* file then the *SM* calculations using *srcSlice* will be as follows: average = 1.08, minimum = 1, and the maximum = 1.47. And these results are closer to the slice intersection than *CodeSurfer* results.

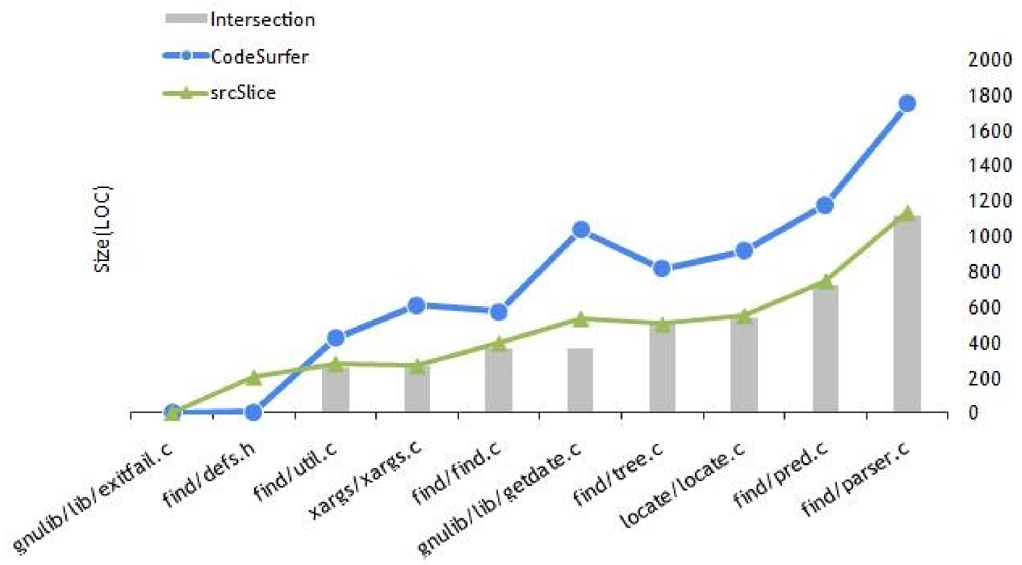


Figure 3.7 Comparison of CodeSurfer, srcSlice, and the slice intersection over 10 files from the program findutils-4.4.2 ordered by the size of slice intersection. Except for a single file, the srcSlice was much closer to the slice intersection than CodeSurfer.

3.6 Scalability of srcSlice

This section demonstrates the scalability of our lightweight slicing approach. We ran srcSlice over the Linux kernel to demonstrate that the approach is effective and scalable for large-scale systems. For a recent version of the Linux kernel, *srcSlice* computed slices for the slicing criterion *(F, M, V)* in 748 seconds and produced a system dictionary of 1,934,557 variables.

The data used in this section originates from slicing 974 versions of the Linux kernel that have been released over the last 17 years (1994 - 2011) with a total lines of code of ~4.4 billion LOC, total slice size is ~2 billion LOC, and with an average slice size relative to system size of 46.0%. The studied Linux versions are identified and ordered by their release date and sequence number (e.g., 1.1, 1.2, 1.3, etc.). The dataset and Linux kernel structure is detailed in CHAPTER 4.

The slicing approach builds a slice profile for each individual variable and then combines the output into a complete system dictionary. This allows for efficient use of memory and computation, thus many scalability issues are avoided. Additionally, the parsing of the code from srcML further avoids computationally intensive searches, since the stream reader pulls tokens from input srcML one after another as needed. As such, very large systems can be sliced in a reasonable amount of time. In other words, large increases of system size do not cripple our tool. The first version of the kernel with 166 KLOC takes 7 seconds. Version 2.6.37.1 with ~13 MLOC takes approximately 13 minutes.

We now examine the slice size of our results, as this is considered to be a crucial issue [Binkley, Gold, Harman 2007], and therefore determines the main aspect of the quality of the generated slices. Ideally, we want to generate the smallest correct slice. Any unrelated statements or variables avoided, will improve the quality of the slice.

Since the average slice size relative to LOC is 46.0%, we feel that our results are in a reasonable margin, based on the work by Binkley et al [Binkley, Gold, Harman, Li, Mahdavi 2006; Binkley, Gold, Harman 2007] and the results obtained in the previous chapter.

Furthermore, the results given in Figure 3.8 represent the difference between the system size and the slice size, both measured in LOC, over 974 versions of the Linux kernel. As can be seen the slice size increases proportionally with the system size. Thus, the steady increase in the system's behavior appears to be reflected in the increasing slice size.

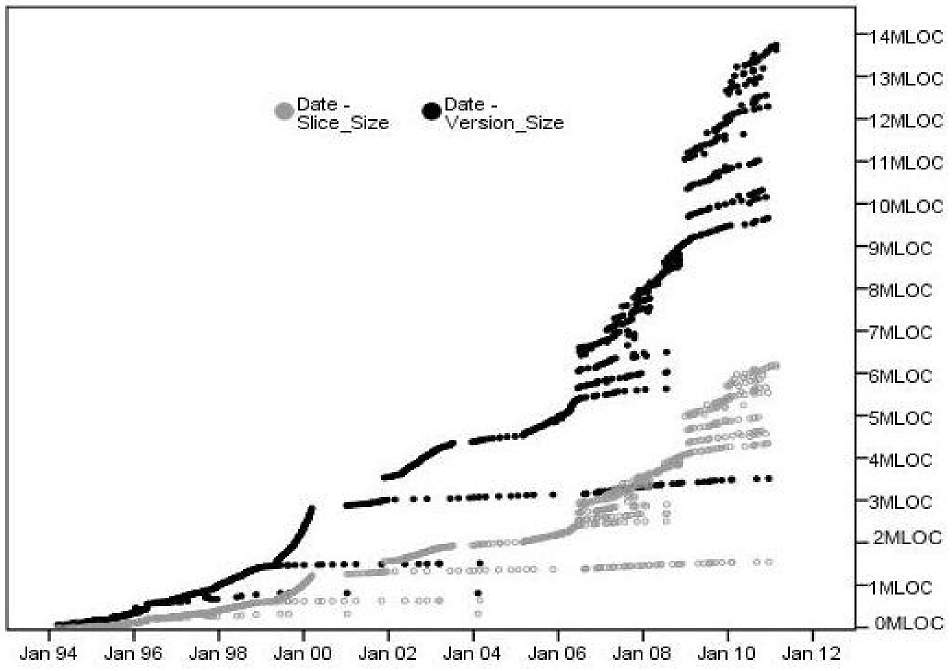


Figure 3.8. A comparison of the size of the srcSlice's slices to the size of the Linux versions measured in MLOC, the x-axis is the version date.

3.7 Related Work

Previously, in CHAPTER 2 we introduced program slicing and some of the key work on that topic. Here we focus on slicing approaches directly related to our approach. We refer interested readers about the PDG based slicing approaches to Tip's and Xu's surveys [Tip 1995] and [Xu, Qian, Zhang, Wu, Chen 2005].

Gallagher et al [Gallagher, Lyle 1991] proposed the definition of decomposition slicing as a maintenance aid in order to capture all the computation related to a given slicing variable. His objective was to define and isolate the parts of the code that are affected by the proposed change or modification of the slicing variable so he could eliminate the need for regression testing.

The decomposition-slicing definition is used by Tonella [Tonella 2003] to construct a concept lattice of decomposition slices. The idea was to combine the decomposition slice graph produced by Gallagher and the lattice program representation model. The concept lattice of decomposition slices is used to support software maintenance by providing information about the computations performed.

A lightweight slicing approach for object-oriented programs using dynamic and static analysis called dependence-cache slicing is proposed in [Ohata, Hirose, Fujii, Inoue 2001]. This approach is based on dynamic data dependence analysis and static control dependence analysis. In general, the slice construction process starts by locating the defined variables, and then extracts the data and control dependencies between statements that include those variables. These dependencies are used to construct the PDG that is traversed backwards for a user given slicing variable.

In the context of maintaining large-scale systems, another lightweight maintenance tool, called *TuringTool*, was proposed by Cordy et al [Cordy, Eliot, Robertson 1990] was designed to support several maintenance tasks using the elision symbols. These symbols are used for viewing large source programs on a small screen by providing source code projection. The importance of this hierarchy view is clear since the user can focus at some point of interest inside the code to any required level of detail. For example, if the debugger is interested only in those statements that affected by the value of a given variable, then only those statements are displayed on the screen. This is the same concept behind using the slicing tools.

Since our approach is scalable in term of time and program size as shown Section 3.6, the need was to evaluate the correctness of our results. Our evaluation criteria was based at the study proposed by Hoffner [Hoffner 1995] in which he discussed several possible aspects to evaluate the performance of proposed slicing approaches. These aspects are the slice size compared to the original size, and the time and space complexities. The author compares a set of dynamic slicing tools, i.e., *Kamkar's* and *Spyder*, with other static tools including the *WPIS*, *FOCUS* and *Schatz's* tool. Only three of the tools support inter-procedural slicing, which are *Kamkar's*, *Spyder* and *WPIS*. In addition, evaluation criterions were established for comparing these different tools. In his evaluation, the slice size was measured using either the number of retrieved statements or the number of vertices in the PDG when comparable approaches are applied with similar languages. Conversely, the author suggests that the code size is the best metric when these approaches handle similar languages. In the context of complexity, the difficulty of the approach is determined by the number of vertices in the intermediate representation models, and as a result the required execution time to complete the slicing process.

Our approach is distinguished from this related work in multiple ways. The method used is not PDG based. There is no graph to traverse or data flow equations to be solved. Only on-the-fly information is retrieved as needed. Unlike most of the others, we do slicing over all the variables inside the system. Our approach supports both system evaluation and comprehension by allowing the user to investigate the program by using the slices at different levels of granularities (e.g., variable, function, file, and system).

3.8 Chapter Summary

A method for efficient and scalable slicing was presented and compared to an existing tool. The results demonstrated that the approach produces fairly accurate slices as compared to an existing tool and is highly scalable. The limitations of the approach are related to deep aliasing and certain array indexing. The *srcSlice* tool was shown to work on a variety of C programs and language features. The approach uses the srcML format and toolkit. As such, it can be applied to incomplete and non-compiling programs. This is particularly useful when external libraries need to be excluded to reduce complexity. Additionally, this feature is very useful for adaptive maintenance tasks involving new features or new API/libraries.

In practice we see the usefulness of this and similar lightweight approaches being as a quick-check mechanism rather than a replacement for more heavyweight (and hopefully more accurate) slicing tools. That is, developers can use this approach to judge if it is prudent to expend the time and money to run a more rigorous analysis on a large software system.

Moreover, the ability to slice multiple versions of large programs in a very short amount of time also opens up new avenues of research. We can investigate how system slices change over the entire history of a large software system, and how slices reflect different types of changes occurring in a system possibly identifying refactoring changes [Pan, Kim, Whitehead 2006; Zhang, Gupta, Gupta 2007]. With current tools this is impractical.

In the future, we plan to also investigate metrics based on program slicing in the context of coupling and cohesion. We also plan to make the *srcSlice* tool part of our srcML toolkit. Currently, *srcSlice* also works for C++ programs but more evaluation is needed for a number of language features.

CHAPTER 4

A CHARACTERIZATION OF MAINTENANCE ACTIVITIES IN LINUX KERNEL USING SLICE-BASED METRICS

This chapter provides empirical support for investigating different possible slice-based metrics. These metrics are evaluated by using them to characterize and identify different types of maintenance activities, (i.e., *corrective*, *perfective*, and *adaptive*). The metrics are also used to help verify *Lehman's laws* of software evolution [Lehman 1980; Lehman 1996].

To address this we first use the information of the forward slices generated from the Linux kernel to investigate and calculate different slice-based metrics (Section 5.2). We then use these metrics to build (Section 5.6) and validate (Section 5.7) a slice-based estimation approach for maintenance effort using statistical tests. The assumption here is that the effort estimation is a model of code change and program slicers are valuable tool in determining the effects. However, before using source-code slicing to measure source-code changes, we should address the following maintenance research question:

- **RQ1:** *Do the slice-based metrics reflect the different maintenance activities?*

Before using statistical tests to build a maintenance-effort estimation model, the results of statistical tests require precise quantifiable definitions for Lehman's laws of software evolution. However, the results of such statistical tests depend on the test and dataset used. Therefore, we address the following software evolution research question:

- **RQ2:** *Are Lehman's laws of software evolution supported using the slice-based metrics?*

In order to answer the above two research questions, we conducted the following two analytical studies:

1. **Analysis of Lehman's laws:**

To study the evolution of the Linux kernel in the slicing context, we plot various potential slice-based metrics as a function of time, as suggested by [Godfrey, Qiang 2000]. Then we use a simple visual inspection to comment on observed patterns. Our objective is to examine weather Lehman's laws are reflected in the evolution of the Linux kernel based on calculated slice-based metrics.

Even though, the growth of the Linux kernel was studied in the literature by [Godfrey, Qiang 2000] who studied the growth of the Linux kernel over its first six years (1994 - 2000). And by Israeli et al [Israeli, Feitelson 2010] who verified Godfrey's results using a much larger data set, i.e., he examined the growth of the Linux kernel over its first 14 years (1994 - 2008) with a total of 810 releases. However, in our work we want to verify these results using a larger data set (1994 - 2011) and using different metrics that have not been used before in this context, e.g., slice-based metrics.

2. **Analysis of maintenance activities:**

To establish whether slice-based metrics can be used to distinguish between the different types of maintenance activities, we will collect data and calculate various slice-based metrics for different versions and groups of directories. For example, we will compare consecutive stable versions of Linux kernel and consecutive development versions to ascertain whether slice-based metrics can be used to differentiate between corrective and perfective maintenance activities. These metrics are then compared to traditional measures of code effort, e.g., LOC.

Our goal is to use the many versions of the Linux kernel to characterize the different types of maintenance activities. Based on the structure of the Linux releases (as will be discussed in Section 4.1.1), we expect corrective maintenance to be reflected in consecutive versions of stable kernels.

Perfective maintenance (as it pertains to the addition of new features) is expected to be reflected in consecutive versions of development kernels. Again, this matches the structure of the Linux versions; we can assume that consecutive versions in the development kernels will be perfective.

Adaptive maintenance is expected also to be reflected in development versions. However we expect it to be reflected in specific directories in the development kernels. For example, those directories encapsulate most of the interactions of the system with its environment, in other words, the location for code that handles interaction with the environment.

Preventative maintenance is much harder to be identified, since is related to code reorganization. We expect this type is reflected in the events that partition, delete, or move source code files mainly in the development kernels. The assumption here that this type of maintenance activity is related directly to code reorganization and therefore could be identified, for example, by the changes in the number of directories.

The data used in this chapter originates from slicing 974 versions of the Linux kernel that have been released over the last 17 years with a total of ~4.4 billion LOC. We examined all the Linux releases from March 1994 to February 2011. This includes 159 stable versions (1.0, 1.2, 2.0, 2.2, and 2.4), 397 development versions (1.1, 1.3, 2.1, 2.3, and 2.5), and 418 versions of 2.6 (up to release 2.6.37.1). This represents a significant extension of the work of [Godfrey, Qiang 2000] and [Israeli, Feitelson 2010], who's cutoff date was January 2000 and August 2008, respectively.

4.1 Background

In this section we provide some background regarding the Linux kernel, Lehman's laws of software evolution, and software maintenance activities.

4.1.1 Structure of the Linux kernel

The GNU Linux kernel operating system was developed originally and announced on the Internet by Linus Torvalds in August 1991. After that, a community of developers developed and released the first production version in March 1994.

The studied Linux versions are identified and ordered by their release date and sequence number. The Linux versions are classified as stable and development versions. Each major version includes several releases identified with either a three or four digit numbering scheme. The first digit represents the generation, i.e., Linux has three generations, initially with generation 1 released in 1994, generation 2 released in 1996, and generation 3 started in 2011 (not part of the dataset). The second digit represents the major kernel versions either even or odd. Up until major version 2.4 even digits (i.e., 1.0, 1.2, 2.0, 2.2, and 2.4) corresponded to stable versions, whereas odd numbers (i.e., 1.1, 1.3, 2.1, 2.3, and 2.5) corresponded to development versions, those versions used to test new drivers and features which leading up to the next stable version.

The third digit is the minor kernel version number used to distinguish new releases in both stable and development versions. That is, new minor numbers of stable versions hypothetically included only security patches and bug fixes, whereas new minor numbers of development versions included new (however not fully tested) feature/functionality. However, in August 2004 this numbering scheme was changed affecting all the versions released after this date. A fourth digit number was added starting with version 2.6.8.1, after that the third number in a version indicates the development of new functionality, and the presence of a fourth number represents bug fixes and/or security patches [Koren 2006]. As a result, kernel 2.6 combines stable and development versions. But, it is actually more like a development version given that new feature is released with relatively little testing [Israeli, Feitelson 2010].

By analyzing the Linux kernel, the kernel sources are arranged in several subdirectories, the main ones are as follows:

- **arch:** this subdirectory contains all the kernel source code that specified for different special processor architectures, in which each one supported by different subdirectories (e.g., *alpha*, *i386*, *mips*, *sparc*, etc.).
- **init:** contains the initialization source code for the kernel.
- **mm:** contains all the memory management source code for the kernel.

- **fs:** contains the file system source code, such as each supported file system has one subdirectory. For example, Linux supported many file systems such as: *nfs*, *ntfs*, *fat*, etc.
- **net:** contains the networking source code of the kernel.
- **lib:** contains the library source code of the kernel.
- **ipc:** contains the inter-process communication source code.
- **drivers:** this directory contains source code of all the system's device drivers. This directory is further divided into subdirectories depending on the source code of device driver it contains.
- **kernel:** this directory is one of the most important directories in kernel. It contains the main kernel source code.
- **include:** contains the include files (e.g., header files) needed to build the kernel source code. Along with the *kernel* directory this directory also is very important for kernel development.

A summary of the dataset from the first version 1.0.0 released in March 13, 1994 up to version 2.6.37.1 released in February 17, 2011 is given in Table 4.1. Also included are statistics related for each version, including the number of source code files in the system (i.e., ".h" and ".c" files), the LOC of each version, the slice size, the slicing time, and the program/slice ratio. The total slice size is ~2 billion LOC, with an average slice size relative to LOC of 46.0%.

Table 4.1. Linux Version data, 11 major versions with 974 individual versions, (RSN) = release sequence number, slice size measured in LOC, slicing time measured in seconds, (%) is the slice size relative to LOC, (R) = number of releases in each major version, (S) = stable version, (D) = development version.

Version	Version Type	RSN	R	Size		Slice Size	%	Date	Slicing Time
				Files	LOC				
1.0	S	1.0.0	1	487	166,144	83,891	50.5 %	3/13/94	7
1.1	D	1.1.0	36	487	165,768	83,696	50.5 %	4/6/94	7
		1.1.95		766	283,492	134,458	47.4 %	3/2/95	12
1.2	S	1.2.0	14	766	283,522	134,446	47.4 %	3/7/95	12
		1.2.13		765	287,104	136,293	47.5 %	8/2/95	12
1.3	D	1.3.0	100	832	312,214	150,199	48.1 %	6/12/95	14
		1.3.100		1,551	655,774	305,404	46.6 %	5/10/96	29
2.0	S	2.0.0	41	1,603	677,958	312,861	46.1 %	6/9/96	31
		2.0.40		1,884	913,716	423,324	46.3 %	2/8/04	41
2.1	D	2.1.0	133	1,618	698,027	324,136	46.4 %	9/30/96	31
		2.1.132		3,732	1,515,441	695,857	45.9 %	12/22/98	47
2.2	S	2.2.0	27	3,775	1,611,328	704,428	43.7 %	1/26/99	47
		2.2.26		4,874	1,615,056	741,157	45.9 %	2/24/04	47
2.3	D	2.3.0	52	3,857	1,621,440	744,060	45.9 %	5/11/99	48
		2.3.51		5,450	2,911,206	1,330,584	45.7 %	3/11/00	76
2.4	S	2.4.0	76	6,742	2,978,667	1,361,262	45.7 %	1/4/01	79
		2.4.37.11		10,629	3,614,387	1,650,357	45.7 %	12/18/10	177
2.5	D	2.5.0	76	8,196	3,633,072	1,658,854	45.7 %	11/23/01	180
		2.5.75		12,412	4,452,342	2,031,419	45.6 %	7/10/03	301
2.6	S&D	2.6.0	418	12,424	4,476,542	2,042,424	45.6 %	12/18/03	309
		2.6.37.1		28,766	13,840,130	6,300,536	45.5 %	2/17/11	748

4.1.2 Categories of Software Maintenance

In software engineering the software evolution dynamics is the study of the processes that include both developing the software initially then updating it for various reasons. This updating process is called software maintenance, which defined in the literature as one of the five primary life cycle processes that may be performed during the life cycle of software. The software maintenance represents all the modification activities (e.g., source code change) made to the software product after its first installation that follows the operational development process. The ratio of these changes is generally accepted by the software community to be around 70% [Lehman 1980; Sage, Palmer 1990; Israeli, Feitelson 2010]. Generally during evolution, these maintenance activities are implemented by modifying existing components and adding new components to the system (e.g., fixing a bug or adding a new feature).

The system maintenance activities was categorized initially by Swanson in 1976 [Swanson 1976] into three categories such as: *corrective*, *adaptive*, and *perfective*. Later these categories have been updated to four categories and presented in ISO/IEC 14764 [ISO/IEC 1999] standard and [Chapin, Hale, Kham, Ramil, Tan 2001] as follow:

1. *Corrective maintenance*: deals with correction of discovered problems that prevent the software to be at the operational state and meets its requirements.
2. *Adaptive maintenance*: adapting the software after delivery to changing internal needs within the original environment so that it operates in a different environment.
3. *Perfective maintenance*: this type of maintenance to add to or modify the system's functionality after delivery, for example the performance or reliability of the system may have to be improved by modifying the software to be able to meet external evolving user needs (e.g., improve performance or maintainability, adding new features).

4. *Preventative maintenance*: this type of maintenance deals with the modification of software after delivery to detect and correct faults before they become effective faults.

An important issue is the maintenance effort spent in the different types of maintenance activities. Studies [Swanson 1976; Sage, Palmer 1990; Israeli, Feitelson 2010] show that the total maintenance effort spent among these four types of maintenance activities is distributed as follows (taking into consideration that the maintenance effort is influenced with a variety of factors, such as system age, system quality).

- 55% Corrective,
- 20% Adaptive,
- 20% Perfective, and
- 5% Preventative.

In CHAPTER 5, we will attempt to estimate the effort involved with a new Linux version using a variety of different slice-based and traditional code-based metrics. One group consists of measures of source code changes, including the slice changes, number of files and functions contain modified slices, and the relative slice coverage of the program size. These measures are based on the Linux source-code slices. Yet other types of metrics are time spent in development/maintenance process, the rate of new versions, lines of code, and number of files. These measures will be used also and their results will be compared with slice-based metrics.

Here in this chapter, our objective is to be able to specify whether maintenance activity is corrective, adaptive, perfective, or preventative by examining changes in an appropriate slice-based and traditional code-based metrics. Particularly, we will compare consecutive stable versions of Linux and consecutive development versions to establish whether slice-based and usual code-based metrics can be used to distinguish between corrective and perfective maintenance activities, respectively.

4.1.3 Lehman's Laws of Software Evolution

The laws of software evolution are a set of empirically derived observations that were originally proposed by Lehman in 1974. The laws, their early development, and detailed discussion of their nature and implications are discussed in details in [Lehman 1980; Lehman, Ramil, Wernick, Perry, Turski 1997]. The laws and their implications are explained in Table 4.2.

Lehman's laws of software evolution explain the forces that driving new developments on the *E-type* software system on one hand, and the forces that slow down the development progress on the other hand. Lehman used the keyword E-type to refer to the real-world systems. He describe these systems as they evolving in a way that cannot be specified in advance, however they are modified in response to a new user requirements or change requests.

The first law (*Continuing Change*) summarizes the observation that large systems are never completed; they just continue to evolve to stay useful. As the system evolving changes, these changes causing growth in the system size (the sixth law *Continuing Growth*), and its structure tends to be more complex (the second law as *Increasing*

Complexity) unless work is done to reduce it. However, the process of evolution is prompted when the user perceives a decrease in quality (the seventh law *Declining Quality*). In addition, the laws also state that the development changes have effect within an environment that forces stability (its rate of development statistically invariant) and a rate of change that permits the organization to keep up (the fourth and fifth laws of *Conservation of Organizational Stability* and *Conservation of Familiarity*, respectively). Finally, in order to maintain the familiarity and stability, the evolution process of the system should be a self regulating process and must be treated as a feedback system incorporating multi-level, multi-agent,multi-loop feedback systems (the thired and eighth laws of *Self Regulation* and *Feedback System*, respectively).

Since, these laws are believed to observe all the changes during the software evolution process, empirical studies were provided to support the laws using the changes of the system size over time [Lehman, Ramil, Wernick, Perry, Turski 1997; Lehman, Perry, Ramil 1998; Lehman, Ramil, Perry 1998]. However, some empirical observations of studying the development of open-source systems appear to challenge some of Lehman’s laws [Godfrey, Qiang 2000; Godfrey, Qiang 2001; Capiluppi, Lago, Morisio 2003] since the laws are believed to apply mainly to strictly managed and closed-source code systems [Israeli, Feitelson 2010]. In addition, there is no broad support exists for all the laws across different empirical studies of open-source systems.

Unlike all of others we will try to validate Lehman’s laws using slice-based metrics, and using the largest number of Linux versions over a longer span time.

Table 4.2. The Lehman's laws of software evolution as proposed by Lehman [Lehman, Ramil, Wernick, Perry, Turski 1997].

NO.	Law	Statement
1	Continuing Change	An E-type system must be continually adapted; else it becomes progressively less satisfactory in use.
2	Increasing Complexity	As an E-type system is changed its complexity increases and becomes more difficult to evolve unless work is done to maintain or reduce the complexity.
3	Self Regulation	Global E-type system evolution is feedback regulated.
4	Conservation of Organizational Stability	The work rate of an organization evolving an E-type software system tends to be constant over the operational lifetime of that system or phases of that lifetime.
5	Conservation of Familiarity	In general, the incremental growth (growth rate trend) of E-type systems is constrained by the need to maintain familiarity.
6	Continuing Growth	The functional capability of E-type systems must be continually enhanced to maintain user satisfaction over system lifetime.
7	Declining Quality	Unless rigorously adapted and evolved to take into account changes in the operational environment, the quality of an E-type system will appear to be declining.
8	Feedback System	E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems.

4.2 Measurements of the Linux kernel

Most of existing studies on software evolution use different metrics to demonstrate their point. In general, there are process metrics that measures the development process (e.g., methodology, development time, the experience of the programmers), and product metrics that measures the system at any stage of its development (e.g., size of the program). Next we will focus on product metrics, specifically the slice-based metrics. We will display and discuss our slice-based results, also compare them to those of traditional code-based measures.

To observe the changes in traditional metric values and slice-based metric values from one version to the next, we typically graph the results for all 974 versions. For each of the figures, and throughout this chapter, the *x-axis* of the graphs represents the release date, and the indication near the lines shows the corresponding Linux kernel version. We make a distinction between the stable versions (1.0, 1.2, 2.0, 2.2, and 2.4), the development versions (1.1, 1.3, 2.1, 2.3, and 2.5), and the 2.6 series using different colors.

4.2.1 System Size

The size of a system could be measured in different ways. We start by considering the number of lines of code (LOC), and the number of files. A line of code (LOC) is one of the most used metric in the literature. However, the problem is the definition, such as, is there a need to collect the commented and empty lines to the code statements. On one hand, we are interested with the code itself, so it seems that only the executable statements interest us. On the other hand, the number of comments affects the understandability of the source code and consequently its maintainability and the empty lines affect the readability of the source code.

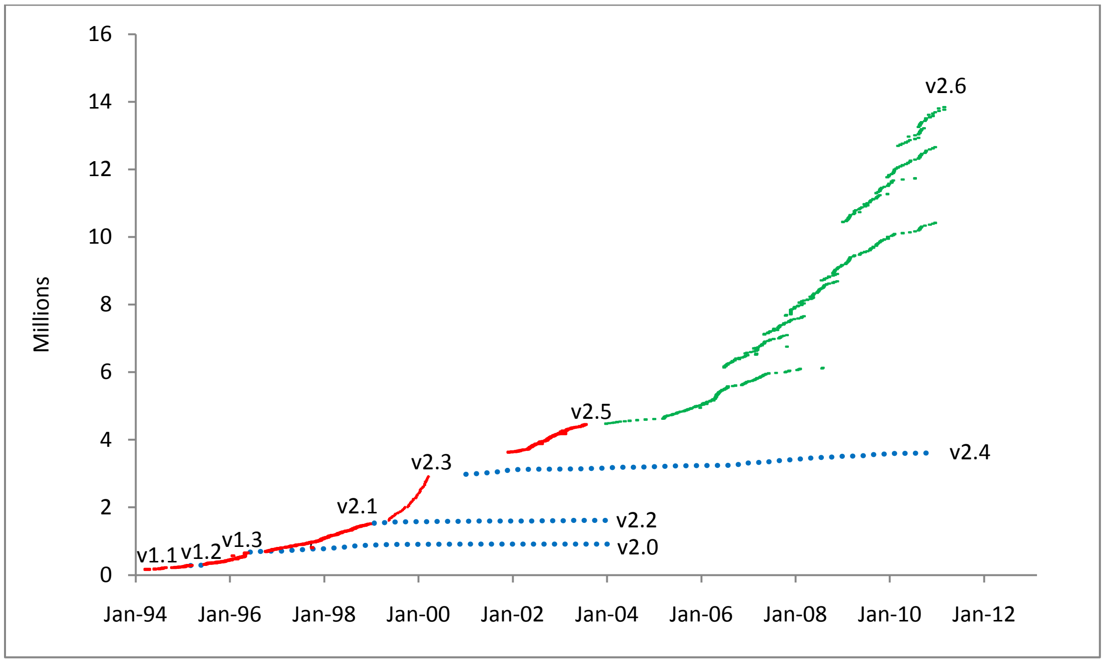


Figure 4.1. The growth of LOC in 974 versions of Linux kernel.

Figure 4.1 shows the growth in the code lines as total lines in the file (measured using *wc –l* Linux command). Based on the structure of the Linux kernel as explained in Section 4.1.1, we see here that most of the growth occurs in the development versions, whereas the stable versions are usually steady, except for some increase at the beginning. For example, the development versions 1.2, 2.0, 2.2, and 2.4 are exhibits an increase at the beginning years of development, and this is particularly noticeable in version 2.2. This extraordinary behavior in the first years of development of stable versions will be discussed in detail below. The version 2.6 includes a combination of the development and stable versions. For each release (new third digit) the number of lines of code is constant as in stable versions, but it grows between them as in development versions.

An interesting observation is that the pattern of growth sometimes exhibits large jumps in select points. For example, a closer investigation of Figure 4.1 (see Figure 6.2 in APPENDIX C for more details about the growth of each version individually) we can distinguish two large jumps as follows: the first jump in version 1.2 (from 284919 to 286632 LOC) between release 1.2.7 and release 1.2.8. The second jump in version 1.3 (from 574310 to 641932) between releases 1.3.92 and 1.3.93. Both jumps will be explained next when we consider the growth using the number of files metric.

Figure 4.2 shows the growth in number of files in Linux (for more details of this figure see Figure 6.1 in APPENDIX C). Overall, we see a similar growth trend to that of LOC. That is, they are increasing over time in development versions and usually steady in stable versions, with the same special cases of initial growth in stable versions. However, some details are different. For example, there is a little increase in the beginning in versions 2.0, 2.2, and 2.4. In addition, the large jump of LOC in version 1.2 is disappeared for the number of files; instead the growth trend is consistently stable with just one file decrease in release 1.2.7. That is, the number of files from release 1.2.0 to release 1.2.6 is always consistent and equal to 766 files, then after that is decrease to 765 (release 1.2.7) files for all subsequent versions.

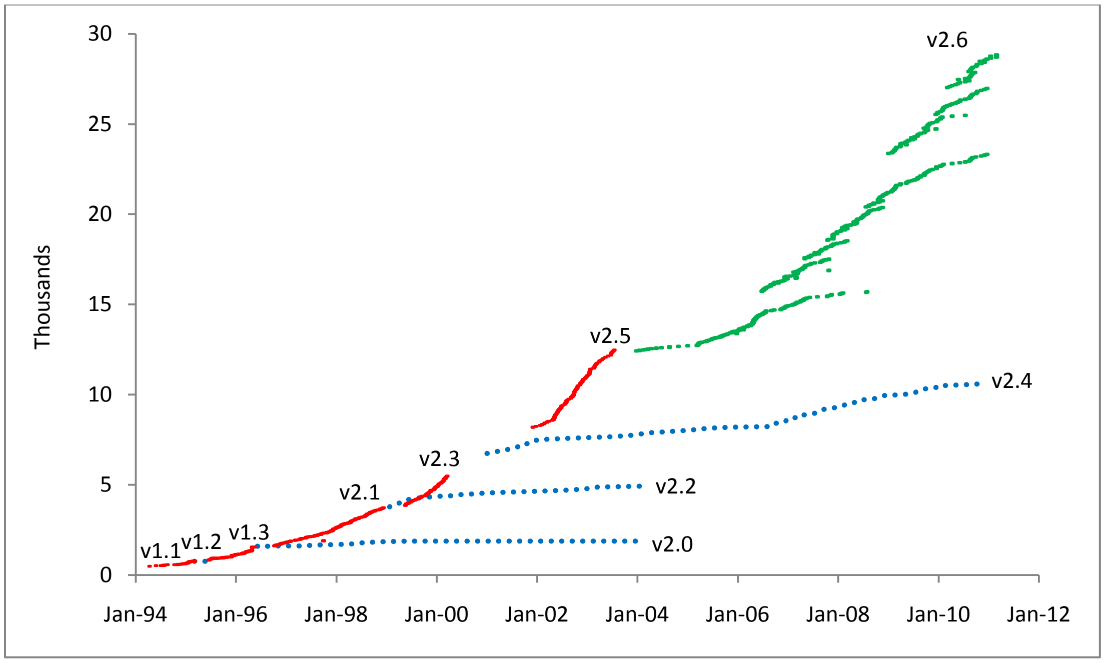


Figure 4.2. The growth of the number of files in Linux.

The best way to explain this phenomenon by combining this information with the general LOC (Figure 4.1), specifically, by relating the jump occurs in the LOC value inside release 1.2.8 to the removed file in the previous (1.2.7) release. We find the following: the large increase in the LOC value in release 1.2.8 corresponding directly to the deleted file in release 1.2.7. This is an example of the effect of reorganization of the code since it was caused by the deletion of only one source file and the lines of source code (LOC) is not deleted.

The second jump in the LOC value in version 1.3 supports the evidence of the above phenomenon. The LOC's large increase in release 1.3.93 caused the large increase in the number of files in release 1.3.94 (from 1389 to 1544 files). In other words, the too much LOC increased in release 1.3.93 necessitate developers to reorganize this large amount of code by adding 155 new files to release 1.3.94.

Another interested observation in the growth of number of files that the growth trend in versions 2.2 and 2.4 exhibits significant growth in the number of files than LOC. We can see that the growth in the number of files in both versions was relatively larger than that of LOC, meaning that the growth in number of files is continually increases as much code is added first then this code is reorganized by adding new files. However, as our aggregate metrics results demonstrate in the next section, we can obtain several hints that indicate the above results. The average LOC and Slice size per file results indicate that the growth trend of versions 2.2 and 2.4 is decreasing over time, so that the growth of number of files is larger than the growth of LOC and slice size.

In the growth of number of files, we also see the same behavior of version 2.6 as in LOC. Each minor version growth is somehow steady, but all together there is a super-linear growth trend (see Figure 4.3).

4.2.2 Slice Size

The literature is full of software metrics that quantifies the complexity of a system. However, many existing software metrics are computed only using syntactic information of the code and use that to model semantic information. For example, cyclomatic complexity [McCabe 1976] quantifies the semantic complexity of a program as the number of independent paths in the program's control flow graph. Semantic information is much more difficult to derive and model. For example, a semantic change in one function might create a ripple effect among other functions. In a semantic complexity context, program slicers often applied and are helpful tool in determining side effects.

By slicing the versions of the Linux kernel we observed that in general the growth of the slice size over time has clearly related trend about the maintenance activity being made. That is, most of the growth occurs in the development versions, whereas the stable versions are usually steady, except for a little increase at the beginning, they are continue slowly to increase later on.

On overall basis, from Figure 3.8 in Section 3.6 (which compares the Linux source code size with the slice size) it is clear that the growth in the development versions is fairly often larger than in stable versions. However, it is not visually noticeable what the growth pattern inside each individual version. For this reason, Figure 4.5 shows the growth in the slice size for 10 major versions (from version 1.1 to version 2.6) of the Linux kernel with a total of 974 releases (version 1.0 is combined with version 1.1 since it contains just one release). For each individual version in this graph we plot the slice size as a function of time as we did in both LOC and number of files.

In general, the results as shown in Figure 4.5 indicate that the slice size grows with time, and the pattern of growth is very similar to that of the number of files except for version 1.2 that has a growth pattern similar to that of corresponding version in LOC. In version 1.2 there is a slow increase at the beginning then a jump in the slice size in release 1.2.8 (853 LOC increased). This is happened as mentioned above due to a deletion of one file in the previous release 1.2.7.

The slice size as a semantic complexity metric shows that there is an increase in the complexity over time. However, the growth trend is more sable and less volatile than of the growth in LOC. For example, in version 1.3 (specifically release 1.3.58) the large increase in the LOC value ~100 KLOC is corresponding to just an increase in the slice size of ~ 4500 LOC, thus the slice size relative to LOC (slice coverage metric explained next) is reduced from 45.7% in release 1.3.57 to 38.2% in release 1.3.58.

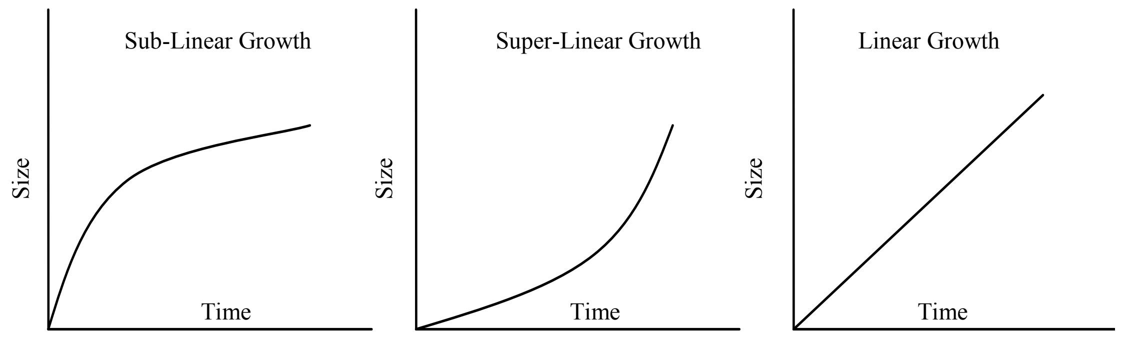


Figure 4.3. Three different types of growth rate.

An interesting observation captured in literature studies [Lehman, Ramil, Wernick, Perry, Turski 1997; Godfrey, Qiang 2000] is that the typical rate growth of the Linux kernel is a sub-linear (see Figure 4.3). However, Godfrey et al [Godfrey, Qiang 2000] mentioned that the Linux grows is at a super-linear rate.

In order to characterize the shape of the growth in the slice size (e.g., linear, super-linear, etc.), we use the two growth models shown in Table 4.3. We can see that using the R^2 value (definition in the next chapter) that the best fitting model for the development versions (1.1, 1.3, 2.1, and 2.3) is the exponential model, also versions 2.4 and 2.6 since as we discussed above these versions start with new numbering schema, thus those two versions include the two types of versions (stable and development). In contrast, the trend growth at the stable versions (1.2, 2.0, and 2.2) is best fitted by the linear model, the exception for version 2.5 that is considered a development version. However, the overall growth of the slice size in Linux kernel is shown to follow the super-linear growth as shown in Figure 4.4.

In both models, the dependent variable represents the slice size, and the independent variable represents the version release date.

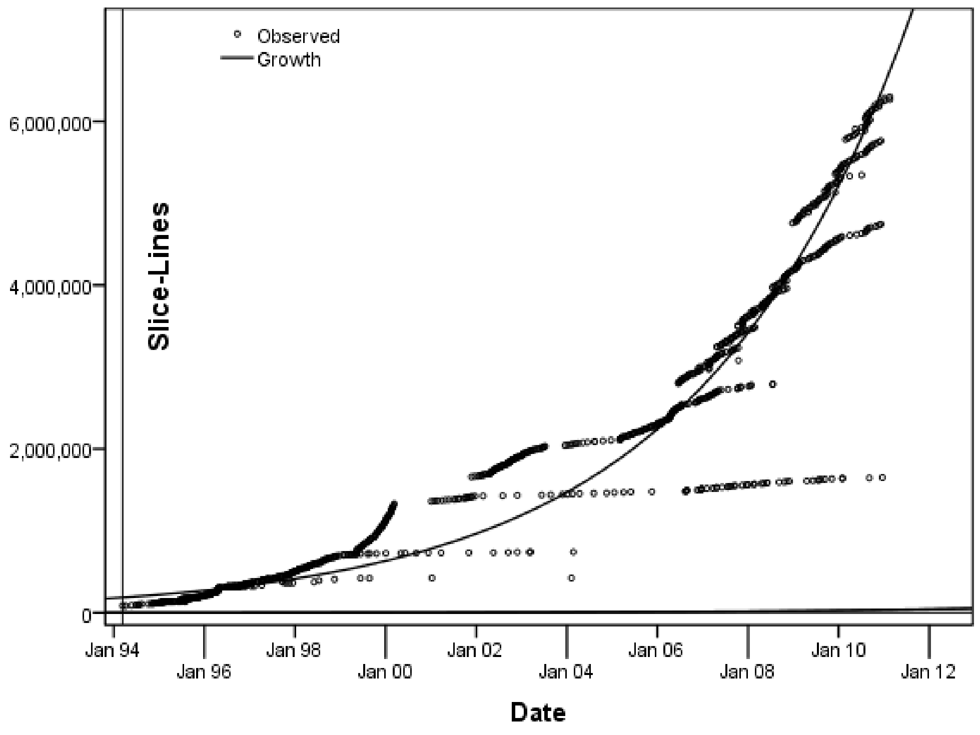
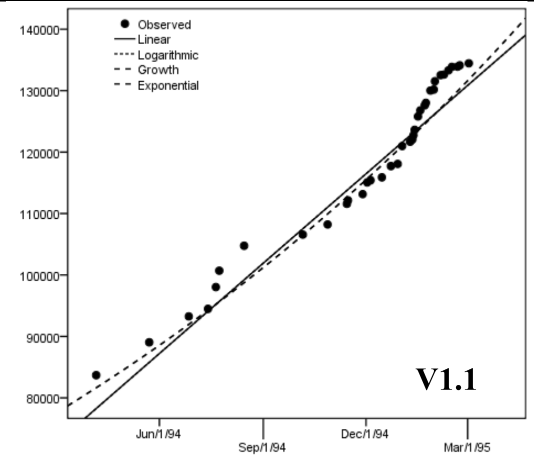
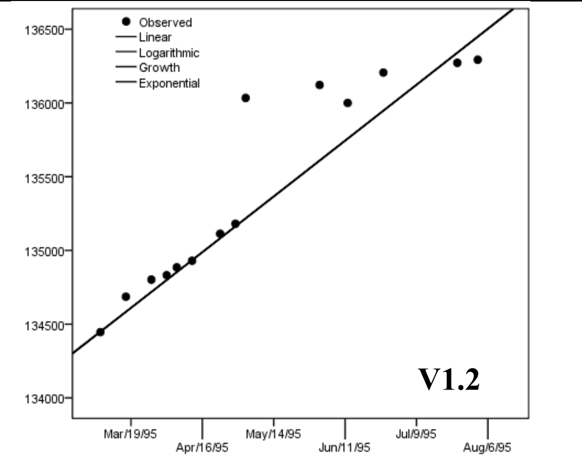
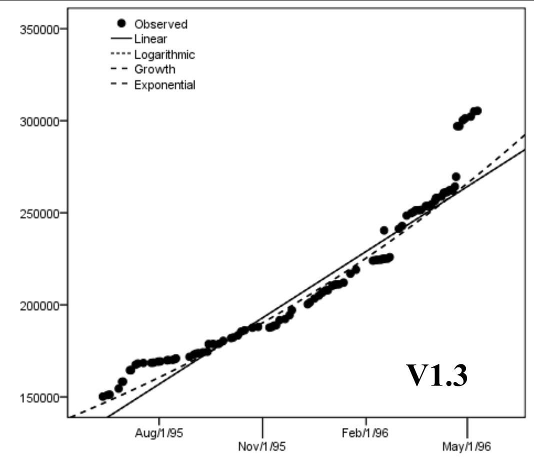
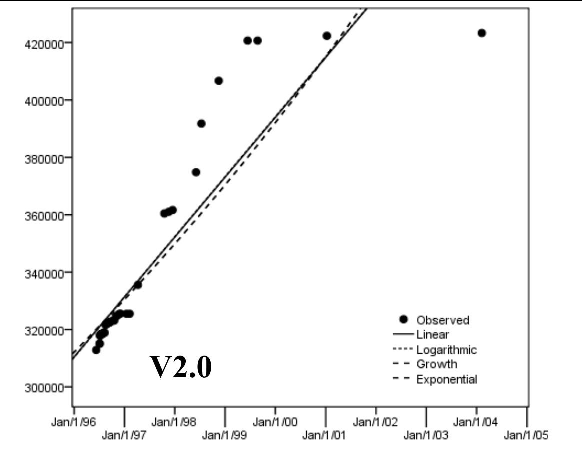


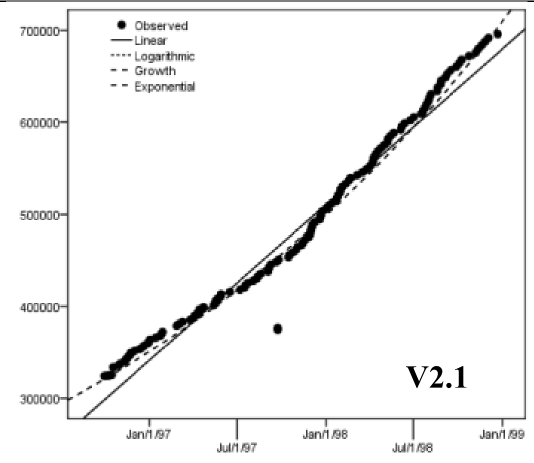
Figure 4.4. Growth rate in Linux kernel using slice lines.

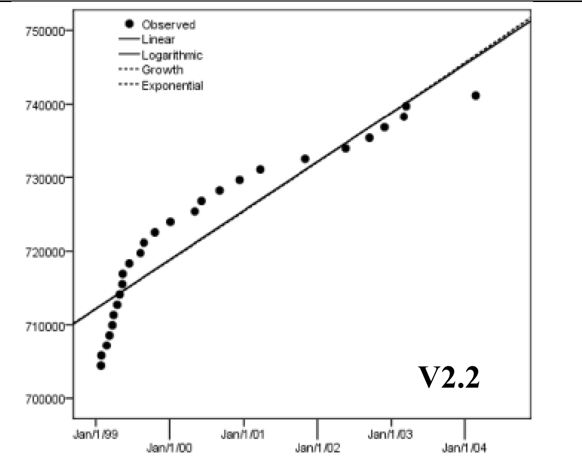

● Observed
Linear
Logarithmic
Growth
Exponential
140000
130000
120000
110000
100000
90000
80000
Jun/1/94
Sep/1/94
Dec/1/94
Mar/1/95
V1.1


● Observed
Linear
Logarithmic
Growth
Exponential
136500
136000
135500
135000
134500
134000
Mar/19/95
Apr/16/95
May/14/95
Jun/11/95
Jul/9/95
Aug/6/95
V1.2


● Observed
Linear
Logarithmic
Growth
Exponential
350000
300000
250000
200000
150000
Aug/1/95
Nov/1/95
Feb/1/96
May/1/96
V1.3


420000
400000
380000
360000
340000
320000
300000
● Observed
Linear
Logarithmic
Growth
Exponential
V2.0
Jan/1/96
Jan/1/97
Jan/1/98
Jan/1/99
Jan/1/00
Jan/1/01
Jan/1/02
Jan/1/03
Jan/1/04
Jan/1/05


● Observed
Linear
Logarithmic
Growth
Exponential
700000
600000
500000
400000
300000
Jan/1/97
Jul/1/97
Jan/1/98
Jul/1/98
Jan/1/99
V2.1


● Observed
Linear
Logarithmic
Growth
Exponential
750000
740000
730000
720000
710000
700000
Jan/1/99
Jan/1/00
Jan/1/01
Jan/1/02
Jan/1/03
Jan/1/04
V2.2

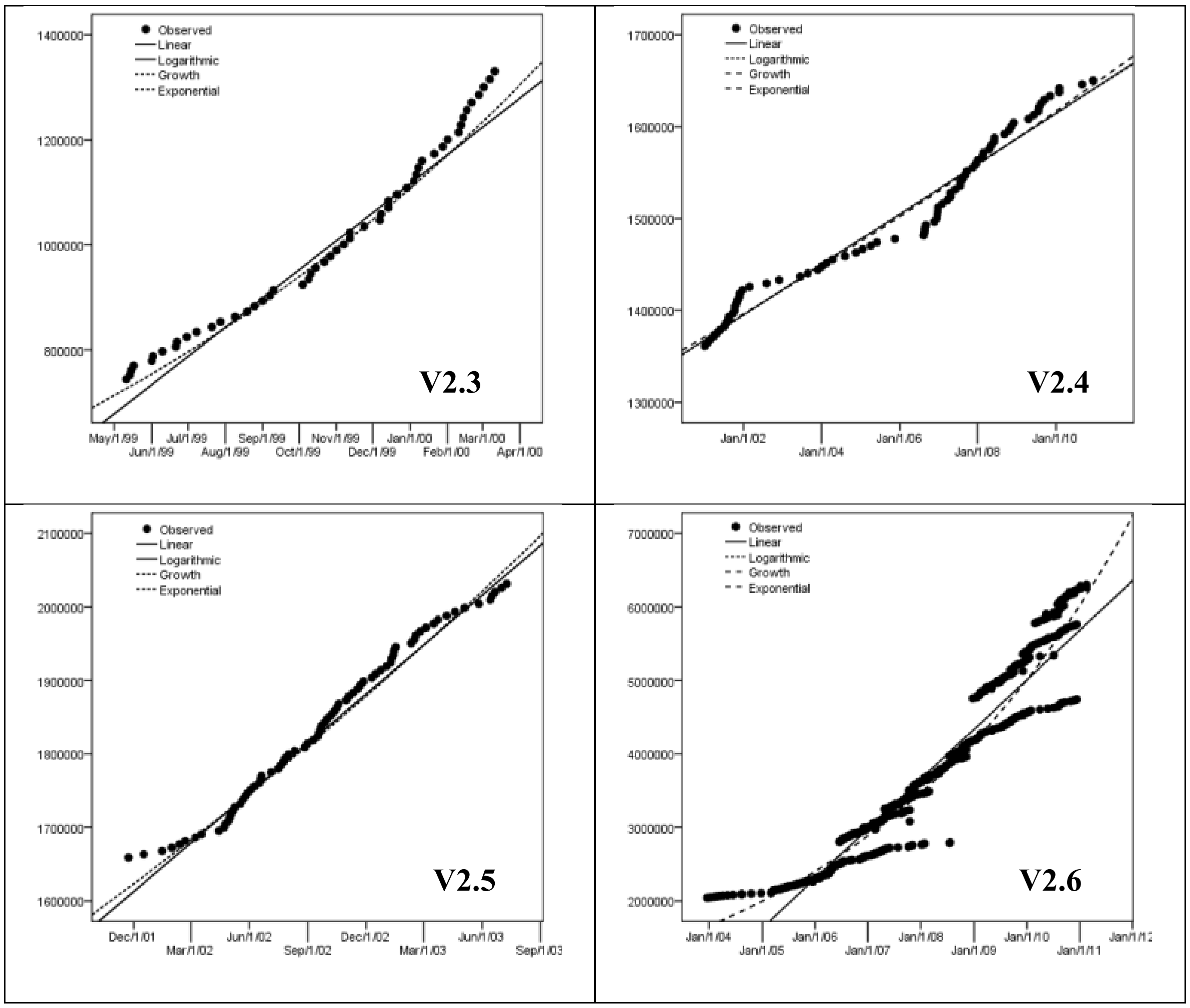


Figure 4.5. The Linux-kernel slice size measured in LOC, 10 major versions.

Next, we calculated LOC and slice size at the file level. That is, averaging the values over the number of files. When comparing the averages per file we see the following picture: the most obvious observation regarding the average size of files as shown in Figure 4.6 is that the average sizes are similar in both LOC and slice size, but with relatively large fluctuations and jumps in LOC. That is, the average size of files is somewhat volatile and more striking using LOC, but much smaller and more stable in slice size.

Table 4.3. Two models summary for 10 major versions with 974 subversions of the Linux kernel, dependent variable is the slice size, independent variable is the version release date.

Linux kernel Version	Models R^2 values		P-value
	Linear	Exponential	
1.1	0.95	**0.97**	<0.001
1.2	**0.95**	0.94	<0.001
1.3	0.92	**0.96**	<0.001
2.0	**0.81**	0.80	<0.001
2.1	0.97	**0.98**	<0.001
2.2	**0.87**	0.86	<0.001
2.3	0.97	**0.99**	<0.001
2.4	0.96	**0.97**	<0.001
2.5	**0.99**	0.98	<0.001
2.6	0.91	**0.94**	<0.001

Generally, by comparing this information (Figure 4.6) with the growth in LOC (Figure 4.1) and the growth in number of files (Figure 4.2), we find the following. The LOC and number of files were increasing the whole time, and increasing between versions, and here we find also decrease, a much more volatile growth-trend and a sometimes a higher increase for stable versions. In common, the growth is not smooth and the changes between successive versions could be large, since the number of files and LOC added is different in each version, also there is an update for existing code.

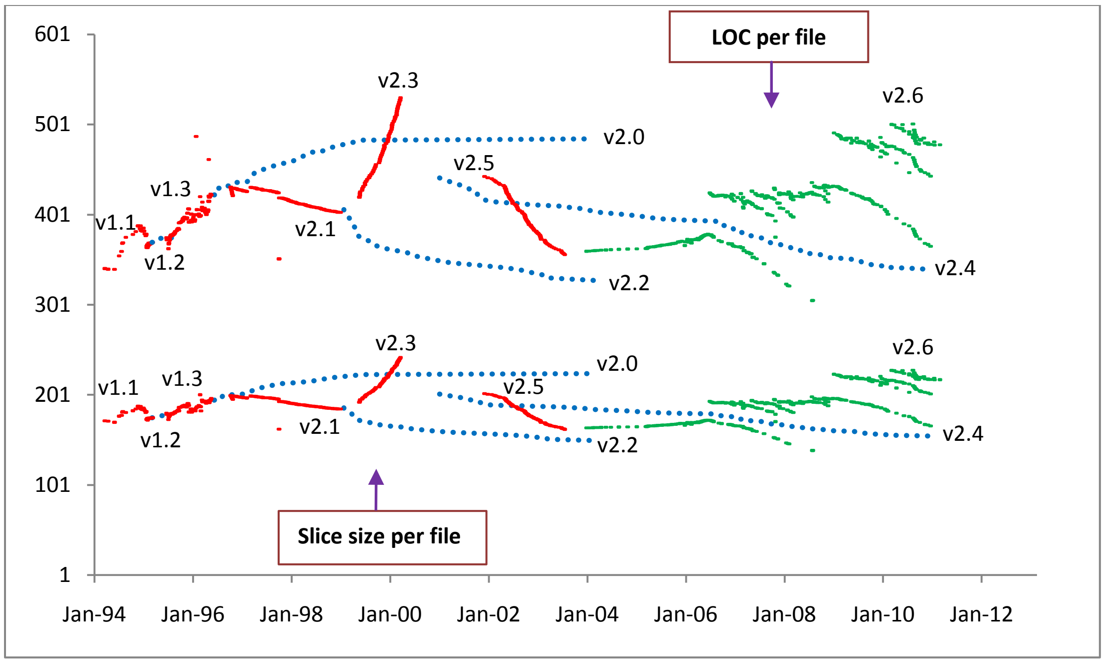


Figure 4.6. The average LOC and slice size per file.

Note that the graph of average LOC and slice size per file (Figure 4.6) is generally more volatile than those of total LOC (Figure 4.1), or number of files (Figure 4.2). The reason is that the average LOC and the average slice size are the quotient of two other metrics (e.g., average slice size per file is equal to: slice size / number of files), and while all of these metrics generally grow, the relative rates of growth may fluctuate. However, there are some patterns do emerge.

For example, an interesting finding is that for the stable version series 2.0 the initial growth is much higher than of successive development versions, indicating that the growth in LOC and slice size was larger than in number of files. The meaning of this is that each file, on average, has gained more LOC. In version 2.6, the average LOC or slice size per file (per subversion not overall) is decreasing with time indicating that the number of files is growing faster than the LOC and slice size.

Finally, looking at each of the graphs, LOC and number of files separately exposes only a small part of the evolution story, i.e., we just see an increase and we can say that Linux kernel is constantly growing. However, when we look at the averages we see a noticeable change per file. For example, in version 1.3.58 we see a small increase in number of files, but a very large increase in LOC and thus a large increase in the average LOC per file. The cause of this is addition of only 7 files (from 1157 to 1164) with total of over 100 KLOC (from 463795 in the release 1.3.57 to 567594 in the release 1.3.58); the jump is especially visible when looking at the growth for the LOC per file or LOC.

The slice size as discussed above quantifies the semantic complexity of a program, and the LOC quantifies the syntactic complexity of a program. We calculated the difference between LOCs, and the difference between the slice sizes for each two consecutive versions. Then these differences, which represent the amount of growth on both the LOC and slice size between two versions, are averaging over the time duration (measured in days) between two versions.

From Figure 4.7 and Figure 4.8 we can see that the average LOC and slice size growth is large in version 2.6. However, the growth in other versions is not noticeable since the growth in version 2.6 spread on an extremely large area, thus the graph is very compact and we miss some sharp drops in values because of this. Therefore, we choose to draw both graphs with the average values of LOC and slice size in a logarithmical form as shown in Figure 4.9 and Figure 4.10, respectively, the logarithmical base here is 10.

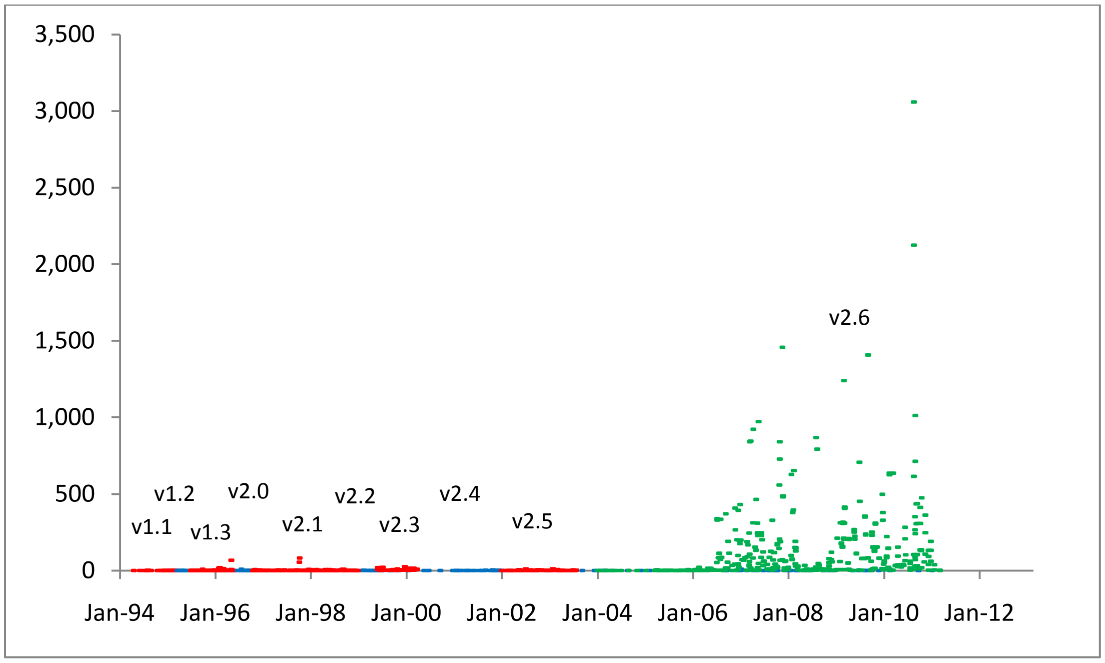


Figure 4.7. The average LOC growth per day measured in KLOC.

The graphs show that the pattern of growth is very similar between LOC and slice size. The results show that the average LOC and slice size growth per day have the highest values in the development versions (1.1, 1.3, 2.1, 2.3, and 2.5) than stable versions (1.2, 2.0, 2.2, and 2.4). The results indicate a pronounced increase and relatively large values over time in the development versions and a decrease, smaller, and more stable in the stable versions. That is, the average LOC and slice size growth per day within development versions tend to have much higher values, meaning the development versions gained more LOC than stable versions.

In version 2.6 the values are fluctuating since the 2.6 series demonstrates both development and stable versions, but the average values for both LOC and slice size are always the highest values.

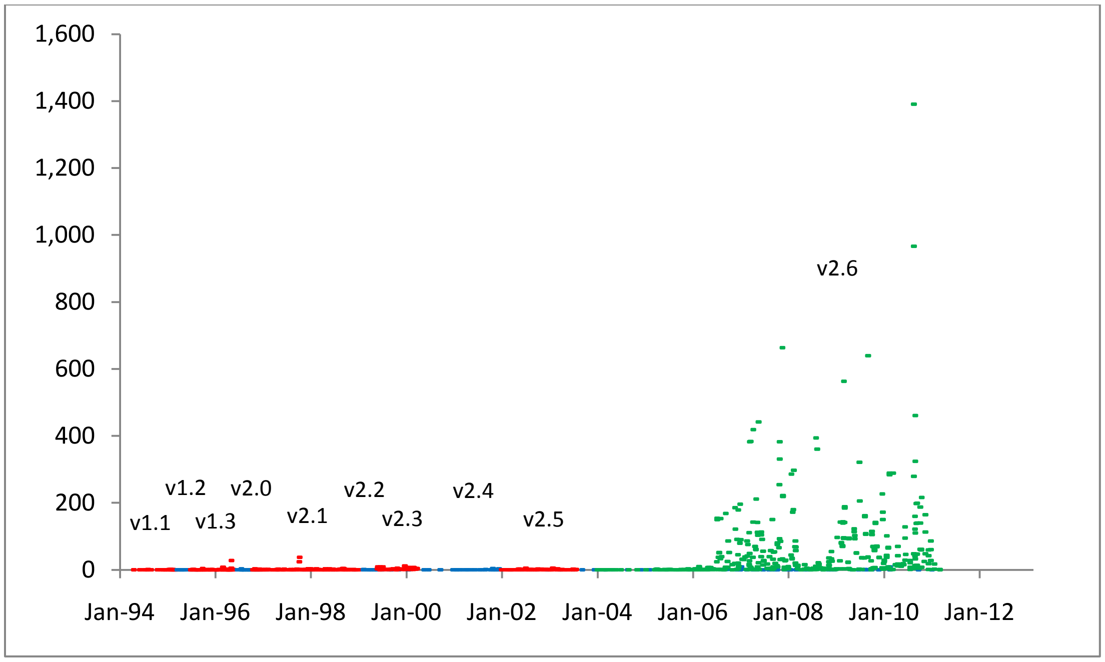


Figure 4.8. The average slice growth per day measured in KLOC.

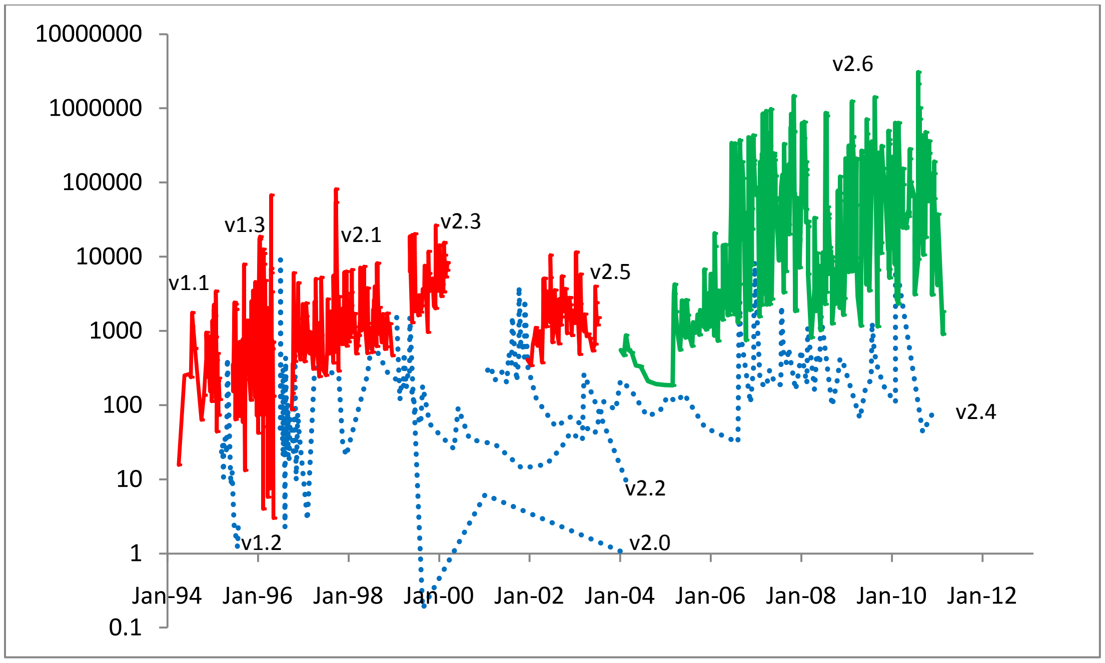


Figure 4.9. Average LOC growth per day measured in KLOC (logarithmic scale).

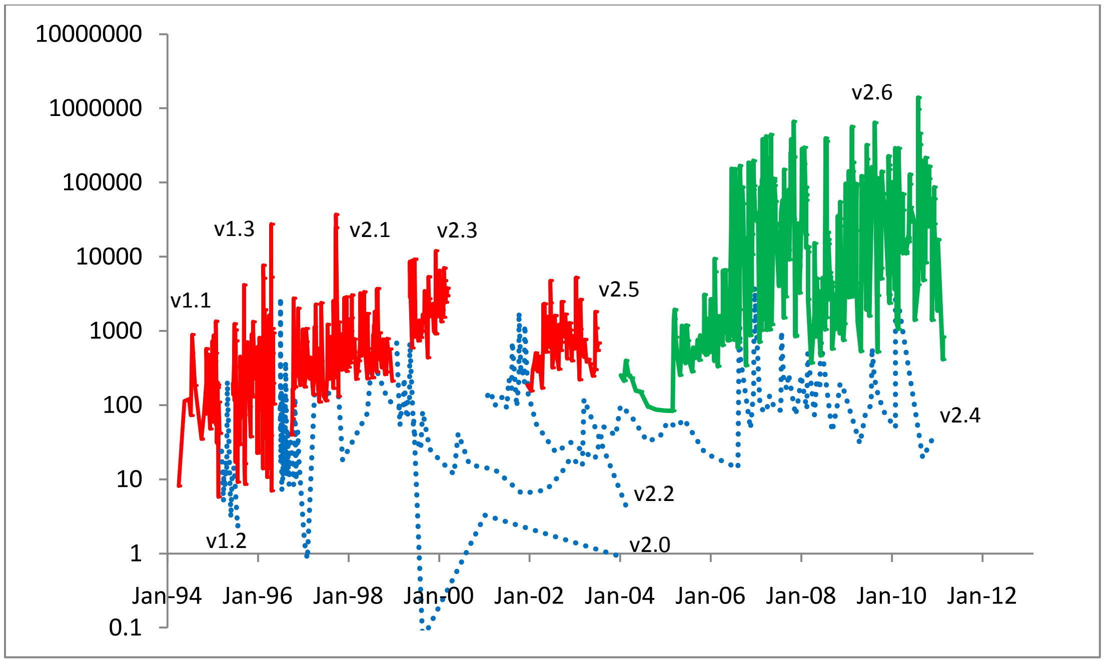


Figure 4.10. Average slice growth per day measured in KLOC (logarithmic scale).

Now, we looked at the slice size relative to LOC (Figure 4.11), which is interesting since this average percent of the slice size represent the "*slice coverage*" proposed originally by Weiser [Weiser 1981] then formulated by Ott et al [Ott, Thuss 1993] over modules as a comparison of the length of the module's slice to the length of the module. Here we extend this definition over the entire version (the version slice size divided by the system size both measured in LOC). The results as shown in Figure 4.11 indicate a small decrease over time (17 years) in the development versions than stable versions, especially in the early development versions (1.1, 1.3, 2.1, and 2.3). This gives the impression that with time the average complexity of the source code is decreasing, and thus maybe the quality of the Linux kernel is improving.

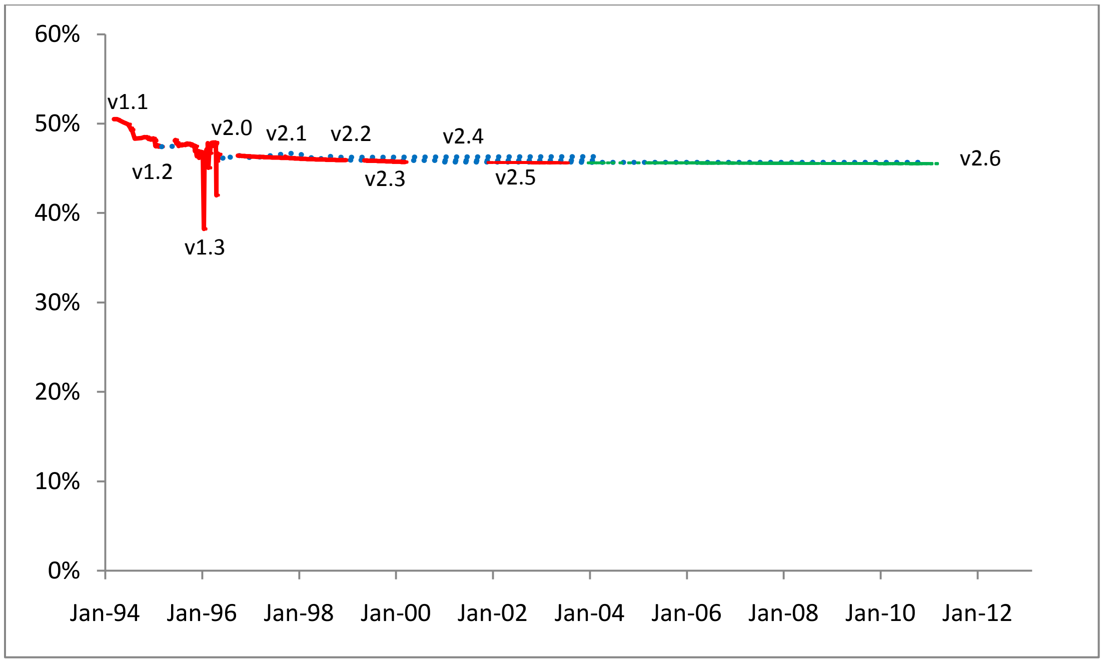


Figure 4.11. The percentages slice size relative to LOC over 10 major versions of Linux kernel.

4.3 Analysis of Maintenance Activities

In this section, we use the above results to reflect and characterize the four types of maintenance activities; which are corrective, perfective, adaptive, and preventative.

4.3.1 Stable and Development Versions

As we mentioned above, we expect the development versions of the Linux to reflect perfective type of maintenance activity, whereas stable versions reflect corrective type of maintenance. This is based on the structure of these two types of branches of the code, which are maintained in parallel to each other. However, it seems that in practice this division was not always followed.

An example of extreme mixing of the roles of the versions arises at the beginning of version 2.4. A closer investigation shows that the last release of version 2.3 was released on March 11th, 2000. The first release of version 2.4 was released on January 4th, 2001. And the first release of version 2.5 only on November 23rd, 2001. Therefore, there is a gap of some 20 months between consecutive development versions (i.e., 2.3 and 2.5). However, it seems that the early fraction of version 2.4 reflected development activity that was being made without officially released in a new development version.

The other gap of ~10 months is between version 2.3 and version 2.4 seems to have been filled (at least partially) by version 2.2, since this version exhibits a strong growth in this time period.

4.3.2 Time-Intervals between Versions

As we have seen from all the graphs above and based on the structure of the Linux kernel, it seems that the growth trend follows a consistent pattern: a new development version is released after a number of releases of stable version, and after there are no more releases in that development version, a new stable version is released. However, during the time interval and releases of the stable version there are still releases of the previous stable version. For example, version 2.0 had continues releases until the end of version 2.2.

In this context, we examined the intervals of time between consecutive releases (measured in days) inside the same major version. Figure 4.12 displays the raw data of the intervals for each version, and Figure 4.13 shows the statistics (median and 25th, 75th, and 95th percentiles) of the intervals for each version. Notice that in Figure 4.13 versions 2.0, 2.2, and 2.4, the 95th percentile values exceed the top of the graph and their true values are appear on the labels above their up-bar boxes.

Looking at Figure 4.12 we can see an interesting pattern such as, generally the development versions have very low values, and so do version 2.6, while the stable versions have much higher values. We also notice that at the end of any development version (red line) there is almost a simultaneous beginning of a new stable version (blue line), the exception here we do see a significant gap between versions 2.3 and 2.4, and between versions 2.5 and 2.6. Those two gaps are results of the structural changes in the Linux kernel numbering scheme.

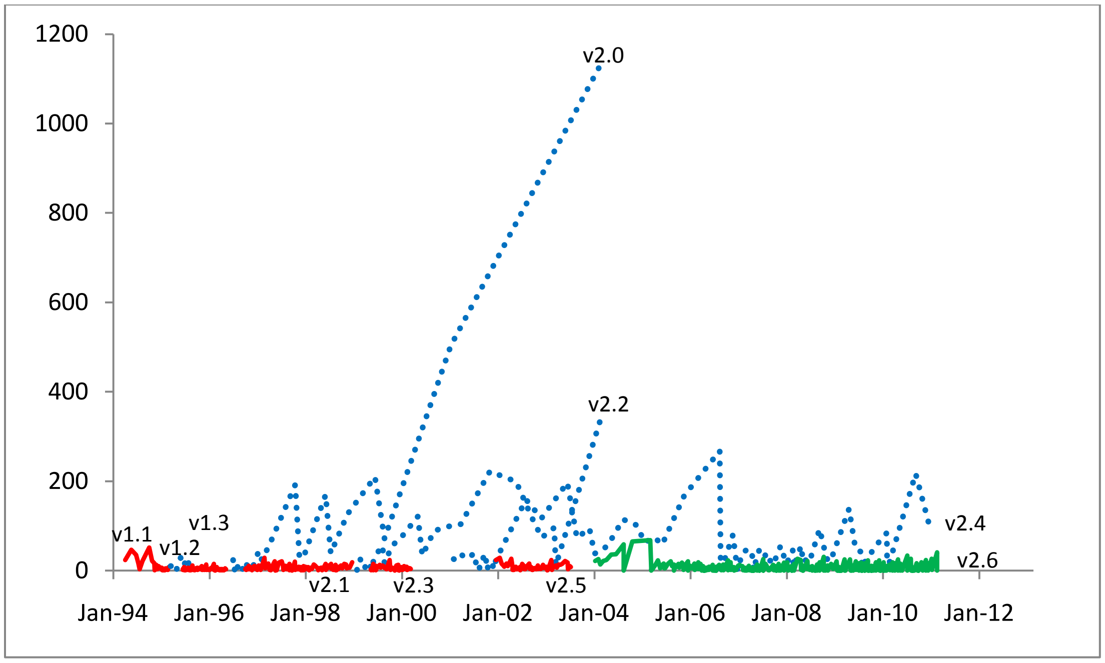


Figure 4.12. Intervals between version's releases measured in days.

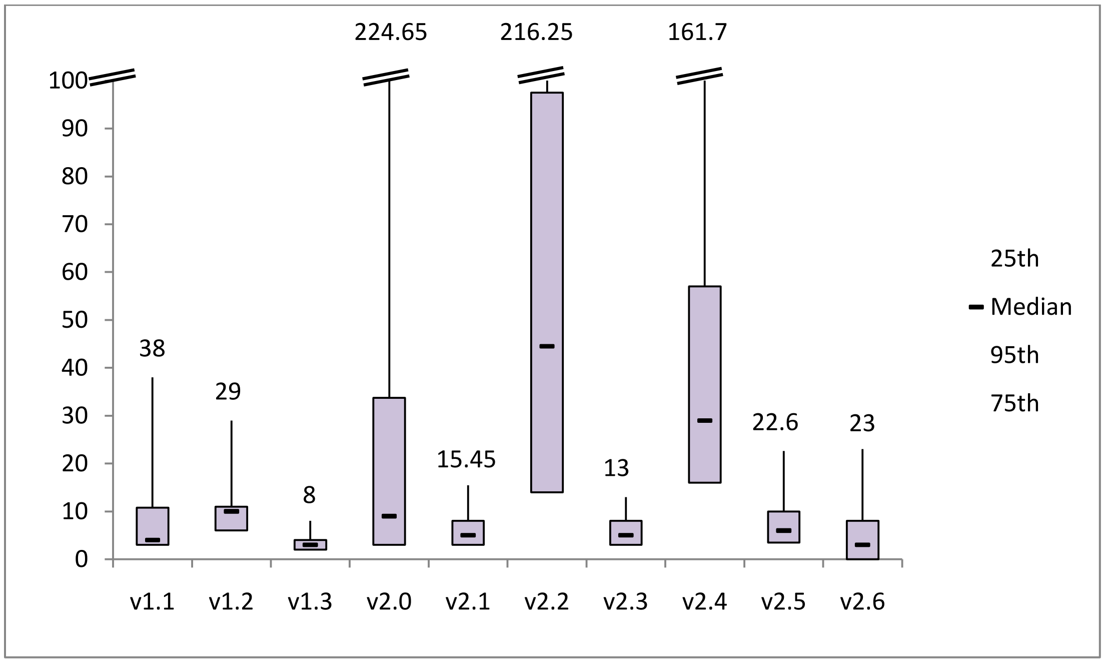


Figure 4.13. Statistics of intervals between releases, within major version measured in days (ordered as: 25th, median, 75th, and 95th percentile).

Looking at Figure 4.13 we can notice that the bodies of the up-bars of the stable versions are usually higher than those of the development versions. In addition, the high values and the variance are usually higher in stable versions. In other words, we can see that the stable versions are released less frequently, while development versions are released quite often. For example, all medians, 25th, and 75th percentiles in development versions are lower than 10 days.

In conclusion, releases of stable versions are less frequent and usually on a weekly to monthly basis, however releases of development versions are very frequent and are on a daily to weekly basis. The interval wait between stable versions was a year and more, and with development versions we find a much faster and more stable release rate.

4.3.3 Corrective Maintenance

We analyze the corrective maintenance as reflected in consecutive versions of stable kernels. The assumption here that changes in successive versions of stable kernels are usually corrective, due to the structure of the releases in Linux kernel as indicated previously in Section 4.1.1.

As we have seen in the results above (Section 4.2), for each of the different metrics we calculated (e.g., LOC, number of files, slice size, etc.) the values of these metrics are fairly often constant for the stable versions. This is seen in versions 1.2, 2.0, 2.2, and 2.4, and also in each of the subversions of version 2.6. However, the metrics values in stable versions do change as follows: the first is the large jumps seen in version 1.2. This is explained by changes in functionality, where new features were added into stable version 1.2, but without a production of a new major version. By excluding this jump in version 1.2 (since the jumps do not reflect corrective maintenance) we can accept the hypothesis that the stable versions represent corrective maintenance fairly often.

The other change noticeable in stable versions is that both the average LOC and slice size per file, and LOC and slice size per day metrics tend to grow initially (increase or decrease) and only then become constant. For example, for the average LOC and slice size per file there is a little growth in version 1.2, more in version 2.0. For versions 2.2 and 2.4, the picture is reversed: there is a little decrease initially then become more constant especially more using slices size per files than LOC per files.

This initial growth may indicate that corrective maintenance tends to add code and complexity to the existing structure, without an appropriate asset in reorganizing and refactoring. However, the slice coverage metric generally indicate a decrease for the stable versions over time however still have a higher values than for development versions. The meaning of this is that the growth of the slice size is faster than the LOC in stable versions.

4.3.4 Perfective Maintenance

We analyze perfective maintenance as reflected in consecutive versions of development kernels. Based on the structure of the Linux kernel the main motivation for new development version is to develop and add new features.

The results above show that the different metrics for consecutive versions in development kernels usually change more than stable versions. For example, using the average LOC and slice size per day metrics, the values for development versions are higher and exhibit an increasing trend. As a consequence, the slice coverage is lower for the development kernels, indicating that development versions are less complex and more maintainable than stable versions. Of course, if we compare at the system size (LOC and number of files), and slice size, the picture will seem the reversed, since there are more LOC and more files in the development versions.

Concerning the improved trend in these metrics over time in development versions, two explanations are possible: first, there is a code improvement in the development versions. Second, there is many small files are being added at a higher rate. That's why on average the growth in the desirable direction. While more detailed research is needed to quantify those two options, we have seen specific examples (e.g., LOC and slice size per day) of improvements to code complexity and structure especially in development versions. Thus as a result we indeed have preliminary evidence for code improvement in development versions.

4.3.5 Adaptive Maintenance

We analyze adaptive maintenance as reflected in specific directories in the development kernels, since they best reflect the adaptation to changes in the environment, e.g., addition of new devices that need to be supported.

For example, Israeli et al [Israeli, Feitelson 2010] analyze the adaptive maintenance activities in the Linux kernel using the traditional and well known metrics such as lines of code (LOC), McCabe's cyclomatic complexity (MCC), Halstead's volume, difficulty, and effort metrics (HV, HD, and HE), and Oman's maintainability index (MI) as reflected in the *arch* and *drivers* directories. An interesting note is that both directories (*arch* and *drivers*) are external to the core of the kernel, and these directories grow not only due to improvements, but also with the need to support additional processers and devices [Israeli, Feitelson 2010]. In addition, once a major change or update is takes place in the *arch* and *drivers* directories, the *include* directory is usually updated with new header files. Therefore we will concentrate study those three directories next.

We believe by analyze the large discrete jumps occurring in various metric values we can specify the adaptive maintenance activity, especially if the large jumps reflected in the *arch*, *drivers*, and *include* directories. To do this, we should measure all the changes made that reflect the adaptive changes. However, a semantic change in one function might create a ripple effect among other functions, and consequently affects other directories. For example, an adaptive change in one directory might affect the directories in interest (*arch*, *drivers*, and *include*).

To this end we introduce a source code change metrics based directly only on slices of source code. It entails computing the slice for all the variables in a system and modeling how slice changes over time. Our assumption here, that slicing can be used focusing on selected aspects of semantics, and provide a detailed analysis of impact of the change code.

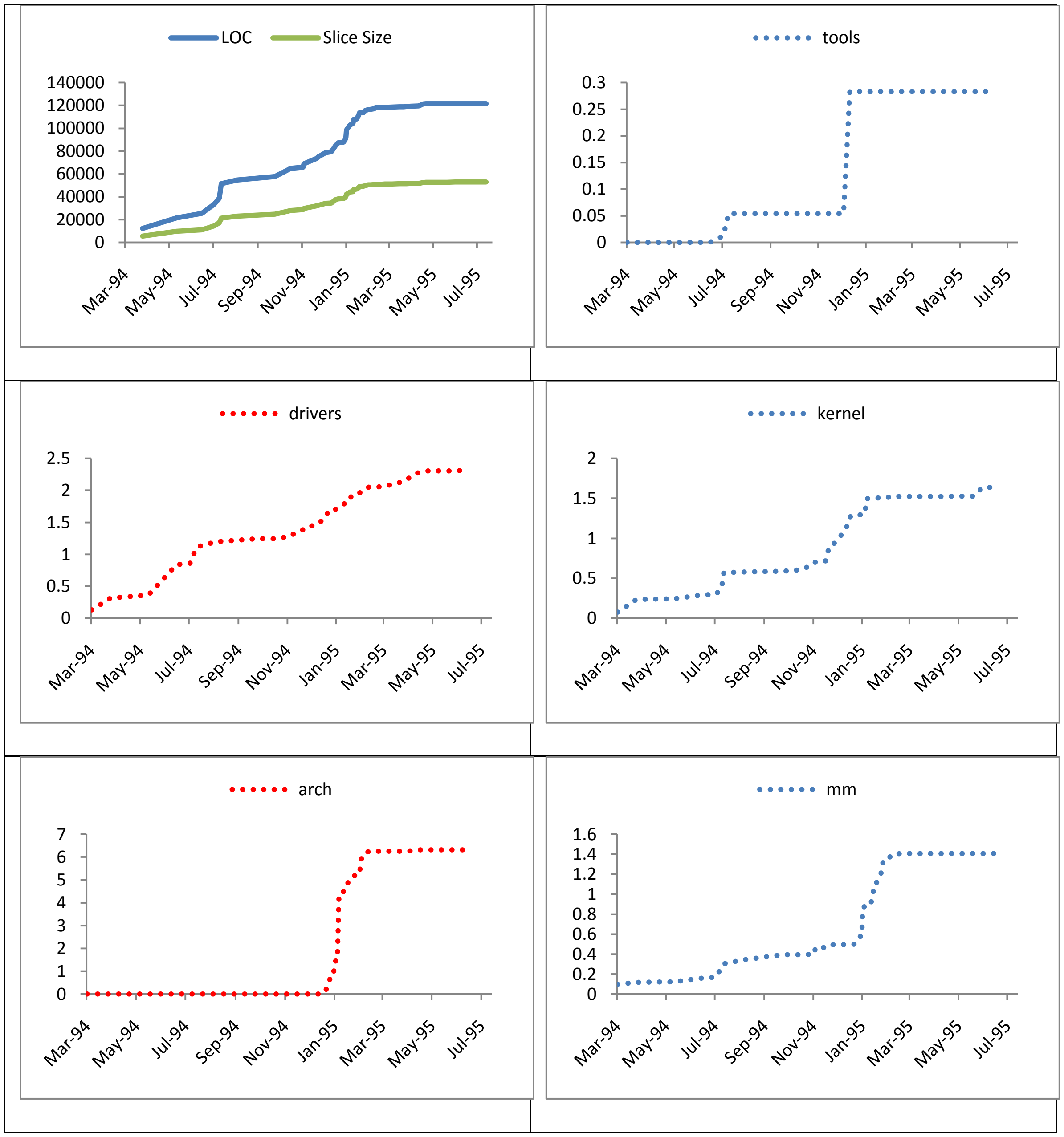
In the next chapter (CHAPTER 5) we will show how to measure forward slicing, and then show how the slice-based metrics are computed using this approach. In short we start by measuring the number of lines that changed at variable level, and accumulating these values over the: function level, file level, module level, and the entire version.

By using the information stored in each variable's slice, we retrieve the size of the slice for each variable measured in number of lines of code. The total slice size for each function is the sum of individual slice sizes for each variable in the function. The total slice size for a given file is the sum of slice sizes for each function in the file. Finally, the slice size for a given module (directory) is the sum of slice sizes for each file in the module.

For example, if we want to measure the average slice-based changes in the *drivers* directory between releases 1.1.0 and 1.1.13, we start by specifying the files (inside the *drivers*) that changed between the two releases, then calculate the relative change (slice size divided by file size) for each file, and finally we take the average over these changes.

As indicated previously, adaptive maintenance may be inferred from the large changes to *arch*, *drivers*, and *include* directories. That is, these directories should be larger in LOC. Such consideration indicates that version 1.1 is special, as seems there is a significant improvement in the code in that version: the slice coverage as a complexity metric decreased considerably, and the average LOC and slice size per file grew. Therefore, we choose to analyze the adaptive changes as reflected in the *arch*, *drivers*, and *include* directories, especially in version 1.1 and we also consider the stable version 1.2 since this version exhibits a large jump in several metric values, and based on the Linux structure it should not reflects an adaptive changes.

To accomplish this, we compare the accumulative growth-trend of the slice-based changes of the 11 major directories in the Linux kernel with the growth trend of LOC and slice size as shown in Figure 4.14. As we can see, all the directories have the same trends as the LOC and slice size, especially *kernel* and *drivers* directories. However, an interesting observation is that the pattern of growth in the *arch*, *drivers*, and *include* directories can be different from the rest of the kernel, especially is selected points.



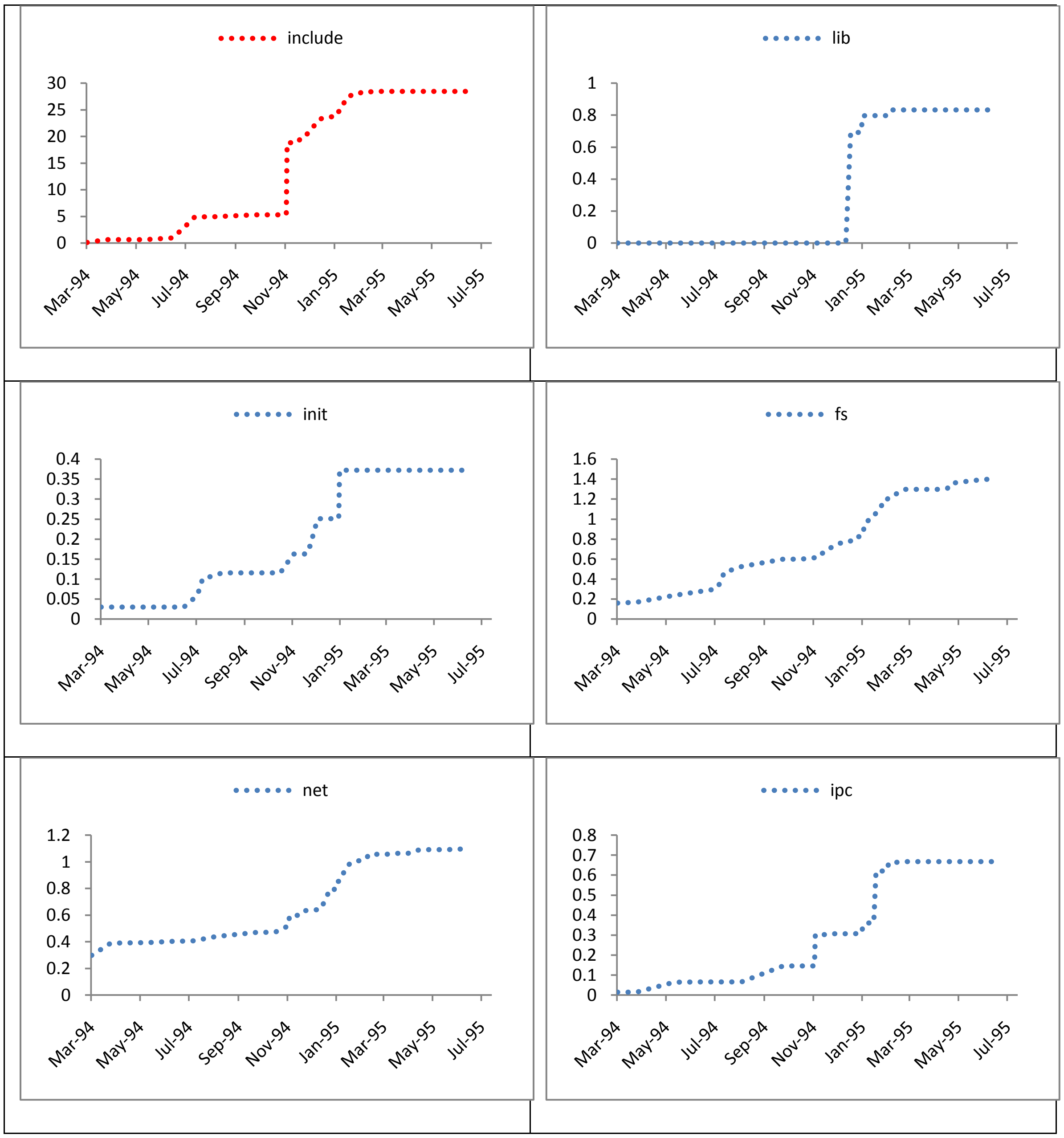


Figure 4.14. The Accumulative growth of LOC and Slice Size compared to the slice-based changes in 11 main directories in Linux kernel, between two versions 1.1 and 1.2.

For example, Figure 4.15 shows that the relative changes in *arch*, *drivers*, and *include* directories have a higher magnitude than other directories in both versions 1.1 and 1.2. However, the changes in the development version are stinking, and somehow constant in the stable version.

So we can conclude that the core kernel and the *arch*, *drivers*, and *include* directories do not have the same amount of changes (in both, the same version and in different versions). For example, the average changes in version 1.1 are more noticeable in these directories, whereas in the stable version 1.2 is almost constant. This seems to indicate that adaptive maintenance, as reflected in these directories, leads to high values of changes.

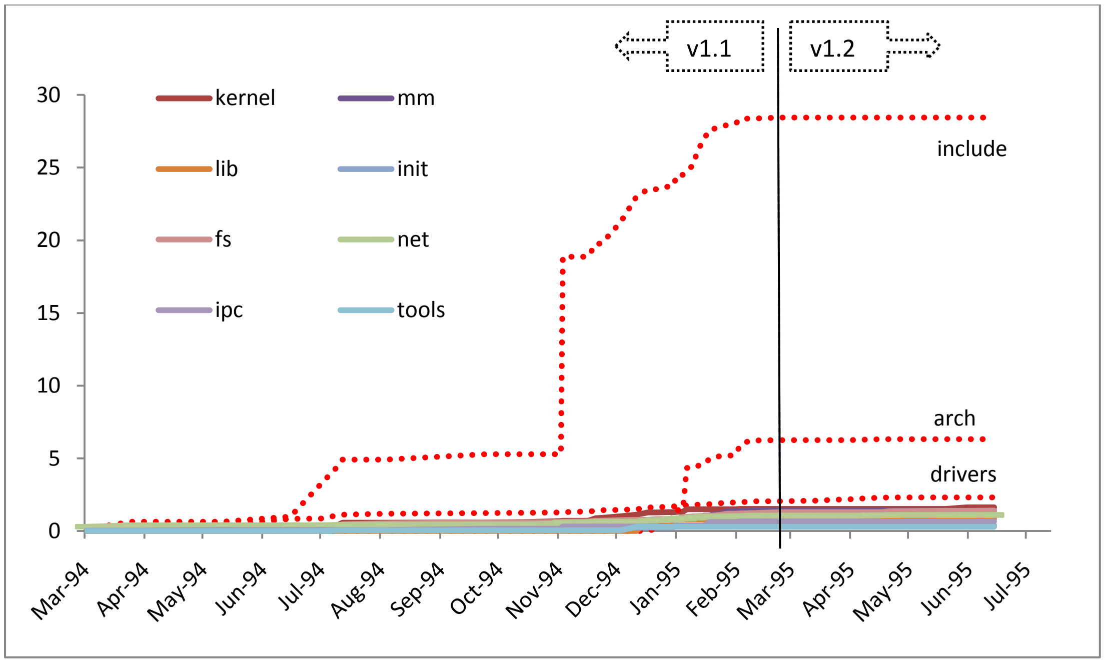


Figure 4.15. Accumulative slice-based changes in the main 11 Linux kernel directories, between version 1.1 and version 1.2.

4.4 Analysis of Lehman's Laws of Software Evolution

In this section we analyze Lehman's laws of software evolution (using the Linux kernel as a case study) based on the calculated slice-based metrics. The objective is to examine whether these laws are supported with our slice-based metrics. The laws are mentioned next based at the date of their introduction. This is similar with Lehman's empirical work to support these laws [Lehman 1996; Lehman, Ramil, Wernick, Perry, Turski 1997].

4.4.1 Continuing Change (law 1)

According to this law, the system that is used must be continually adapted else it becomes less useful. Typically, it is hard to distinguish between the general growth (sixth law) and adaptation to environment, and between adaptation changes and general changes. For example, when support for USB is added, this could be considered as a new feature or an adaptation to a changing environment, or probably it is a feature that was added in response to a change in the environment.

As we explained before, the adaptive changes in the Linux kernel are easily identifiable by examining the *arch* and *drivers* directories (which contain all he code that pertains to processor architectures and peripherals, respectively), and consequently the *include* directory which is usually updated with new header files once an update or a major change in *arch* and *drivers* directories is made. Thus, the code that is added to these directories should reflect adaptation to the changing environment.

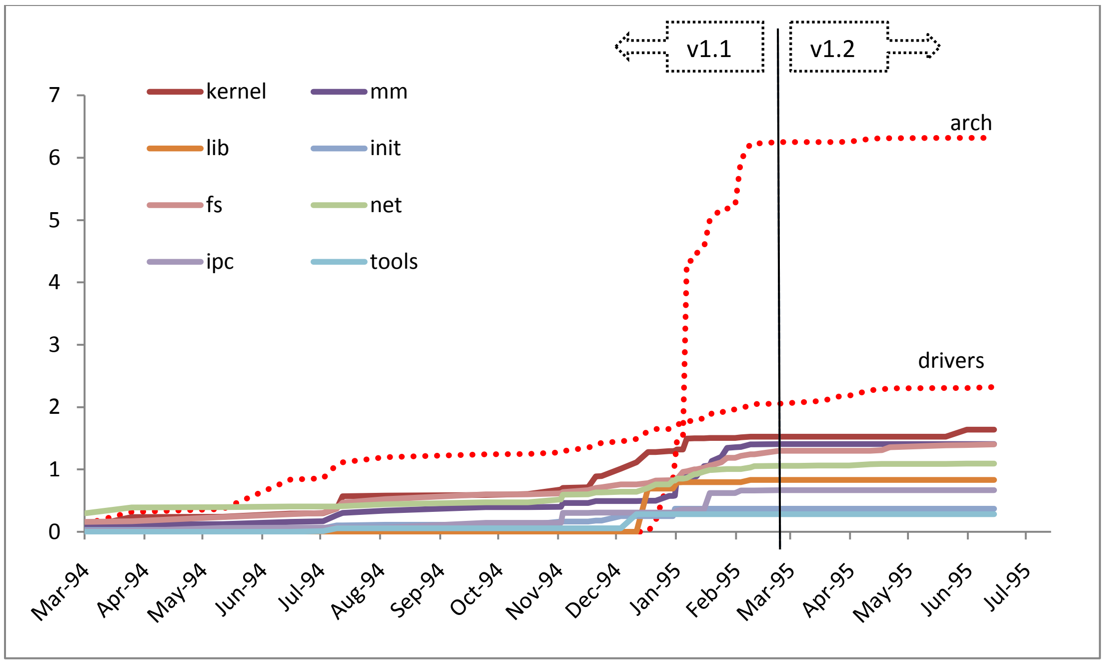


Figure 4.16. Accumulative slice-based changes in the Linux kernel directories, version 1.1 and version 1.2, without *include* directory.

As shown in Figure 4.14, we can see that the accumulative slice-based changes in those directories mirror the growth of the Linux kernel as a number of files and LOC over the first two versions (v1.1, and v1.2), specifically the *drivers* directory. In addition, it is easily seen in Figure 4.15 (repeated in Figure 4.16 without the *include* directory) that the *arch* and *drivers* directories accommodate the high value of changes over other kernel directories.

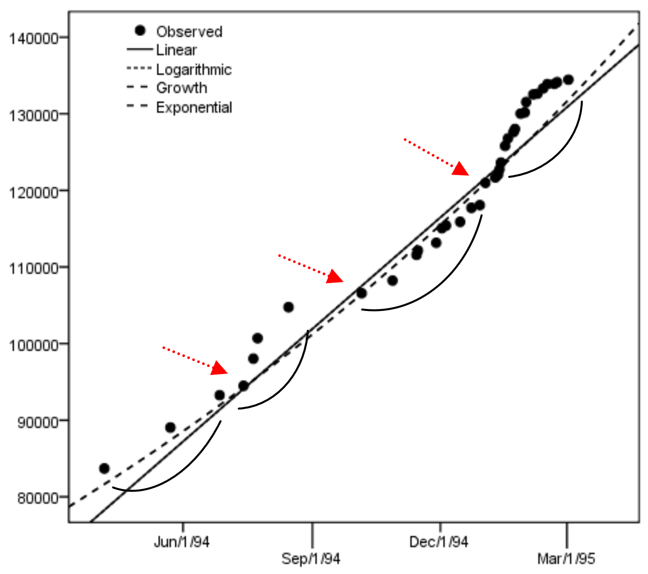


Figure 4.17. Slice size growth at the development version 1.1 with 36 releases.

Based in above results, we can assert that Linux (at least in the first two versions) exhibits an adaptation to its environment. However, the original law is probably of wider scope than this particular example, i.e., continuing changes in general. Therefore, the need is to consider the general changes not only the adaptation ones.

A closer investigation of slice growth in version 1.1 as shown in Figure 4.17, we can distinguish around the growth trend of the slice size a ripple shown by the arcs and the red arrows. The higher increase with the slice size is followed by a lowers than average.

Lehman [Lehman, Ramil, Wernick, Perry, Turski 1997], discussed briefly the possible causes of such cycles as users requests for enhancements which leads to exceeds in the growth in the next version, however this incremental growth may increase the possible number of defects to be fixed, so as a result its brings the growth back down at the followed versions. The existence of ripple cycles provides evidence that the growth rate is sometimes decline with time. The explanation of these cycles by Lehman was based on terms of feedback loops (the eighth law).

This phenomenon can be proved if we consider a stable version, such as version 1.2 with 14 releases. Based at the Linux structure, the stable versions are more likely to include the corrective maintenance activity. As shown in Figure 4.18, release number (8) suffers a performance problem, since the amount of change (slice-based) at release number (9) (red arrow) is extremely increased, which indicate that release number (8) is unplanned release. Therefore, the pre planning is very important to conserve the consistency with the production of versions.

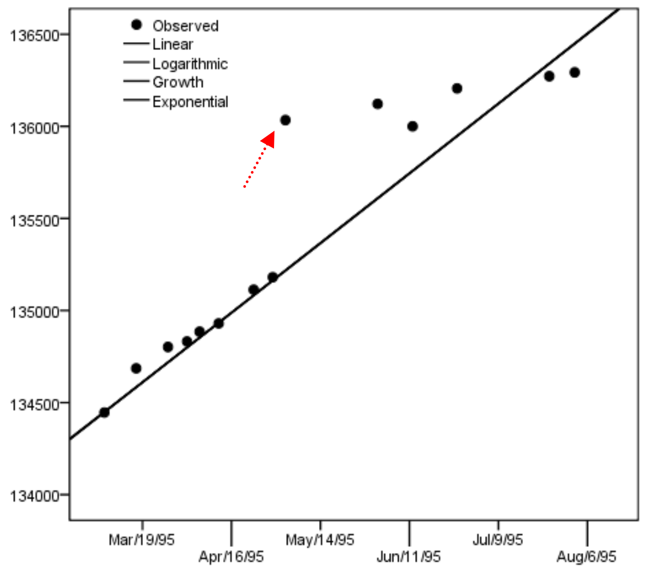


Figure 4.18. Slice size growth at the stable version 1.2 with 14 releases.

4.4.2 Increasing Complexity (law 2)

This law states that as a program evolves its complexity increases unless there is a work done to reduce the complexity. This law is very hard to prove, since both trends are possible; either the complexity is increases, or decreases by a work done to reduce it.

A number of researchers [Lehman 1980; Lehman, Ramil, Perry 1998; Israeli, Feitelson 2010] support this law by studying data that growth rates decline over time. However, an alternative approach is to measure directly the source code complexity given that the availability of the full source code for each version. In particular, we measured the system slices, which is equivalent to the number of paths in the system's representation model (e.g., program dependence graph) and provide a detailed analysis of impact of the changed code.

The results of slicing the full source code of all Linux versions are shown in Figure 4.5. As we discussed before, when the size of the code grows, so does the slice size. Therefore, it's more interesting to look at normalized values, such as the slice size relative to LOC. The results in this case indicate a declining trend. Thus the total slice size is in general growing more slowly than LOC, and thus the average complexity is decreasing.

Focusing on the overall basis, the complexity of the system is increasing (as system size, number of files, and slice size are increasing) and decreasing in slice coverage metric. For example, when we look at the slice size values relative to LOC in version 1.1 within different releases, as shown in Figure 4.19, we can see that the distribution is improving in the development version over the time. So maybe we can say that there was work done to reduce the complexity of the kernel in selected points. However, it is possible that a large part of this apparent improvement is due to increasing in the number of source code (LOC). Indeed, tabulating the slice coverage in all the versions leads to values that are typically lower for the stable versions than the development versions. The accumulative slice coverage of all versions (stable and development) is shown in Figure 4.20.

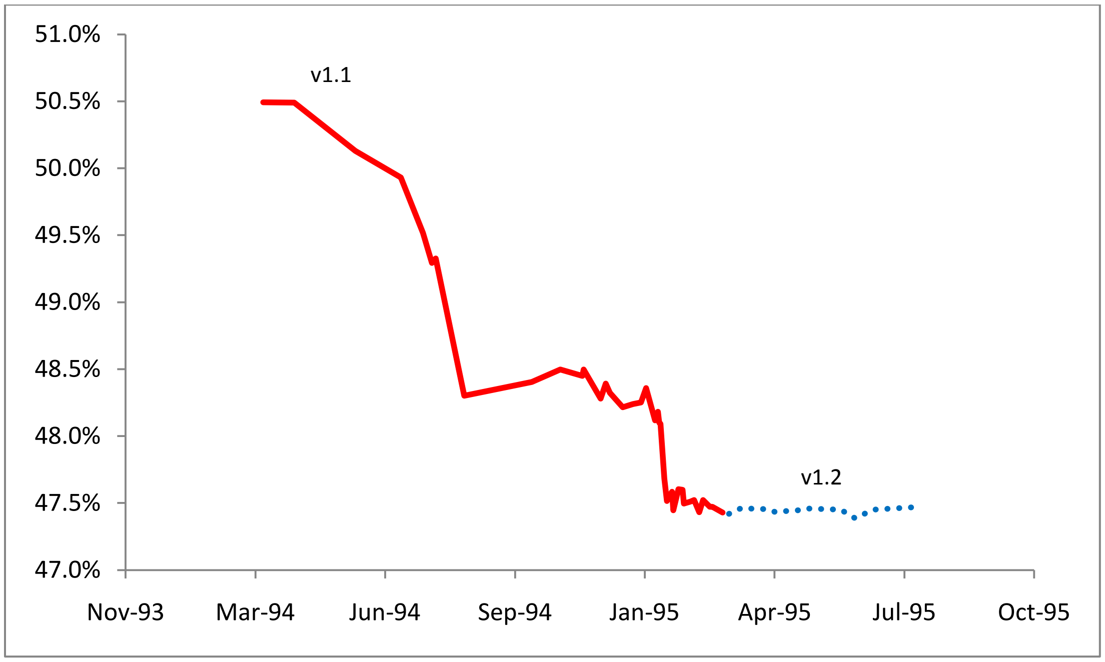


Figure 4.19. Slice size relative to LOC (Slice Coverage) for versions 1.1 and 1.2 only.

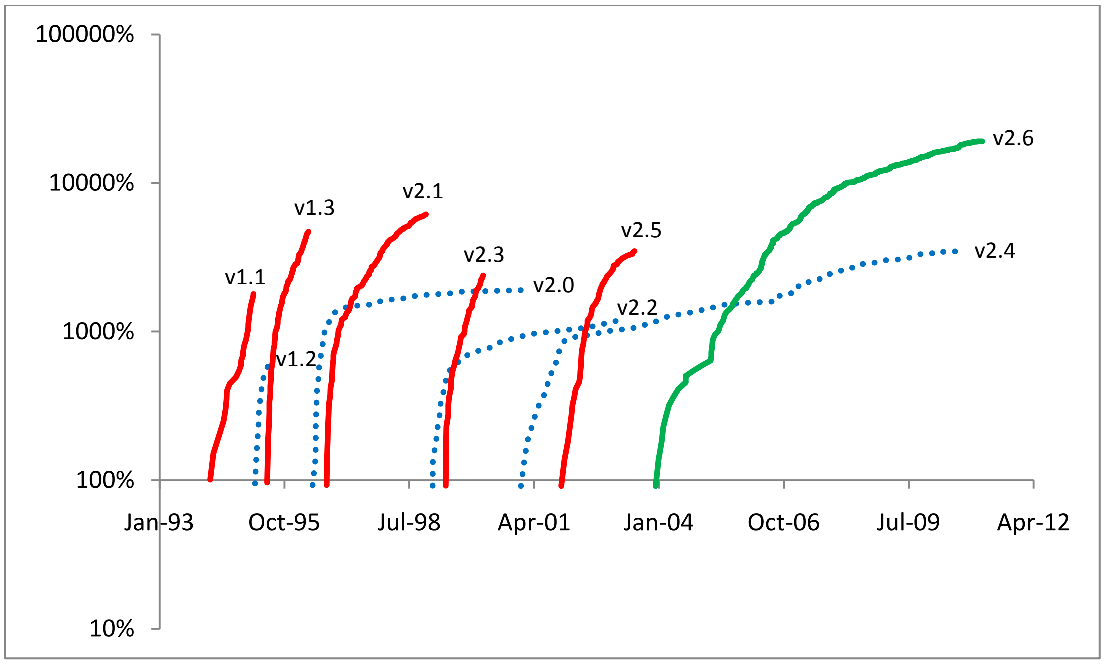


Figure 4.20. Cumulative slice size relative to LOC, (logarithmical scale).

Disregard day's interval between releases, we can observe that the pattern for the stable versions represented by the blue doted lines is increased initially indicating that the fraction of LOC with a high slice size is increasing. This can mean that more high complexity LOC is added. Then the pattern exhibits a dramatic decrease in the top releases, indicating that the fraction of LOC with a high slice size is decreasing, which means that the percentage of increase in the slice size and LOC tends to be equal.

For the development versions (red doted lines and version 2.6), the results is more concave over time then the stable versions, indicating that the fraction of LOC with a high slice size is increasing. This means that over time we indeed have a higher complexity.

In conclusion, we see evidence for an investment of work over time to reduce code complexity, both in stable versions and development versions and in specific the stable versions in later releases. This result is expected since in the development versions the developers tends to add more LOC without paying attention to the code complexity, later on in the subsequent stable version there is a significant reduction in code complexity. Therefore we can state that the number of impacted statement lines of code is stable which means the cohesion between the system parts is improved and thus that the quality of system is increased.

4.4.3 Self Regulation (law 3)

In accordance to this law, the system evolution process is self regulating, leading to a steady trend. Lehman in [Lehman, Ramil, Perry 1998] indicated ripples in the graph of the system size measured in number of modules as a function of release sequence number in order to explain this phenomenon. He claimed that this ripple indicates the existence of feedback system which balances the system so as drive it to its goal.

We used the slice size here to study this phenomenon. Again as shown in Figure 4.17 we can see that the ripples is repeated pattern of deviations from the average growth rate, such as alternating between periods of faster growth and fixing of the trend by periods of slower growth. These ripples in the growth of slice size as well as the number of files (Figure 6.1) might suggest self regulation.

4.4.4 Conservation of Organizational Stability (law 4)

According to this law, the average work rate on an evolving system is statistically invariant. In order to examine this law we should study the maintenance effort spent on the system. However, as we will explain in the next chapter the data about person-hours, and number of maintenance tasks is hard to be getting in open-source systems [Yu, Schach, Chen 2005]. Therefore, we will try to study this law looking at slice-based source code changes in the files level and also looking at the amount of versions released per time.

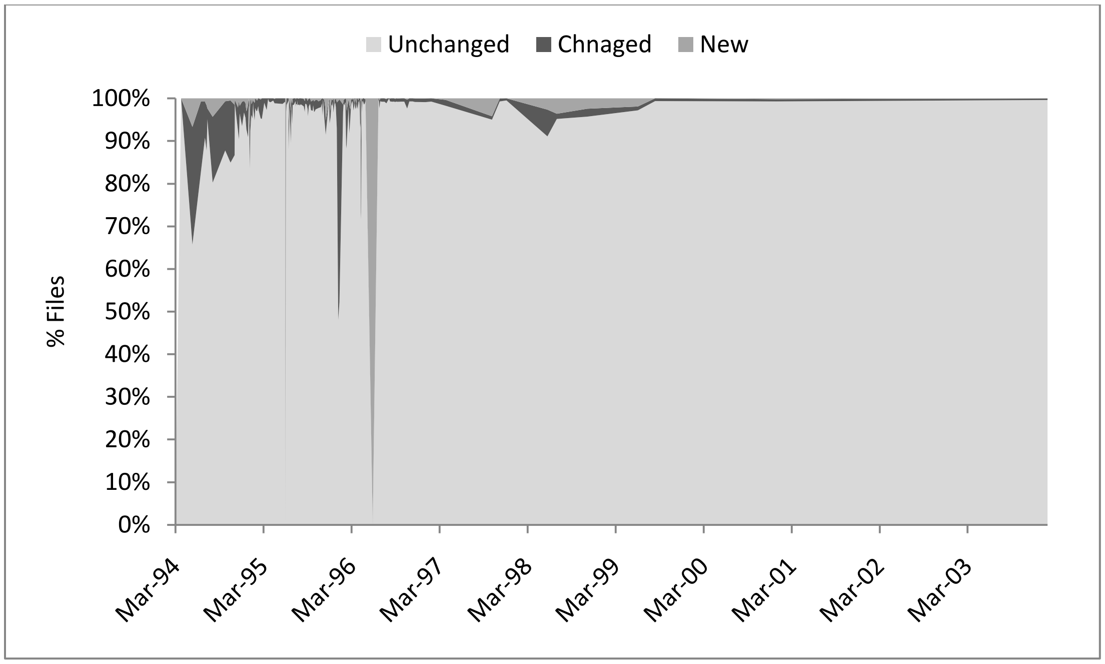


Figure 4.21. Files change evolution in the first four versions (v1.1, v1.2, v1.3, and v2.0) of Linux kernel. This graph illustrates change property captured by slicing.

We start by consider the number of elements handled, as suggested by Lehman. We will focus on the average rate of change code for files. That is, the likelihood that a file will change from one release to the next. To assess the likelihood of a file changing, we gather the *ratio of files that are unchanged*, *ratio of files that are changed*, and *ratio of files that are added or removed*, by comparing successive releases in a given version. Figure 4.21 shows the number of files that were added, deleted, and modified (divided into those that grew) between consecutive releases.

As may be expected, the fraction of files that are handled seems to be relatively stable, except perhaps for some decline in the first years. On average across all versions we observed that 96% of the files are unchanged, 3% are modified, and 1% is added/removed files. Thus if we interpret rate to mean the fraction of source code that is modified in each release then the data support the claim that the work rate is almost constant.

The invariant work rate can also be observed with regard to the release rate itself, such how often releases happen. We start analyzing the number of releases per month for the development versions as shown in Figure 4.22. In the *x-axis* each stub represents a year, and each bar represents a month. The vertical lines with the version label (i.e., 2, 4, 6, 8, and 10) represent the start of that new major version. It is obvious that since the mid of 1997 the rates seem stable (around 3 – 6 release per month) and with a minimum is equal to 1 and the maximum is 8.

From Figure 4.23, we can see that the stable versions are released less frequently than the development versions (compared with Figure 4.22). That is, usually there was one release per month, and the maximum is 10. Starting with version 2.6 as shown in Figure 4.24, the versions are timed to be released once every ~ 3 months. It is important to remember that the Linux release are organized into major (e.g., 1.1, 1.2, etc.) and minor (e.g., 1.1.13, 2.2.3, etc.) version. Therefore, one should consider the intervals between major releases independently from those leading to minor releases.

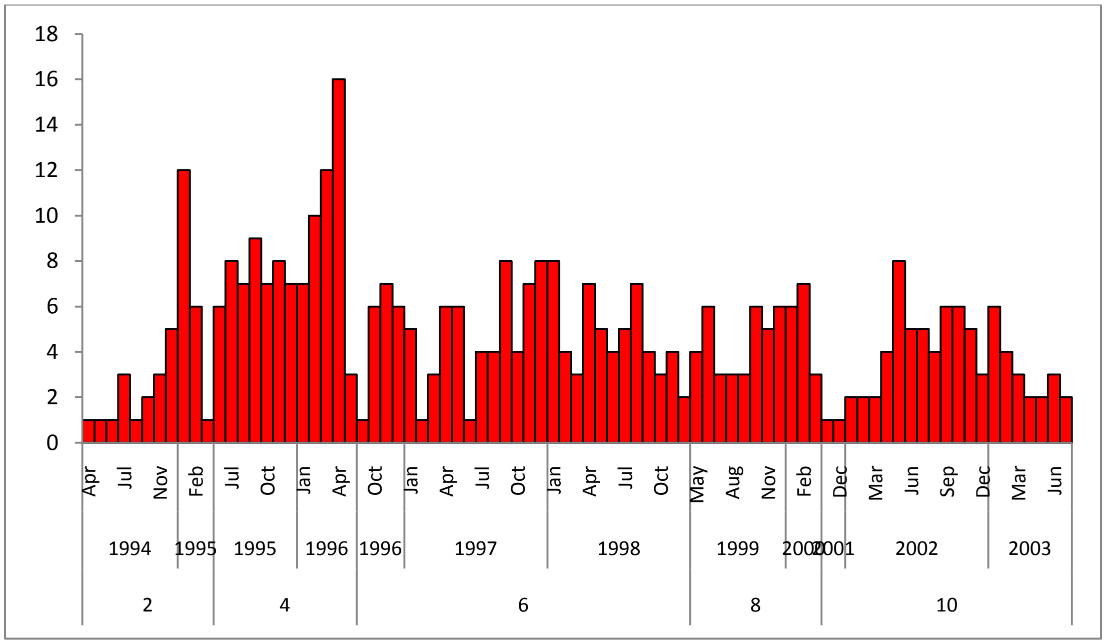


Figure 4.22. Number of releases per month for development versions, in *x-axis* (2) = v1.1, (4) = v1.3, (6) = v2.1, (8) = v2.3, and (10) = v2.5.

Again we see that development versions are consistently related at a high rate, stable versions are related much less frequently, and with version 2.6 all this changed such as in the first 11 releases the distribution was similar to that of previous stable versions.

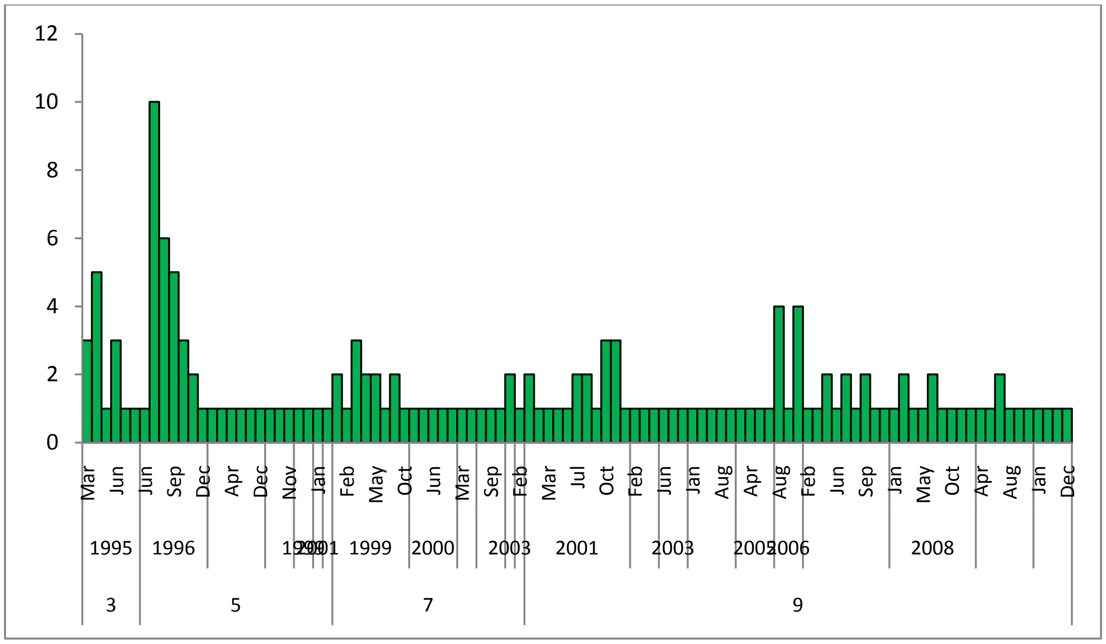


Figure 4.23. Number of releases per month for stable versions, in the *x-axis* (3) = v1.2, (5) = v2.0, (7) = v2.2, and (9) = v2.4.

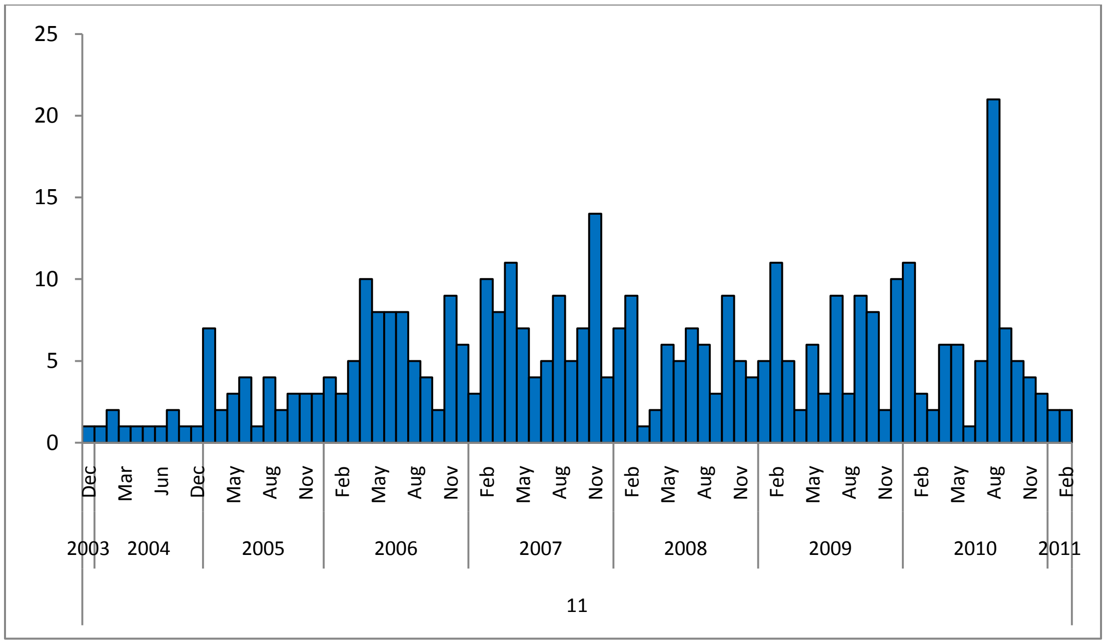


Figure 4.24. Number of releases per month for version 2.6.

4.4.5 Conservation of Familiarity (law 5)

The *Conservation of Familiarity* law is clearly supported through the successive versions at the development versions of the system. According to this law, the content of consecutive releases is statistically invariant. That is, the change between consecutive releases is limited so as the developers could maintain their familiarity with the code from one side and using the system from the other side [Israeli, Feitelson 2010]. In order to study the conservation of familiarity with the code, we should consider the pattern of releasing development versions. As shown in Figure 4.13 we can see that the development releases come in rapid succession, typically only days apart. Moreover, Figure 4.22 provides clear evidence as the rate of these releases per month is high. As shown in Figure 4.1, Figure 4.2, and Figure 4.4, the development versions form a nonstop line plotting, and the stable versions are stem out directly from the last available development version.

All of these finding indicate that when the developers are familiar with one version, then may expect a little change in the subsequent ones. Our results above indicate that in consecutive releases of the same major stable version, the changes are very small, thus we can say that within stable versions Linux indeed conserves user familiarity.

4.4.6 Continuing Growth (law 6)

According to this law, the functional capability of the system must be continually improved in order to maintain the user needs over its lifetime. Clearly, there are significant functionality additions in differences between consecutive releases of development versions to the Linux kernel all the time.

Additionally, the super-linear rate of growth of system size and slice size metrics in the development versions, and the little increase in the beginning of the stable versions can support this law directly.

4.4.7 Declining Quality (law 7)

According to this law, the quality of system is declining unless continually maintained and adapted to a changing environment.

This law is somehow related to the *Increasing Complexity* (second law) and could be supported using the slice size relative to LOC as shown in Figure 4.11. This is a comparison of the length of the slice to the system size. The results indicate a declining trend, thus the slice size is in general growing more slowly than the system size (LOC).

While it is hard to obtain a precise quantified metrics to measure the quality, we believe as others, that the system slice could be used safely to indicate the quality of the system.

4.4.8 Feedback System (law 8)

We already discussed the feedback issue as an element of the self regulation law above. Lehman discussed briefly possible causes as user requests for enhancements leads to incremental growth in the next release, as a result increase number of possible defects to fix which brings growth down back.

Lehman in [Lehman, Ramil, Perry 1998] try to support this law by studying the stability of growth models in the FEAST project, specifically he consider the assumption that initial releases are enough to extract growth model to predict future sizes accurately. However, in our study of the Linux kernel, we have the evidence of this law since the continued development of the Linux as an open-source system is guided by feedback from the user community. For example, bug reports and fixes, contribution of developers.

4.5 Chapter Summary

In this chapter, we calculated metrics for the different versions of Linux by slicing all versions (11 major versions with a total of 974 releases). Then we analyzed the calculated metrics in order to provide a model of the maintenance process in the Linux using the new slice-based metrics.

We compared the potential slice-based metrics with the traditional code-based metrics over the two types of versions in the Linux kernel, which are development and stable versions. Focusing on the slice-based metrics, we found that Linux (in general) is growing in a super-linear rate, a linear rate in the stable versions, and exponential in the development versions. When looking at the slice size per file we find that it is decreasing for the development versions 2.1, 2.5 and the stable versions 2.2, 2.4. This is due to the higher rate of growth in number of files than slice lines, especially after version 2.0 (except for version 2.3). In addition, the growth is increasing for the development versions 1.1, 1.3, 2.3 and the stable versions 1.2, 2.0. This is due to the growth in slice lines that has a higher rate than that of files.

We also found that the growth of the system size (measured by both LOC, number of files) in the stable versions is much more smooth than in the development versions. For the average slice size per day metric, we found that over time the development versions have a higher rate of changes than the stable versions. However, the stable versions grow a little initially then become more stable. In addition, the general trend for the slice coverage metric is a decreasing one, where the values for stable versions are usually higher than of successive development versions, meaning they are more complex, and the values for development versions are more volatile, especially in early versions. The reasons for this can be real improvements and perfective changes in the development versions, or it is just an addition of more LOC/files in those versions. The comparison between the *arch*, *drivers*, and *include* directories and with the other core kernel directories shows that the three directories exhibit the largest change in the slice size over two types of versions (development, version 1.1 and stable, version 1.2).

In light of the above metrics we attempt to characterize the different types of maintenance activities in the Linux kernel. We found that the stable versions do not always have only corrective maintenance activities, but also reflect some large jumps and a little growth in the beginning. For the perfective maintenance activity, we found that the development versions reflect this, as there is a noticeable improvement of the complexity and maintainability over time in these versions. The adaptive maintenance activity is shown to be reflected by the arch, drivers, and include directories. It seems that the metric values over these directories are higher than other kernel's directories.

We have also attempted to identify whether some of Lehman's laws of software evolution are reflected in the evolution of the open-source Linux kernel using the slice-based metrics. We found there is evidence for Continuing Change, Continuing Growth, Invariant Work Rate, Declining Quality, Conservation of Familiarity, and Increasing Complexity. For other laws, we were not able to conclude any relationship; however we did not find contradicting evidence.

CHAPTER 5

A SLICE-BASED ESTIMATION APPROACH FOR MAINTENANCE EFFORT

This chapter addresses a direct application of our lightweight slicing technique: computing all the slices for an entire lifetime of version history. We seek to understand how those slices are used to build an estimation approach for maintenance effort. A case study of the GNU Linux kernel with over 900 versions spanning 17 years of history is presented. For each version the forward static slice of all variables is computed using our lightweight slicing approach. The slice size is computed as the total number of line, functions, or files in the slices. Changes to the system are then modeled using the difference between the slice sizes of two versions. The hypothesis is that this model is predictive of maintenance effort. The three different granularities of slice sizes (i.e., line, function, and file) are analyzed. The results demonstrate that the approach accurately predicts effort in a scalable manner.

5.1 Problem Statement

Systems must be maintained so as to remain useful [Lehman 1980] and estimating the amount of effort for particular maintenance tasks is a key aspect for any system (closed or open). Additionally, as systems grow, maintenance typically becomes more complicated and costly. Thus, the maintenance process should be well planned in advance through an accurate effort estimation of the maintenance tasks [De Lucia, Pompella, Stefanucci 2005; Yu 2006].

Traditionally, maintenance effort is calculated using historical process and coarse-grained system information such as person hours, number of tasks, and system size [Asundi 2005]. The predictor variables used to estimate this value typically compose a measures of the system size and complexity, productivity factors, as well as size and number of maintenance tasks [Binkley, Schach 1997]. However, the number of maintenance tasks is not known at the start time of the system, and must be estimated.

Conventionally, an estimation process for maintenance effort contains three steps, as follows:

1. Extract maintenance data, such as maintenance effort (person-hours), number of maintenance tasks, system size.
2. Build and validate the maintenance-effort model. Conventionally, this is a mathematical model that represents the maintenance effort as a function of other software measures. The model should be validated against additional maintenance data.
3. Predict future maintenance effort using the maintenance-effort model.

While using maintenance-task information is very attractive for managers of a typical closed-source system, who have to estimate the effort required maintaining the system as a number of developers, this approach is not that useful for larger corrective, adaptive, or perfective tasks during the system evolution of open-source system [De Lucia, Pompella, Stefanucci 2005]. In this case, the effort of a maintenance period greatly depends on the amount of source-code changes made to generate a new software version from an earlier operational version. For open-source systems, this data is not recorded or documented [Yu, Schach, Chen 2005]. Additionally, because of the nature and complexity of the maintenance tasks in open-source systems, there are many negatives to directly using effort-estimation models built on closed-source data. Hence, we cannot follow the same process to estimate maintenance effort.

However, the availability of the source code and history allow for other measures that are related to the maintenance effort. To this end we introduce a maintenance effort estimation based directly only on source code. It entails computing the slice for all the variables in a system and modeling how the slice changes over time.

Specifically, we identify and validate slice-based software measures and a corresponding process that can represent maintenance effort in open-source systems. We analyze 974 versions of Linux kernel, and construct, validate, and compare three indirect maintenance-effort models. The estimation approaches of maintenance effort are built and evaluated using residual-analysis statistics. Statistical measures include R^2, *adjusted-*R^2, $PRED_{25}$, $PRED_{50}$, *MMRE*, *MdMRE*, and *SPR* [Kendall, Stuart, Ord 1987; Jorgensen 1995]. The prediction results are encouraging and the production of the estimate is very scalable.

To the best of our knowledge, this is the first work applying a slice-based metric to build an estimation approach for maintenance effort in open-source systems. We consider the hypothesis that the historical source code changes can be used to regulate effort estimation approaches with a high sensible degree of predictive power. Furthermore, our work is the first to uncover the maintenance changes using slicing over a large amount of data of the Linux kernel by slicing versions from over 17 years.

5.2 Slice-Based Metrics

Many existing software metrics are computed only using syntactic information of the code and use that to model semantic information. For example, cyclomatic complexity is computed by counting the number of branch (i.e., conditionals) to infer semantic complexity. Semantic information is much more difficult to derive and model. For example, a semantic change in one function might create a ripple effect among other functions. In a maintenance context, the effort estimation is a function of the code that is to be (was) changed. To help identify such problem program slicers often applied and are valuable tool in determining side effects.

We note that the maintenance effort for open-source systems is not given as the number of person-hours expended as the case in closed-source system [Asundi 2005; Yu 2006]. However, it has been argued [Niessink, Vliet 1997; Niessink, Vliet 1998; Ramil, Lehman 2000; Yu 2006] that source-code changes in open-source systems could be used as an indirect measure for estimating maintenance effort. That is, the amount of source-code changes from version *k* (*base version*) at time *t* to version *k+1* (*evolved version*) at time *t+1* indirectly represents the effort spent maintaining the system from version *k* to version *k+1*. The amount of these changes can be viewed at three different granularities, line, function, and file, such as the number of lines, functions, and files that were added, deleted, and modified during the maintenance activity.

A number of researchers have observed that the source-code change can be found using textual, syntactic, or semantic differencing [Maletic, Collard 2004]. For example, previous studies [Ramil, Lehman 2000; De Lucia, Pompella 2002; De Lucia, Pompella, Stefanucci 2005; Yu 2006], determine the source-code change between two consecutive versions either from:

1. CVS logs,
2. Using some computer aided software (CASE) tools,
3. Or system utilities such as UNIX *diff*.

Chen et al [Chen, Schach, Yu, Offutt, Heller 2004] discussed the limitations of using the change logs to detect source-code changes in three open-source case studies. He shows that up to 78% of changes made to the source code are omitted from the system's change logs, and concludes that before using change logs as a research base for development and maintenance of open-source systems, experimenters should check carefully for errors and inaccuracies. Additionally, this tracking data is not always available. For example, the change logs for Linux kernel only started to be released after the major version 2.4 (version 2.4.1, January 29, 2001). That's why Yu [Yu 2006] in his study of the Linux kernel built two models to estimate the maintenance effort using the change logs for major versions 2.4 and 2.5 only, with a total of 121 versions.

Since here we are interested with the problem of finding the changes between two versions of the system that exhibits all changed behaviors of an evolved version with respect to the base version, textual differencing tools (*diff*) are not satisfactory because they cannot detect behavioral changes.

By using semantic-differencing approaches that depend on static-program analysis we can extract facts and other information from versions of the system. For example, static program slicing can be applied to the available source code of the version, focusing on selected aspects of semantics. This process removes from consideration parts of the program that are determined to have no effect upon the semantics of interest. It is possible to determine the parts with different behaviors by comparing the slices of the base and the evolved versions with respect to corresponding points. The assumption here is that if the slice of the evolved version at statement *S* differs from the slice of the base version at the same statement *S*, then by the mean of the slicing definition, statement *S* potentially exhibits behavior changes between the versions.

By using source-code slicing to measure source-code changes, we provide a detailed analysis of the impact of the changed code, as opposed to other methods which are based only on tracking information provided by developers.

We will first show how to measure forward slicing, and then show how the slice-based metrics are computed using this approach

5.3 Measuring Forward Slicing

We consider that the extent of source-code change in one version could indirectly represent the effort spent in this version. The source-code change measured based on the changes to the slicing profiles. We consider a change of interest to be any change which could have an effect on the size of the slice profile. An example of a change that is not of interest is changing some operator to a different operator. Examples of changes of interest include adding code, deleting code, or changing the variable used in a given context. Each of these changes would result in a change to at least one slice profile.

Unlike most other approaches, we perform slicing over all variables inside the system. That is, we compute a slice profile for each variable in the system. This is in contrast to slicing just a specific variable where when a statement is modified the statements in the slice may not be change.

```
(base version)

f/func/x/@ 1, slines {1, 2}

f/main/i/@ 1, slines {4, 5, 7, 9}, cfunctions {func}

f/main/sum /@2, slines {4, 5, 6, 8}

f/main/y/@ 3, slines {4, 9, 10}
```

```
(evolved version)

f/func/x/@ 1, slines {1, 2}

f/main/i/@ 1 slines {4, 5, 6, 7}

f/main/sum/@ 2, slines {4, 5, 6, 8, 9}, cfunctions {func}

f/main/y/@ 3, slines {4, 9, 10}, dvariables {sum}
```

Figure 5.1. Two system dictionaries, for two versions base and evolved.

Figure 5.1 shows a small sample of the system dictionary of two different versions (base and evolved) over the same file. As we can see, the slice profiles of variables *i*, *sum,* and *y* changed between the two versions. These changes can be seen in the evolved version by the function call *func* using the variable *sum* instead of variable *i* in the base version. In addition, the variable *sum* becomes a dependent variable for the variable *y*. That is, any change with the value of the variable *y* now affects the value of the variable *sum*.

By using the information stored in each slice profile, we can easily retrieve the size of the slice for each variable (denoted by *Sz*) in the system by accumulating the number of lines in the *slines*, *cfunctions*, *dvariables*, and *pointers* fields. In addition, the total slice size for each function (denoted by *MSz*), is the sum of individual slice sizes for each variable *v* in the function. If there are *n* variables inside a given function *m*, then the total slices size for function *m* is denoted by:

$$MSz(m) = \sum_{i=1}^{n} Sz(v_i)$$

And if there are *n* methods inside a given file *f*, then the total slice size for file *f* is denoted by:

$$FSz(f) = \sum_{i=1}^{n} MSz(m_i)$$

Finally, if there are *n* files in the system, then the slice size for the entire system is denoted by:

$$TSz = \sum_{i=1}^{n} FSz(f_i)$$

For example, from Figure 5.1 we can see that the *Sz* value for the variable *i* in the base version is 6 which includes 4 statement lines form the *slines* field of the variable *i*, plus 2 statement lines from the *slines* field of the function call *func()*. However, this value for the same variable in the evolved version is equal to 4 since as shown from the slicing dictionary the function call *func()* is deleted.

The number of statements that are added, deleted, or modified in the slice is extracted by considering the following source-code changes made over the base version:

- Adding a new line either inside an already existing function or a new line in a new function,
- Deleting a line from an already existing function or deleting the entire function,
- Modifying an already existing line.

Renamed variables are treated as added variables in the evolved version and as deleted variables from the base version. Both cases are considered and recorded in the system dictionary and may decrease or increase the slice size of the variable, and consequently the slice size of the enclosing function, file, and entire system. Therefore, the amount and the exact position of change could be determined using the size changes over the *Sz*, *MSz*, *FSz*, and *TSz* values from the base version to the evolved version.

In order to detect the modified variables, these variables must exist in both versions with the same path name. For instance, from Figure 5.2 which shows a snapshot of the system dictionary for two Linux versions in XML, we can see that the variables *tot_len* and *retval* have the same path in both versions (with filename *linux/fs/read_write.c*, and function name *do_readv_writev*). That is, the change over the *Sz* values for both variables is easily identified, where the *Sz* value for variable *tot_len* changed from 4 to 5, and for variable *retval* changed from 11 to 14. The variable *flag* in the evolved version is not detected as a modified variable between versions, because it does not exist in the base version. However, this variable is detected as an added variable with a new slice profile in the evolved version.

```
<Project>
<version name = "linux-2.2.23" TSz = "258059">

     <file name = "linux/fs/read_write.c" FSz = "286">

           <function name = "do_readv_writev" MSz = "79">

                  <variable name = "tot_len" Sz = “4”/>

                  <variable name = "retval" Sz = “11”/> …………..

           </function> …………..

     </file> …………..

</version>

<version name=” linux-2.2.24” TSz = "258865">

     <file name="linux/fs/read_write.c" FSz = "309">

           <function name="do_readv_writev" MSz = "87">

                  <variable name="tot_len" Sz = “5”/>

                  <variable name="retval" Sz = “14”/>

                  <variable name="flag" Sz = “7”/> …………..

           </function> …………..

     </file> …………..

</version>
</Project>
```

Figure 5.2. The system dictionary for the linux-2.2.23 (base version), and the linux-2.2.24 (evolved version), in its XML representation, values of Sz, MSz, FSz, and TSz are different between two versions.

In contrast, if a variable that exist at the base version is deleted from the evolved version, then this variable detected as a deleted variable. Finally, the same scenario similarly applies to added, deleted, and modified functions and files.

The extracted information of the slicing process is then categorized by the means presented in the next subsection.

5.4 Extract Slice-Based Metrics

We use the information on the forward slices generated as shown in previous section to calculate slice-based metrics. Here we consider three different granularities of the slice sizes (line, function, and file); consequently different levels of the slice-based metrics can be computed.

Unlike most other metrics, slice-based metrics are based on program slice information, which is of finer granularity than the measures in many other metrics. Program slices have the additional advantage of capturing program behavior, and hence the slice-based metrics are more directly related to the program behavior.

In order to build the slice-based maintenance-effort model, for each of the 974 versions of the Linux kernel, we extract ten measures from the source-code repository and the changes between slice profiles. These measures are described in Table 5.1. In these measures, *ΔsliceSize*, *Δfunction*, and *Δfile* are the extent of change between the two versions, and hence could be used to indirectly represent maintenance effort. We explain each of the slice-based metric items as follows.

Table 5.1. Code and Slice Based Extracted Measures.

Measure	Description
sliceSize	Total slice size measured in LOC
ΔsliceSize	Indirect maintenance effort at the system level, measured as the difference of slice sizes
Δfunction	Indirect maintenance effort at the function level, measured as the number of functions which contain modified slices
Δfile	Indirect maintenance effort on the file level, measured as the number of files which contain modified slices
LOC	Total size of the system measured in LOC
files	Total size of the system measured in number of files
LOC-g	Difference between LOCs for two consecutive versions
files-g	Difference between files for two consecutive versions
lag-time	Time duration between two versions in days
Scoverage	The slice coverage, the slice size relative to LOC

The first metric that we introduce is *sliceSize*, the slice size measured in LOC. For an individual slice this is just the *Sz* value measured at the above section. For a function and file, we summed the *sliceSize* of all slices of the variables inside the function (*MSz*) and file (*FSz*), respectively. For a system, we sum the *sliceSize* for all slices in the system (*TSz*).

For two versions of the system, we can then measure the difference between the *sliceSize*s. This forms a new metric *ΔsliceSize*. This gives us some idea of the growth of the system in terms of complexity, i.e., the *ΔsliceSize* metric represents the increase or decrease in the number of impacted statements.

Additionally, the number of modified slices between two versions is used to introduce two more metrics, *Δfunction* and *Δfile*. The metric *Δfunction* is the number of functions which contain modified slices, and the metric *Δfile* is the number of files which contain modified slices. These two metrics indicate how much the changed statements in a slice profile depend on each other by intra-procedural or inter-procedural control or data dependencies. A high *Δfunction* value indicates more logically complex code, and a high *Δfile* value may indicate that the changes in the system were very broad.

Each version of the system has its own release date. The *lag-time* (measured in days) measures the time between the start and the completion of a maintenance task. The *lag-time* includes the duration from the date when a base version is released, until the date the evolved version is released. The assumption here that the maintenance requests start when the base version is released, and the tasks are completed when the evolved version is released. That is, the *lag-time* is the sum of the individual times for each maintenance task in a version of Linux. Obviously, *lag-time* is related to maintenance effort. That is, an increase in *lag-time* is expected to indicate an increase in maintenance effort, and vice versa.

By comparing the slice size (*sliceSize*) to the system size (LOC), we can measure the slice coverage using the *Scoverage* metric [Weiser 1979]. This metric represents the active portion of the system and is included as a factor of maintenance activity. It is related to different types of maintenance because each type has a different effect on the maintenance effort [De Lucia, Pompella 2002; Hayes, Patel, Zhao 2004; De Lucia, Pompella, Stefanucci 2005]. For example, corrective maintenance requires more effort than the other types of maintenance.

Finally, since the size of the system is shown in the literature to be related to maintenance effort, we also extract the LOC and number of files. For two versions of the system, we can then measure the difference. This represents the system growth and forms the metrics *LOC-g* and *files-g*.

5.5 Slice-Based Metrics on the Linux kernel

As a way of showing the application of our indirect maintenance-effort metrics on a real system, we have applied the metrics to the Linux kernel. These metrics are then compared to traditional measures of code effort, e.g., LOC.

We analyzed 11 major versions containing 974 separate releases released over 17 years. Table 5.2 shows a summary of these major versions along with statistics related to the individual releases and their slices. The major versions are identified and ordered by their sequence number and release date. The column *Releases* is the total number of releases for each major version. The column *Files* shows the sum of the number of source files of each release in the major version. And the column *LOC* shows the sum of the individual LOCs for each release in the major version. The last column shows the total *sliceSize* for each release in the major version. As shown, this total slice size is ~2 billion LOC, with a slice size relative to LOC of 46.0%. The full sources of the kernel (*.gz*) files were downloaded from the official Linux kernel archives[5].

Table 5.2. A summary of Linux kernel versions data, 11 major versions with a total of 974 releases, where (S) = stable, (D) = development, version 2.6 includes both (S) and (D).

Major Version	Releases	Files	LOC	sliceSize
1.0 (S)	1	487	166,144	83,891
1.1 (D)	36	23,676	8,815,860	4,243,784
1.2 (S)	14	10,717	3,995,650	1,895,802
1.3 (D)	100	113,173	45,289,545	21,301,236
2.0 (S)	41	67,718	30,010,473	13,903,791
2.1 (D)	133	337,790	140,356,670	64,719,507
2.2 (S)	27	116,777	42,507,265	19,511,355
2.3 (D)	52	240,527	114,681,685	52,500,499
2.4 (S)	76	649,111	249,771,469	114,093,834
2.5 (D)	76	774,524	306,209,149	139,758,991
2.6(S, D)	418	8,146,805	3,469,166,975	1,580,414,630
Total	974	10,481,305	4,410,804,740	2,012,343,429

[5] The Linux Kernel archives: http://www.kernel.org.

We use the data extracted from the 11 major versions shown in Table 5.2 to indirectly build estimation models for maintenance effort. The models were built on data from 783 versions, and then validated on the maintenance data of 191 versions from major version 2.6

Table 5.3. Descriptive statistics of the collected measures.

Measure	Min	Max	Mean	Median	Standard deviation
sliceSize	83,696	3,781,285	1,372,048	1,167,153	1,095,625
ΔsliceSize	5.0	899,045	47,708	4,130	134,547
Δfunction	0.0	1,903	123.8	73.3	224.9
Δfile	0.0	597	32	14.7	67.9
LOC	165,768	8,300,297	3,002,400	2,551,821	2,409,270
files	487	19,604	7,554	5,068.4	6,101.6
LOC-g	8.0	1,977,000	105,119	9,148.1	295,832.6
files-g	0.0	2,434	34.3	28.9	105.4
lag-time	3,312	1,125	6.6	5.0	174.4
Scoverage	0.38	0.50	0.46	0.46	0.01

Table 5.3 summarizes the descriptive statistics of the collected measures. As we can see for some measures, such as *ΔsliceSize*, *Δfunction*, *Δfile*, *LOC-g*, *files-g*, and *lag-time*, the standard deviation is greater than the corresponding mean, indicating that the data are widely spread.

For example, for *ΔsliceSize* the minimum value is 5 and the maximum value is 899,045 representing an extremely wide range. It is worth noting that the wide range of data does not indicate a fault in the measurements. In other words, these measurements do not necessarily need to be a certain value or within a certain range.

It is common for historical datasets to contain a considerable number of missing values, and several techniques have been developed to deal with this [De Lucia, Pompella, Stefanucci 2005]. The main advantage of our dataset is that it does not contain missing values. This is because our data is the source code, and no external metadata or other records are used.

5.6 Slice-Based Maintenance Effort Models

As discussed in Section 5.1, building an accurate maintenance-effort estimation model should be derived from accurate maintenance-effort data, which is rarely recorded for open-source, and many closed-source, systems [De Lucia, Pompella, Stefanucci 2005; Yu 2006]. Therefore, we cannot apply an effort-estimation model built from a closed-source system directly to an open-source system because the absence of maintenance-effort data prevents validation. Alternatively, we take the following approach:

Phase 1: Identify measures that are theoretically related to and can indirectly represent maintenance effort. The candidate measures should be available for most systems, both closed and open source. If such measures can be found and validated, we can construct an indirect model for maintenance effort and use it to predict the indirect maintenance effort of open-source systems.

Phase 2: Extract the maintenance data. The data include indirect maintenance-effort identified and validated in previous phase (aka *dependent variables*) and the data of other related measures that can be used to predict the indirect maintenance effort (aka *independent variables*). For example, if we identify source-code changes from version *k* to version *k+1* as the indirect maintenance effort, then LOC change between both versions is a measure of source-code change. In this paper, we use slice-based changes to measure the indirect maintenance effort, so *sliceSize* change is a measure of source-code change.

Phase 3: Validate the correlation between the dependent variables and independent variables. We used Spearman's rank-correlation coefficient since there are no assumptions regarding the underlying distribution of the data, and its use is recommended for hypothesis testing when the number of data points exceeds 30 [Binkley, Schach 1997]. Strong correlation means that the independent variables can be used to indirectly represent maintenance effort; weak correlation indicates the measure is not eligible to represent maintenance effort.

Phase 4: Multiple linear regression analysis is used to build the effort-prediction approach. Specifically, the indirect maintenance-effort is represented as a function of other related measures. We validate this approach against collected maintenance data from the Linux kernel. In addition, we show how we can improve this approach by considering three different granularities of slice sizes.

Phase 5: Predict the indirect maintenance effort based on the models built in the previous phase.

The indirect maintenance effort can be represented at three levels, namely line level, function level, and file level. Therefore, the dependent variables are *ΔslizeSize*, *Δfunction*, and *Δfile*, and the independent variables are LOC, *files*, *sliceSize*, *files-g*, *LOC-g*, *lag-time*, and *Scoverage*. Table 5.4 shows Spearman's rank correlations between the dependent variables and independent variables based on the maintenance data of 783 versions of Linux. The correlation coefficient that is statistically significant at the 0.01 level (*2-tailed*) is shown in bold. The strong linear relations are not necessarily significant, since the significance is specified by the *p-value*.

From Table 5.4, we can distinguish multiple significant linear correlations between the three dependent variables and some of the independent variables. The *ΔsliceSize* and *Δfile* are significantly correlated with *files-g* and *LOC-g*. There is also statistically significant linear correlation between *Δfunction* and the four measures (LOC, *files*, *LOC-g*, and *sliceSize*).

Based on this observation, we built three indirect maintenance-effort estimation models to represent the line-, function-, and file-level changes. These models consider only those independent variables that have significant correlations with the dependent variable at the 0.01 level.

The three models are:

(1) $\Delta sliceSize = c_1 + c_2\,(LOC\text{-}g) + c_3\,(files\text{-}g)$

(2) $\Delta file = c_1 + c_2\,(LOC\text{-}g) + c_3\,(files\text{-}g)$

(3) $\Delta function = c_1 + c_2\,(LOC\text{-}g) + c_3\,(sliceSize) + c_4\,(files) + c_5\,(LOC)$

Table 5.4. The correlations between dependent variables and independent variables based on the training dataset (783) versions, significant at 0.01 level is shown in bold.

Independent Variables	Dependent Variables		
	Δfunction	*Δfile*	*ΔsliceSize*
LOC	**-0.358**	-0.119	-0.073
files	**-0.362**	-0.121	-0.072
files-g	0.026	**0.491**	**0.750**
LOC-g	**0.234**	**0.805**	**0.702**
sliceSize	**-0.367**	-0.129	-0.081
lag-time	0.122	0.169	0.214
Scoverage	0.142	0.127	0.219

The c_1 variable represents the constant factor or the intercept, which characterizes the height of the regression line when it crosses the y-axis where the dependent variable is plotted, or we can say that the c_1 represent the predicted value of the dependent variable when all the independent variables are equal to zero. The c_i (where i = 2 to 5) represents the slope of the line regression which indicates the sensitivity of the dependent values to the changes in the independent values. That is c_i represent the increase or decrease in y for each unit change in x. For example, in model (3) the coefficient for the LOC is equal c_5, so for every unit decrease in the LOC a c_5 unit increase in *Δfunction* is predicted, when holding all other variables constant.

The three effort-estimation models are linear, and linear regression is used to estimate the coefficient. Table 5.5 shows the linear regression analysis of the models.

The *p-value* demonstrates the ability of the independent variable to have a significant predictive capability in the presence of other variables. If the independent variable has a non-significant *p-value*, then we can remove this variable and refit the model again, since this variable does not have predictive capability in the presence of other independent variables. If adding a new independent variable can improve the accuracy of the model, then this variable is said to have the predictive capability.

Table 5.5. Linear regression analysis of the three indirect effort estimation models, not significant at 0.01 levels is shown in bold.

Model	Independent variable	*ci*	*p-value*	R^2	*adjusted-*R^2
(1)	LOC-g	0.029	0.004	0.424	0.415
	files-g	104.429	0.000		
(2)	LOC-g	0.006	0.000	0.963	0.962
	files-g	0.238	**0.054**		
(3)	sliceSize	0.007	0.001	0.469	0.452
	files	-0.564	**0.067**		
	LOC	-0.002	**0.013**		
	LOC-g	0.006	0.000		

As can be seen from Table 5.5, the independent variable *files-g* in model (2) does not have the significant predictive capability at the 0.01 level (*p-value* = 0.054). And the same case occurs with the independent variables *files* and LOC in model (3). Therefore, we need to refit these two models by either removing the *files-g* variable from model (2) and removing *files* and LOC variables from model (3), and/or adding new independent variables.

Table 5.6 shows the results after fitting the two models. In model (2) we remove the *files-g* variable and adding the independent variable *lag-time*. In model (3) we removed the *files* variable and we added two other variables *lag-time* and *Scoverage*. In this case, as can be seen from the table, all the independent variables in model (2) and model (3) have significant predictive power.

Table 5.6. Refit linear regression analysis of Models (2) and (3).

Model	**Independent variable**	***ci***	***p-value***	R^2	*Adjusted-*R^2
(2)	LOC-g	0.006	0.000	0.968	0.968
	lag-time	0.726	0.000		
(3)	LOC-g	0.005	0.000	0.609	0.593
	sliceSize	-0.014	0.004		
	LOC	0.007	0.004		
	lag-time	2.836	0.000		
	Scoverage	91.653	0.000		

The models after refitting models (2) and (3) are:

(2) $\Delta file = c_1 + c_2\,(LOC\text{-}g) + c_3\,(lag\text{-}time)$

(3) $\Delta function = c_1 + c_2\,(LOC\text{-}g) + c_3\,(sliceSize) + c_4\,(lag\text{-}time) + c_5\,(LOC) + c_6\,(Scoverage)$.

The R^2 coefficient of determination value is important to determine whether or not the regression model was helpful. If the regression line provides an estimate of the predictable values that closely match the observed values, then the R^2 value will be close to one (the better the data fits the model), and with zero indicating no relation between independent and dependent variables.

The *adjusted-*R^2 that adjusts for the number of independent variables in a model is also calculated. The value of *adjusted-*R^2 only increases if a new independent variable improves the model more than would be expected by chance.

From Table 5.5 and Table 5.6, we can see that model (2) has both larger R^2 and *adjusted-*R^2 values than model (1) and model (3), which means, based on the data of 783 versions, model (2) is more accurate than other two models in predicting the indirect maintenance effort. Unfortunately, a high value of R^2 does not guarantee the goodness of the model and does not indicate whether the appropriate independent variables have been used in the model or not [De Lucia, Pompella 2002; De Lucia, Pompella, Stefanucci 2005]. In addition, the high values of R^2 and *adjusted-*R^2 do not assess the quality of future prediction, but only the capability of fitting the sample data. That is, if an effort estimation model is developed using a particular dataset and the accuracy of this model is evaluated using the same dataset, then the value obtained will be optimistic. The error will be low and will not represent the performances of the model on future datasets.

5.7 Evaluating Model Performance

To study the quality of the three models for future predictions, we apply the models to predict the indirect maintenance effort of 191 versions from major version 2.6.

These versions range from version 2.6.25.3 released May, 10 2008 to version 2.6.37.1 released Feb, 17 2011. The predicted results and the actual observed measurements are compared to study the accuracy of predictions.

Model validation is the most important step in the model building process. The validation of a model often consists of the analysis of residuals [Ramil, Lehman 2000; De Lucia, Pompella, Stefanucci 2005; Yu 2006]. The residual represents the difference between the predicted value estimated by the model and the observed value of the dependent variable.

Our residual analysis includes:

- *SPR statistics*: is the sum of absolute value of the residuals (e.g., prediction errors). That is, the $SPR = \sum_k | Observed_k - Predicted_k |$.
- MRE statistics: the magnitude relative error, which include the MMRE (mean magnitude relative error), and MdMRE (median magnitude relative error). The MRE is defined as: MRE k = (|Observed k – Predicted k|) / Observed k. The MdMRE is calculated, since the MMRE is known to be very sensitive to the extreme values, such as a few very high relative error MRE values could influence the overall result.

Other indicators commonly used to evaluate the prediction model based on *MRE* are the percentage of prediction at specific level *PRED*, which measures the percentage of predicted values within *X*% of the observed values. The value of *X* is suggested in [Conte, Dunsmore, Shen 1986] to be at least 25% and a good prediction model should predict 75% of the observed values. The two variants of the measure *PRED* we calculated are:

- *$PRED_{25}$*: the number of predicted values for which *MRE* was less than or equal to 25%.
- *$PRED_{50}$*: the number of predicted values for which *MRE* was less than or equal to 50%.

The predicted results and the measurements are compared to study the accuracy of the predictions. Figure 5.3, Figure 5.4, and Figure 5.5 illustrate the comparisons of the predictions and measurements for *ΔsliceSize*, *Δfile*, and *Δfunction*, respectively. In these figures we plotted the *ΔsliceSize*, *Δfile*, and *Δfunction* values (observed and predicted) on the y-axis with the version date on the x-axis.

Table 5.7. Model Predictive Performances over 191 releases in the test dataset, as seen Model (2) outperforms other models.

Measure	Model (1)	Model (2)	Model (3)
PRED25 %	33.3	43.9	33.4
PRED50 %	47.4	64.9	50.9
SPR	39451.6	448.5	2960.1
MdMRE %	53.4	30.2	45.8
MMRE %	74.1	44.2	63.9

On a per-model basis, it is clear that the predicted results are fairly often comparable with the actual observed measurements. However, it is not visually noticeable that one model outperforms another. For this reason, the class-of-models measurement (discussed above) was performed to assess the quality of future predictions and evaluate their performances quantitatively. The results of the application of these measures over the 191 versions test dataset are shown in Table 5.7.

The coefficient of *files-g* has the largest value in Table 5.5and Table 5.6, thus indicating that the number of Linux files has a greater influence on system complexity and the maintenance effort.

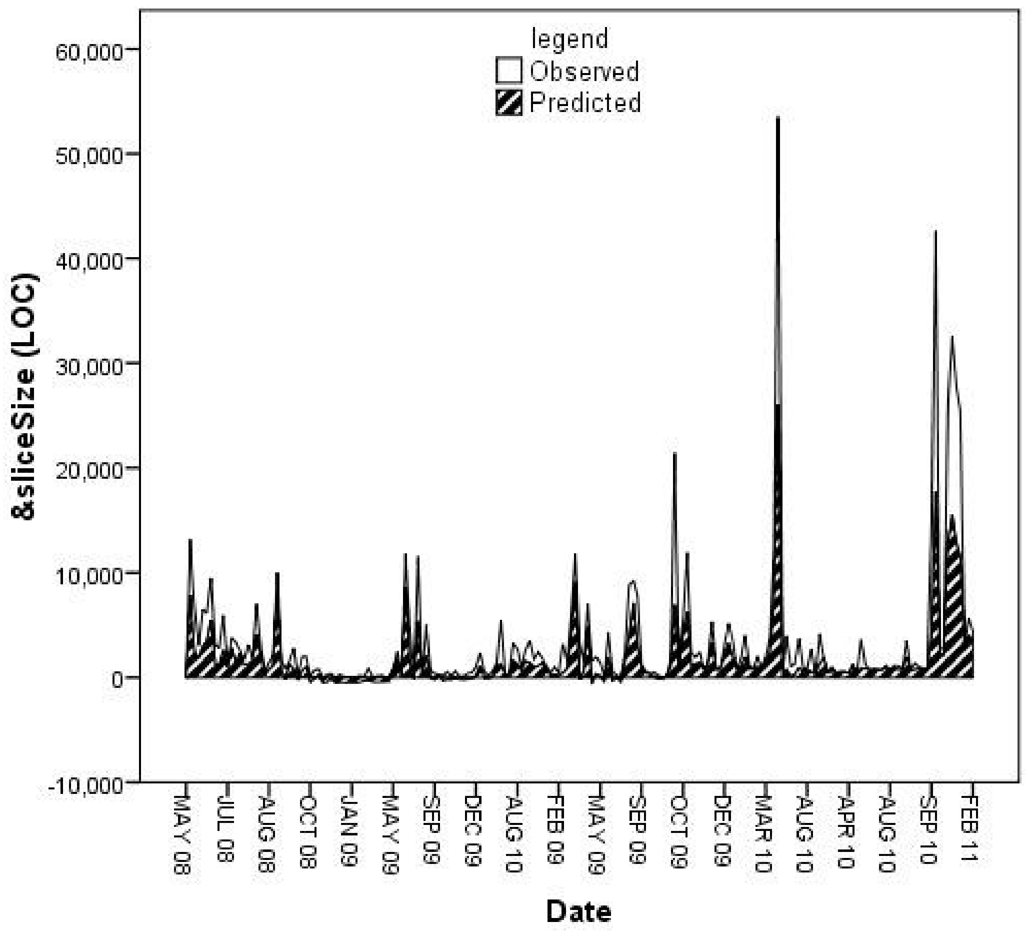


Figure 5.3. ***ΔsliceSize*** **Observed and Predicted values in 191 versions using model (1), PRED$_{25}$ = 33.3, PRED$_{50}$ = 47.4.**

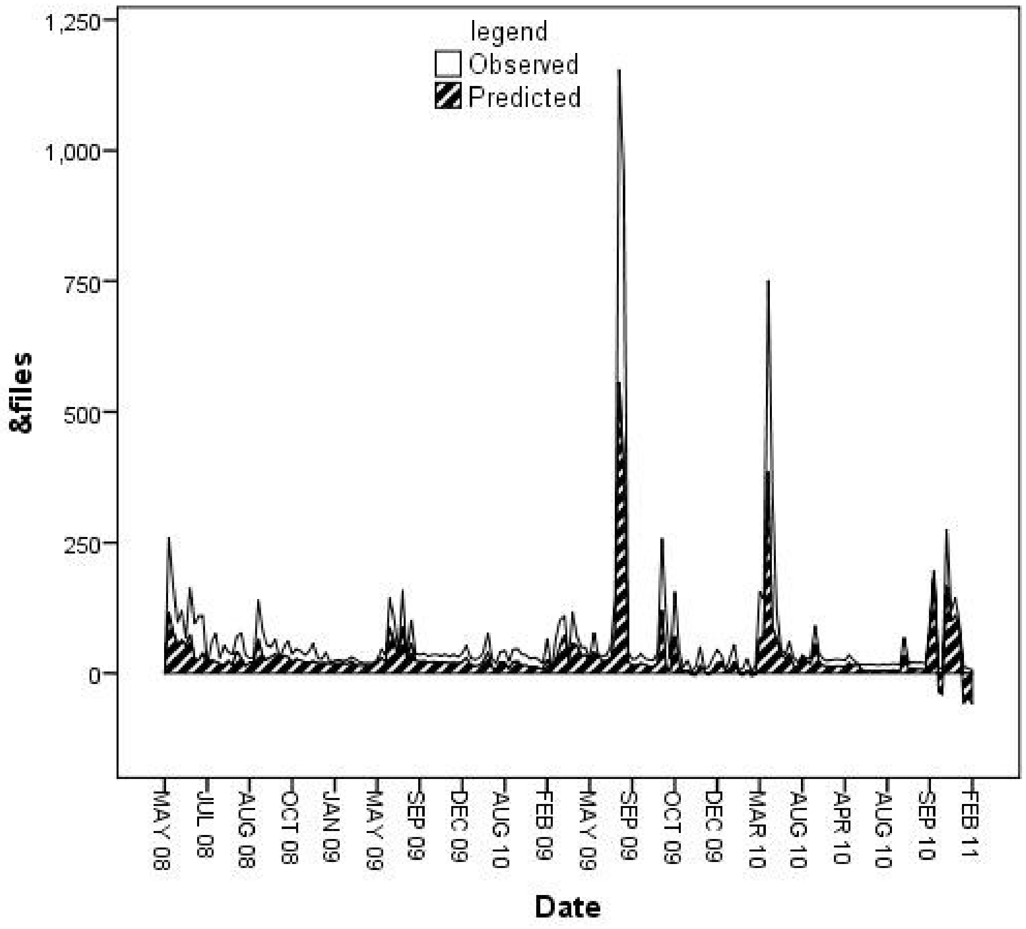


Figure 5.4. ***Δfile*** **Observed and Predicted values in 191 versions using model (2), $PRED_{25} = 43.9$, $PRED_{50} = 64.9$.**

As expected from the analysis of R^2 values for model (1) in Table 5.5 and the refitted models (2) and (3) in Table 5.6, Table 5.7 shows that models (2) and (3), which include the *lag-time*, generated higher R^2 values than did model (1). Moreover, model (2) performs better than model (3). In particular, model (2) predicts ~44% of the cases within a relative error less than 25% and ~65% of the cases within relative error less than 50%.

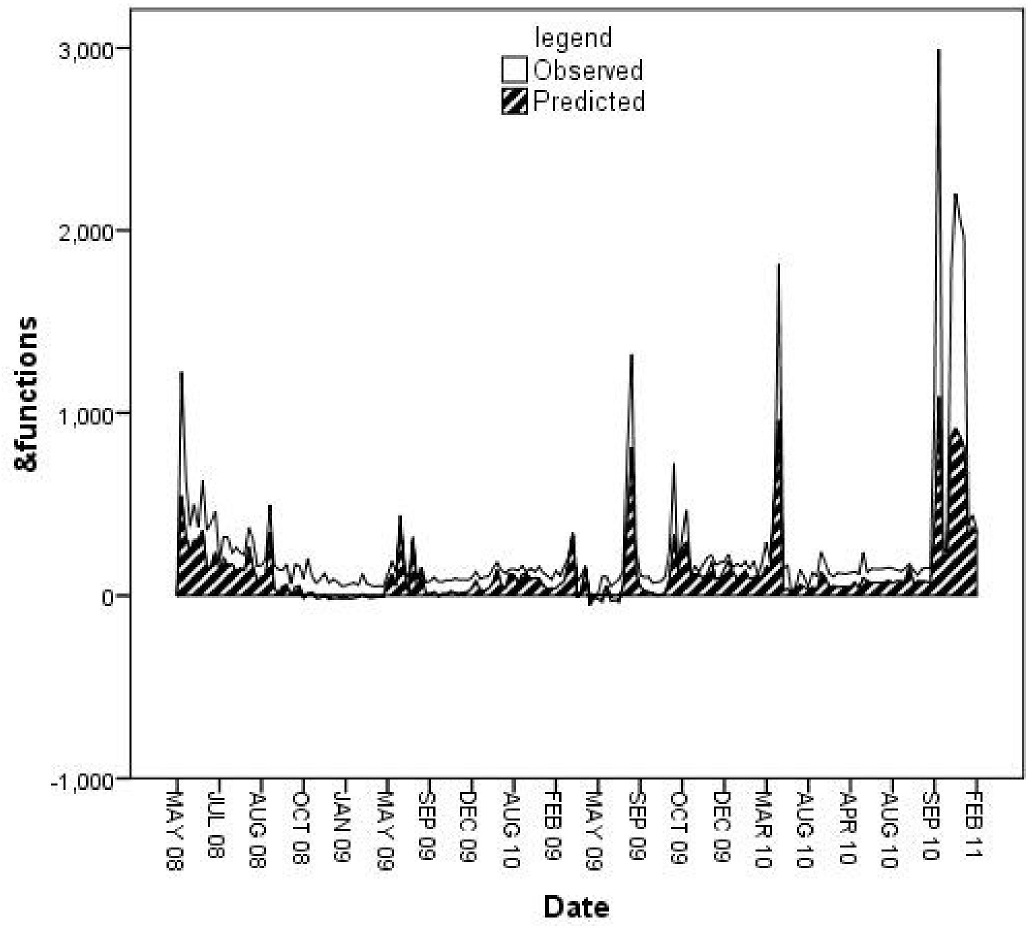


Figure 5.5. ***Δfuncions*** **Observed and Predicted values in 191 versions using model (3), PRED$_{25}$ = 33.4, PRED$_{50}$ = 50.9.**

The *MMRE*, *MdMRE*, *SPR*, *PRED$_{25}$*, and *PRED$_{50}$* values indicate that model (2) is more accurate than models (1) and (3) in predicting the indirect maintenance effort in the Linux kernel. This also implies that the number of files containing modified slices (*Δfile*) is more accurate in representing the complexity of the system than the number of functions containing modified slices (*Δfunction*) and the number of lines in the slices (*ΔsliceSize*). From model (2) we could say that the maintenance effort not only depends on the growth size of LOC itself, but also depends on the *lag-time* spent developing LOC, as model (2) integrates both the *LOC-g* and the *lag-time*.

5.8 Threat to Validity and Limitations

There are several threats to the construct validity of this study. The first threat to the statistical validity comes from the outliers in the data. An outlier is an observation that its value lies out of the overall pattern of a distribution. Outlier points can therefore indicate faulty data where a particular model might not be valid. When performing linear regression fitting to data, it is often preferable to discard outliers before computing the model of best fit. That is, there are two ways to determine the outliers:

1. Study the scatter plots of the dependent variables against the independent variables, and
2. Study the distribution of a variable.

However, our measures are not limited to a certain value; we cannot simply come to a decision this is an outlier just because the measurement is far from the mean value. Similarly, without supporting evidence and by just looking at the scatter plots of dependent variables against independent variables for our dataset, we cannot determine the outliers either. Therefore, we had no proof of outliers in our datasets.

Removing some data points in order to improve the prediction accuracy of the models is not meaningful. Hence, we did not remove any data points. However, if more information about the Linux versions and the supporting data was available, and we could remove some confirmed inaccurate data, the performance of the models will be drastically improved.

The major threat in building the indirect maintenance-effort models comes from the difference between the closed-source and open-source systems. Since, our prediction models depend on source-code measurement to predict the volume of changes as an indication of the maintenance effort. However, this is not the case for closed-source systems which used person-hours as a metric for maintenance effort.

5.9 Related Work

Many approaches to the effort-estimation problem have been derived using different assumptions, data sources, and methods to process the data to estimate the effort in the context of maintaining strictly managed and closed-source systems [Ramil, Lehman 2000; Yu 2006]. These models can be categorized into three main categories namely *Analogy*, *Delphi*, and *Parametric* [Shepperd, Schofield, Kitchenham 1996]. The first two categories derive the estimation models based on the past experience of similar systems, or using expert opinions. In contrast, Parametric effort estimation models involve the construction of statistical models from empirical data, e.g., using regression analysis on available data. Moreover, the Parametric models mathematically relate the effort and duration (e.g., days) to the variables that influence them.

Boehm et al [Boehm 2002] was the first to presents an algorithmic software cost estimation model named the constructive cost model COCOMO. In [Boehm, Clark, Horowitz, Westland, Madachy, Selby 1995] the same author extended the COCOMO model to estimate the maintenance effort by using a size-change factor to estimate the development effort. This factor represents the estimation of the size of changes expressed as the fraction from the total size of the system measured in LOC, this factor is change over a year period. De Lucia et al [De Lucia, Pompella, Stefanucci 2005] called this factor the "*annual change of traffic*" since this metric estimates the total software LOC changes during the year. Another work based on the size of changes is presented by Hayes et al [Hayes, Patel, Zhao 2004] who built a model for adaptive-maintenance effort using the changed LOC and the number of operators changed.

An estimation model based on historical data and previous project experience is proposed and discussed in [Shepperd, Schofield, Kitchenham 1996]. In this estimation model, the condition statements of the in progress software project is evaluated using a historical maintenance dataset from previous closed-source projects.

Belady and Lehman [Belady, Lehman 1972] suggest a model to approximate the cost and effort of releasing a new version from an old one. The suggested model estimates the efforts that are related to both the functionality updating and anti-regressive activities.

The maintenance-effort estimation that involves the convention of linear regression analysis was introduced by De Lucia et al [De Lucia, Pompella 2002]. In this research, the authors claimed that the types of the different maintenance tasks should be considered to improve the outcomes of the estimation model being used.

Jorgensen [Jorgensen 1995] derived different estimation models for maintenance effort using log linear regression, neural networks, and pattern recognition. He compares the prediction accuracy of these models using an industrial dataset. All the models estimate the size of the system measured in the summation of added, deleted, and modified LOC during the maintenance phase. Another linear model based on the size and the number of maintenance tasks is proposed in [De Lucia, Pompella, Stefanucci 2005], furthermore, other work done by Niessink et al [Niessink, Vliet 1997; Niessink, Vliet 1998] use linear regression analysis to extract estimation based on function points.

Coarse granularity measures have an impact on predicting required changes during the maintenance activities of the software project. For example, Lindvall [Lindvall 1998] demonstrates that the number of classes outperform the finer grained metrics in change prediction. In contrast, non-linear cost estimation models were proposed by several researches. For example, in [Eick, Graves, Karr, Marron, Mockus 2001] a code decay and a related number of measurements were illustrated to construct a non-linear changes prediction model.

Because of the nature and complexity of the maintenance tasks in open-source systems, there are many negative aspects to using existing effort estimation models directly for open-source projects. Little work of maintenance-effort estimation has been conducted for open-source systems. The major guidelines and tips to build an estimation model in these crucial systems are reported in [Asundi 2005].

Yu [Yu 2006] derived two indirect maintenance-effort models for the Linux kernel system using multiple linear regression. Nevertheless, these estimation models are based on and used factors which are derived from the closed-source software projects. In addition the validation process determined using the recorded maintenance information from closed-source systems, i.e., both estimation models depend on the number of maintenance tasks for the next revision of the system. Therefore, the models are not applicable if the maintenance tasks for the next revision are not included.

5.10 Chapter Summary

In this chapter, we presented a large-scale empirical study aimed at building an indirect maintenance-effort estimation models for open-source systems. The dataset used was obtained from the Linux kernel and used as a case study to build and validate the models performance using multivariate linear regression models. Our proposed maintenance-effort estimation models are able to accurately determine the source-code changes based only on the source code. Our proposed models estimate the maintenance effort based at the amount of changes made maintaining the system.

It is worth noting that we did not construct a direct maintenance-effort model (person-hours) for open-source systems. However, we decided to use the available source code, because:

1. There is limited direct maintenance-effort data available for open-source systems and we therefore cannot validate the correctness of such a model, and
2. Maintenance effort represented as person-hours is less meaningful for open-source systems. We are more interested in the amount of change on source code and the lag-time to perform the maintenance task.

In order to perform the forward slicing for multiple versions of large systems (e.g., Linux) we used a lightweight forward static slicing approach. That is the main reason that the analysis over all the Linux versions was even possible. A partial parsing of the source code is done (source statements are in interest) and the time and space required is small and scalable.

Our future research will study other open-source systems to determine more measures that can be used to indirectly represent maintenance effort and construct new more accurate prediction models.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

This dissertation addresses several practical concerns regarding the maintenance and evolution of open-source large-scale software systems. A lightweight program slicing technique is introduced and demonstrated. The technique is also shown to be highly scalable to large-scale systems (e.g., Linux kernel). Given this highly scalable method, new slice-based metrics that reflect the maintenance activities and supports some of laws of software evolution are derived and evaluated. Lastly, the technique is used to estimate the maintenance effort in open-source systems.

A lightweight forward static slicing approach and a tool (*srcSlice*) were developed that generates slicing for all variables in a given system. The approach does not require complete and compiling-able programs, since the approach uses the srcML format and toolkit. It also does not compute the program dependence graph for the entire program but instead dependence information is computed as needed (on-the-fly) while computing a slice on a given variable. The results demonstrate that the tool produces accurate slices as compared to an existing industrial tool (*CodeSurfer*) and is very efficient and highly scalable.

An empirical study is presented to understand and investigate how slices are changed during the evolution of the Linux kernel. The *srcSlice* tool was used to generate slices for every individual variable in the Linux kernel over its seventeen years of history, with more than a total lines of code ~4.4 billion LOC. It was observed that the slice size increases proportionally with the system size and the average slice size relative to LOC is 46.0%.

The data obtained by slicing the Linux kernel was used to identify and validate new slice-based software metrics for maintenance effort. These metrics were used to build and validate a maintenance-effort estimation models and a corresponding process that represents maintenance effort in open-source systems. The results show that these models are able to accurately determine source code changes based only on the source code. These models designed to indirectly estimate the maintenance effort spent maintaining the system, especially open-source systems.

The slice-based metrics are evaluated by applying them to characterize the different types of maintenance activities (corrective, perfective, and adaptive) and they are shown to support some of the Lehman's laws of software evolution in open-source systems. It was observed that some of these laws are reflected by the slice-based metrics.

To the best of our knowledge, this is the first work applying slice-based metrics in order to build an estimation model for maintenance effort in open-source systems. Furthermore, our work is the first to uncover the different maintenance activities using slicing over a large amount of data. The results demonstrate that the approach and the estimation models accurately predict the maintenance effort in open-source system in a highly scalable manner, and with a high rank of predictive power.

The work presented in this dissertation provides a solid foundation for the further explanation of problems related to the maintenance of large-scale systems. Future work supported by the research presented in this dissertation includes:

- Investigate how system slices change over the entire history of a large software system, and how slices reflect different types of changes occurring in a system possibly identifying refactoring changes.
- Investigating different slice-based metrics in the context of coupling and cohesion.
- Studying other large-scale open-source systems to determine more measures that can be used to represent the maintenance activities, and consequently the maintenance effort and construct new more accurate prediction models.
- Using the system slices change of the source code to leverage the details about the system's changes. Then each change in the slice will be analyzed to see if it meets a set of criteria that categorizes changes as design altering or not. The objective is to automatically determine if a given source code change (in the context of slice change) impacts the design of the system. This allows code-to-design traceability to be consistently maintained as the system evolves.

APPENDIX A

XML TTRANSLATION INOFRMATION

Table 6.1. srcML Translation Information for the 16 Open-Source Programs using *src2srcml* toolkit.

Program	Version	Number of Files			
		Translated	Skipped	Error	Total
ed	1.2	10	219	0	229
ed	1.6	10	201	0	211
wdiff	0.5	13	27	0	40
which	2.20	14	29	0	43
barcode	0.98	18	64	0	82
enscript	1.4.0	52	134	0	186
enscript	1.6.5	107	386	0	493
enscript	1.6.5.1	107	387	0	494
enscript	1.6.5.2	107	387	0	494
a2ps	4.10.4	188	584	0	772
findutils	4.4.2	314	832	0	1146
cvs	1.12.10	340	440	0	780
acct	6.5	27	48	0	75
dico	2.2	332	652	0	984
make	3.82	58	303	0	361
radius	1.0	196	382	0	578
Total		**1893**	**5075**	**0**	**6968**

APPENDIX B

SLICE INTERSECTION COMPARISON

Table 6.2. Intersected slice over 9 files from enscript-1.6.1, where (%) in the CodeSurfer (CS) and srcSlice (sS) columns is the slice size relative to LOC, (%) in the intersection column is the intersected slice relative to both tools slice size, (SM) is the relative safety margin for a slice.

enscript-1.6.1	Size	Slice Size						Intersection		
File Name	SLOC	CodeSurfer (CS)			srcSlice (sS)			Lines	sS %	CS %
		Lines	%	SM	Lines	%	SM			
\src\psgen.c	1868	1235	66.1	1.73	794	42.5	1.11	713	89.8	57.7
\src\util.c	1406	1028	73.1	1.49	720	51.2	1.05	688	95.6	66.9
\src\main.c	1291	960	74.4	1.61	638	49.4	1.07	596	93.4	62.1
\compat\alloca.c	224	39	17.4	7.80	55	24.6	11.00	5	9.1	12.8
\afmlib\strhash.c	267	124	46.4	1.46	144	53.9	1.69	85	59.0	68.5
\afmlib\afmparse.c	759	501	66.0	1.62	313	41.2	1.01	310	99.0	61.9
\states\lex.c	1475	695	47.1	7.81	162	11.0	1.82	89	54.9	12.8
\states\gram.c	1084	397	36.6	2.32	271	25.0	1.58	171	63.1	43.1
\afmlib\afm.c	584	440	75.3	1.24	355	60.8	1.00	355	100	80.7
Total	8958	5419			3452			3012		
Average	995.3	602.1	55.8	3	383.6	39.9	2.4	334.7	73.8	51.8
Min	224	39	17.4	1.24	55	11	1	5	9.1	12.8
Max	1868	1235	75.3	7.81	794	60.8	11	713	100	80.7

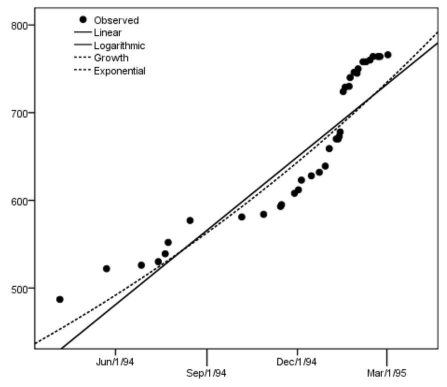
APPENDIX C

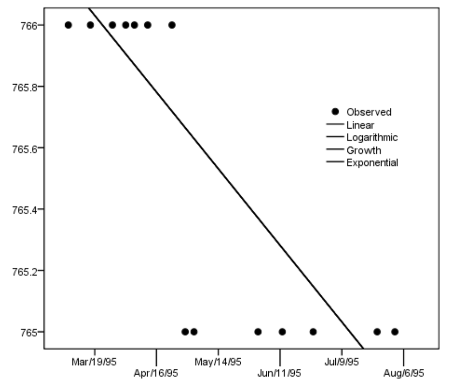
LINUX KERNEL VERSIONS

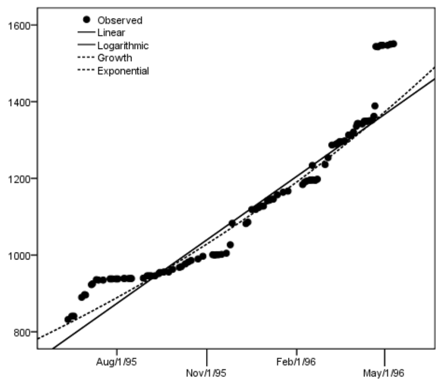
Figure 6.1 shows the growth in the number of files in the Linux kernel, and Figure 6.2 shows the growth in the LOC values in the Linux kernel. Both figures ran over 10 major versions with a total of 974 releases spent over 17 years of the history.

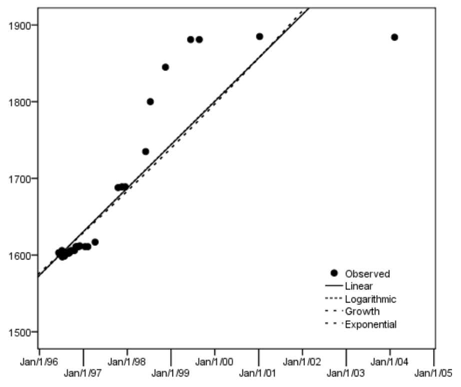
Table 6.3 shows the Linux kernel versions, from version 1.0.0 to version 2.6.37.1. All the kernel ("*dot"-bz2*) files were downloaded from www.kernel.org website, and sliced using the *srcSlice* tool. In total, Table 6.3 examines a total of 974 stable and development versions. In each version the study examines a variety of parameters such as: number of code lines (LOC), lag-time (the duration from the date when a base version is released, until the date the evolved version is released, measured in days), slice size measured in LOC, etc.

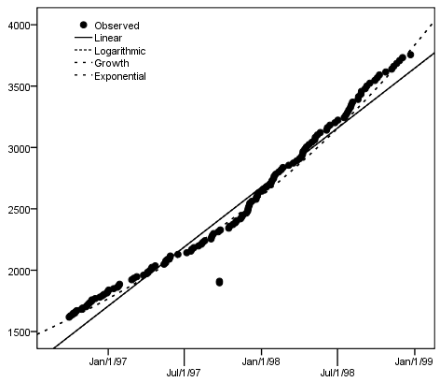
The version publish date (*Date* column) was examined by reviewing the most updated modified file in the source code root directory.

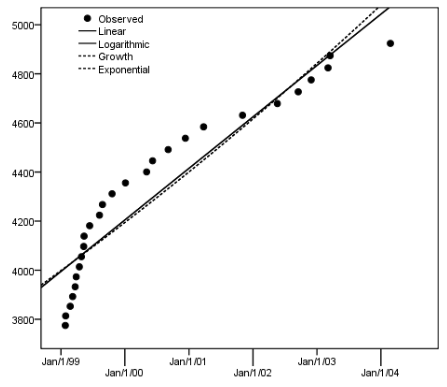

● Observed
Linear
Logarithmic
Growth
Exponential
800
700
600
500
Jun/1/94
Sep/1/94
Dec/1/94
Mar/1/95


766
765.8
765.6
765.4
765.2
765
● Observed
Linear
Logarithmic
Growth
Exponential
Mar/19/95
Apr/16/95
May/14/95
Jun/11/95
Jul/9/95
Aug/6/95


● Observed
Linear
Logarithmic
Growth
Exponential
1600
1400
1200
1000
800
Aug/1/95
Nov/1/95
Feb/1/96
May/1/96


1900
1800
1700
1600
1500
● Observed
Linear
Logarithmic
Growth
Exponential
Jan/1/96
Jan/1/97
Jan/1/98
Jan/1/99
Jan/1/00
Jan/1/01
Jan/1/02
Jan/1/03
Jan/1/04
Jan/1/05


● Observed
Linear
Logarithmic
Growth
Exponential
4000
3500
3000
2500
2000
1500
Jan/1/97
Jul/1/97
Jan/1/98
Jul/1/98
Jan/1/99


● Observed
Linear
Logarithmic
Growth
Exponential
5000
4800
4600
4400
4200
4000
3800
Jan/1/99
Jan/1/00
Jan/1/01
Jan/1/02
Jan/1/03
Jan/1/04

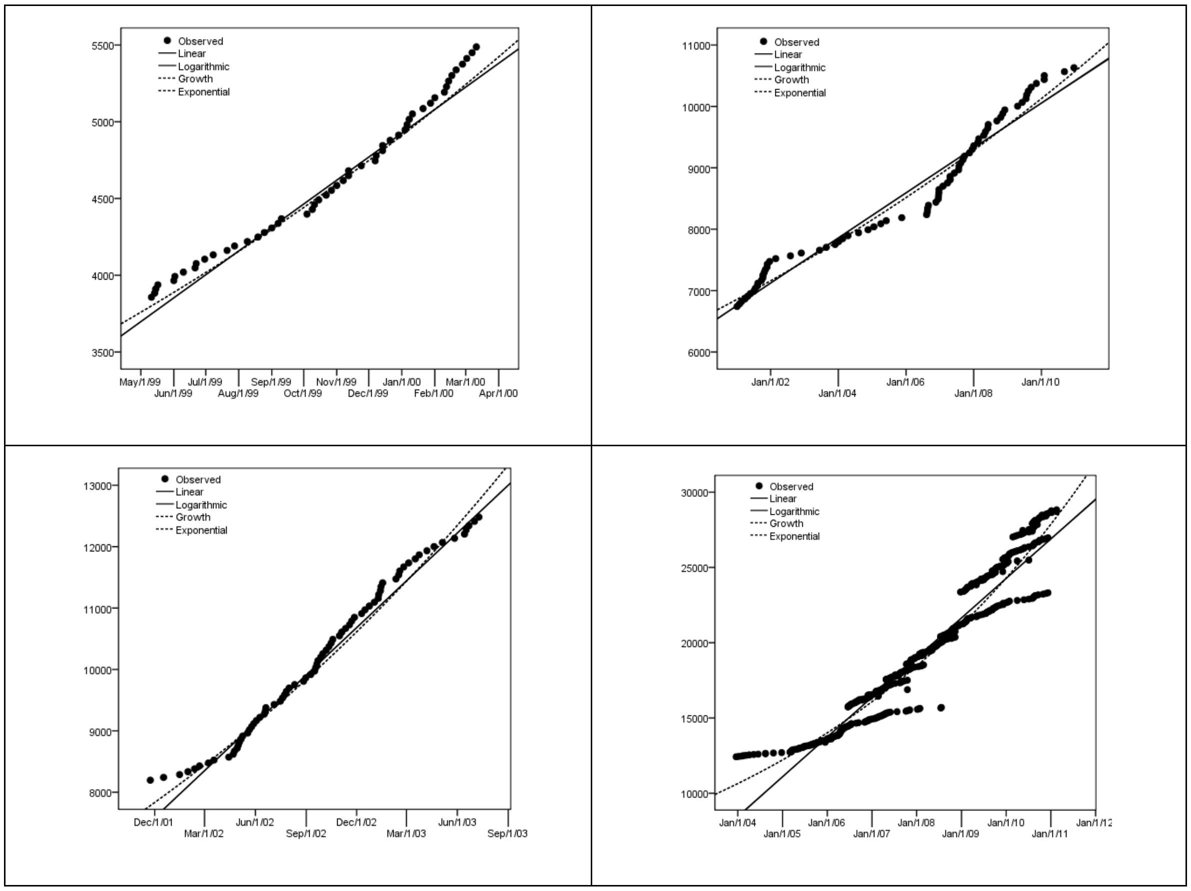


Figure 6.1. The growth of the number of files in Linux-kernel over 10 versions with a total of 974 releases.

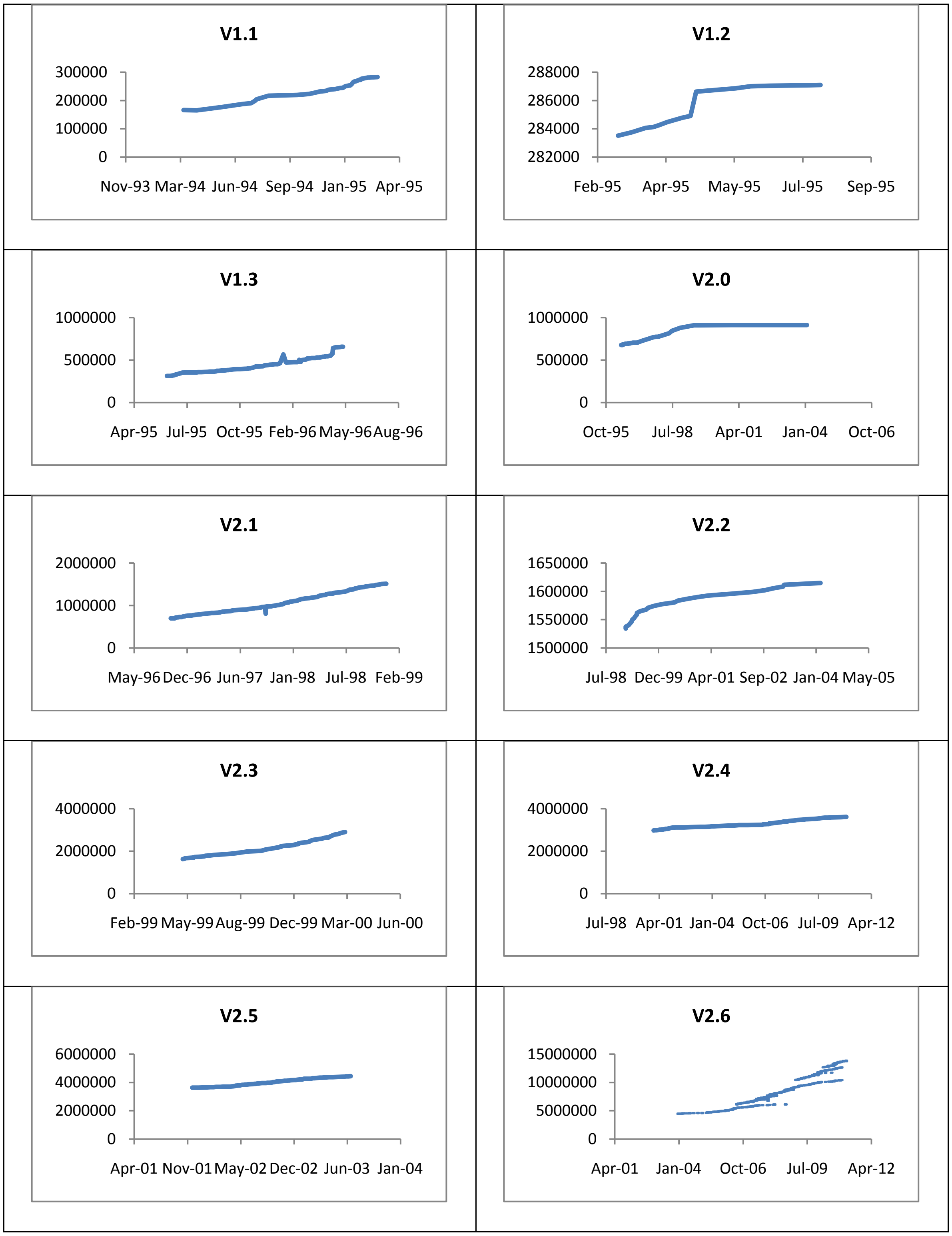


Figure 6.2. LOC growth in Linux, with a total 974 versions.

Table 6.3. Linux kernel versions, Percentage is the slice size relative to program size measured in LOC, versions are sorted and ordered by their release date.

Version	Date	LOC	Slice Size	Percentage	Lag-Time
1.0.0	3/13/1994	166,144	83,891	50.5%	
1.1.0	4/6/1994	165,768	83,696	50.5%	24
1.1.13	5/23/1994	177,630	89,046	50.1%	47
1.1.23	6/27/1994	186,785	93,261	49.9%	35
1.1.29	7/14/1994	190,831	94,494	49.5%	17
1.1.33	7/21/1994	198,877	98,034	49.3%	7
1.1.35	7/24/1994	204,162	100,703	49.3%	3
1.1.45	8/15/1994	216,889	104,759	48.3%	22
1.1.52	10/6/1994	220,179	106,573	48.4%	52
1.1.59	10/28/1994	223,165	108,229	48.5%	22
1.1.63	11/14/1994	230,310	111,582	48.4%	17
1.1.64	11/15/1994	231,266	112,155	48.5%	1
1.1.67	11/28/1994	234,373	113,157	48.3%	13
1.1.70	12/2/1994	237,793	115,075	48.4%	4
1.1.71	12/5/1994	238,844	115,417	48.3%	3
1.1.73	12/15/1994	240,370	115,894	48.2%	10
1.1.74	12/23/1994	244,042	117,721	48.2%	8
1.1.75	12/29/1994	244,716	118,076	48.3%	6
1.1.76	1/2/1995	250,188	120,986	48.4%	4
1.1.78	1/9/1995	252,895	121,686	48.1%	7
1.1.79	1/11/1995	253,326	122,058	48.2%	2
1.1.80	1/12/1995	255,196	122,753	48.1%	1
1.1.81	1/13/1995	257,051	123,620	48.1%	1
1.1.82	1/16/1995	263,826	125,801	47.7%	3
1.1.83	1/18/1995	266,825	126,785	47.5%	2
1.1.84	1/22/1995	268,217	127,627	47.6%	4
1.1.85	1/23/1995	269,823	128,017	47.4%	1
1.1.86	1/27/1995	273,136	130,021	47.6%	4
1.1.87	1/30/1995	273,503	130,185	47.6%	3
1.1.88	1/31/1995	276,934	131,533	47.5%	1
1.1.89	2/5/1995	278,956	132,533	47.5%	5
1.1.90	2/8/1995	279,088	132,626	47.5%	3
1.1.91	2/12/1995	281,075	133,315	47.4%	4
1.1.92	2/15/1995	281,711	133,877	47.5%	3
1.1.93	2/20/1995	282,078	133,906	47.5%	5

1.1.94	2/22/1995	282,540	134,125	47.5%	2
1.1.95	3/2/1995	283,492	134,458	47.4%	8
1.2.0	3/7/1995	283,522	134,446	47.4%	5
1.2.1	3/17/1995	283,757	134,686	47.5%	10
1.2.2	3/27/1995	284,061	134,802	47.5%	10
1.2.3	4/2/1995	284,125	134,832	47.5%	6
1.2.4	4/6/1995	284,256	134,885	47.5%	4
1.2.5	4/12/1995	284,477	134,930	47.4%	6
1.2.6	4/23/1995	284,788	135,113	47.4%	11
1.2.7	4/29/1995	284,919	135,181	47.4%	6
1.2.8	5/3/1995	286,632	136,034	47.5%	4
1.2.9	6/1/1995	286,873	136,122	47.5%	29
1.2.10	6/12/1995	287,011	136,000	47.4%	11
1.3.0	6/12/1995	312,214	150,199	48.1%	0
1.3.2	6/16/1995	313,630	150,986	48.1%	4
1.3.3	6/18/1995	313,936	151,263	48.2%	2
1.2.11	6/26/1995	287,045	136,206	47.5%	8
1.3.4	6/26/1995	323,249	154,492	47.8%	0
1.3.5	6/29/1995	330,525	158,224	47.9%	3
1.3.6	6/30/1995	330,909	158,340	47.9%	1
1.3.7	7/6/1995	344,697	164,527	47.7%	6
1.3.8	7/7/1995	344,761	164,544	47.7%	1
1.3.9	7/11/1995	352,533	167,486	47.5%	4
1.3.10	7/13/1995	353,360	168,133	47.6%	2
1.3.11	7/18/1995	353,953	168,472	47.6%	5
1.2.12	7/25/1995	287,080	136,272	47.5%	7
1.3.12	7/25/1995	354,444	168,536	47.5%	0
1.3.13	7/27/1995	354,586	168,658	47.6%	2
1.3.14	7/31/1995	355,287	169,184	47.6%	4
1.2.13	8/2/1995	287,104	136,293	47.5%	2
1.3.15	8/2/1995	355,887	169,317	47.6%	0
1.3.16	8/8/1995	356,417	169,914	47.7%	6
1.3.17	8/9/1995	357,133	170,024	47.6%	1
1.3.18	8/13/1995	357,417	170,142	47.6%	4
1.3.19	8/15/1995	358,326	170,524	47.6%	2
1.3.20	8/16/1995	358,865	170,974	47.6%	1
1.3.21	8/28/1995	360,355	171,801	47.7%	12
1.3.22	9/1/1995	363,404	173,025	47.6%	4

1.3.23	9/3/1995	364,036	173,587	47.7%	2
1.3.24	9/5/1995	364,154	173,750	47.7%	2
1.3.25	9/9/1995	364,897	174,250	47.8%	4
1.3.26	9/13/1995	366,198	174,547	47.7%	4
1.3.27	9/14/1995	374,100	178,717	47.8%	1
1.3.28	9/18/1995	374,978	178,782	47.7%	4
2.3.29	9/23/1995	375,044	178,825	47.7%	5
1.3.30	9/27/1995	377,913	180,485	47.8%	4
1.3.31	10/4/1995	381,183	181,956	47.7%	7
1.3.32	10/6/1995	381,797	182,317	47.8%	2
1.3.33	10/10/1995	384,929	183,431	47.7%	4
1.3.34	10/13/1995	389,100	185,546	47.7%	3
1.3.35	10/16/1995	391,388	186,233	47.6%	3
1.3.36	10/23/1995	394,078	187,589	47.6%	7
1.3.37	10/28/1995	395,168	188,068	47.6%	5
1.3.38	11/7/1995	396,340	187,693	47.4%	10
1.3.39	11/9/1995	397,245	187,951	47.3%	2
1.3.40	11/11/1995	397,807	188,694	47.4%	2
1.3.41	11/13/1995	402,833	188,973	46.9%	2
1.3.42	11/16/1995	403,687	191,688	47.5%	3
1.3.43	11/21/1995	409,388	192,333	47.0%	5
1.3.44	11/25/1995	418,189	194,468	46.5%	4
1.3.45	11/27/1995	425,205	197,128	46.4%	2
1.3.46	12/11/1995	426,991	200,330	46.9%	14
1.3.47	12/13/1995	436,120	201,326	46.2%	2
1.3.48	12/17/1995	439,514	203,464	46.3%	4
1.3.49	12/21/1995	442,048	205,036	46.4%	4
1.3.50	12/24/1995	444,574	206,683	46.5%	3
1.3.51	12/27/1995	444,789	207,942	46.8%	3
1.3.52	12/29/1995	449,132	207,987	46.3%	2
1.3.53	1/2/1996	450,467	210,424	46.7%	4
1.3.54	1/4/1996	450,932	210,965	46.8%	2
1.3.55	1/6/1996	451,016	211,135	46.8%	2
1.3.56	1/8/1996	452,968	211,184	46.6%	2
1.3.57	1/12/1996	463,795	212,100	45.7%	4
1.3.58	1/18/1996	567,594	216,963	38.2%	6
1.3.59	1/23/1996	474,390	219,104	46.2%	5
1.3.60	2/7/1996	475,490	224,003	47.1%	15

1.3.61	2/9/1996	475,754	224,292	47.1%	2
1.3.62	2/11/1996	476,041	224,446	47.1%	2
1.3.63	2/14/1996	477,096	224,488	47.1%	3
1.3.64	2/15/1996	477,664	224,892	47.1%	1
1.3.65	2/17/1996	477,766	225,055	47.1%	2
1.3.66	2/17/1996	477,758	225,125	47.1%	0
1.3.69	2/17/1996	502,991	240,456	47.8%	0
1.3.67	2/20/1996	479,248	225,115	47.0%	3
1.3.68	2/22/1996	501,394	225,956	45.1%	2
1.3.70	3/1/1996	505,855	241,344	47.7%	8
1.3.71	3/4/1996	520,232	242,860	46.7%	3
1.3.72	3/8/1996	521,848	248,521	47.6%	4
1.3.73	3/12/1996	523,217	249,799	47.7%	4
1.3.74	3/14/1996	525,205	250,422	47.7%	2
1.3.75	3/16/1996	525,363	251,406	47.9%	2
1.3.76	3/19/1996	525,639	251,438	47.8%	3
1.3.77	3/21/1996	529,728	251,620	47.5%	2
1.3.78	3/25/1996	529,751	253,696	47.9%	4
1.3.79	3/26/1996	529,807	253,682	47.9%	1
1.3.80	3/28/1996	531,749	253,643	47.7%	2
1.3.81	3/30/1996	535,370	254,513	47.5%	2
1.3.82	4/2/1996	538,475	256,406	47.6%	3
1.3.83	4/3/1996	538,610	258,022	47.9%	1
1.3.84	4/4/1996	540,187	258,059	47.8%	1
1.3.85	4/8/1996	544,966	258,865	47.5%	4
1.3.86	4/10/1996	545,493	260,928	47.8%	2
1.3.87	4/12/1996	545,478	261,248	47.9%	2
1.3.88	4/13/1996	547,341	261,218	47.7%	1
1.3.89	4/15/1996	547,516	262,226	47.9%	2
1.3.90	4/16/1996	548,269	262,187	47.8%	1
1.3.91	4/18/1996	562,143	262,600	46.7%	2
1.3.92	4/20/1996	574,310	264,260	46.0%	2
1.3.93	4/21/1996	641,932	269,562	42.0%	1
1.3.94	4/22/1996	641,888	297,048	46.3%	1
1.3.95	4/24/1996	648,611	297,062	45.8%	2
1.3.96	4/27/1996	650,702	300,291	46.1%	3
1.3.97	4/29/1996	651,848	301,289	46.2%	2
1.3.98	5/4/1996	655,378	302,276	46.1%	5

1.3.99	5/7/1996	655,774	305,094	46.5%	3
1.3.100	5/10/1996	655,783	305,404	46.6%	3
2.0.0	6/9/1996	677,958	312,861	46.1%	30
2.0.1	7/3/1996	679,132	315,020	46.4%	24
2.0.2	7/5/1996	679,280	315,116	46.4%	2
2.0.3	7/6/1996	688,391	317,940	46.2%	1
2.0.4	7/8/1996	687,441	317,908	46.2%	2
2.0.5	7/10/1996	687,681	318,005	46.2%	2
2.0.6	7/12/1996	687,962	318,376	46.3%	2
2.0.7	7/15/1996	688,083	318,397	46.3%	3
2.0.8	7/25/1996	688,408	318,512	46.3%	10
2.0.9	7/26/1996	688,724	318,639	46.3%	1
2.0.10	7/27/1996	688,704	318,628	46.3%	1
2.0.11	8/5/1996	689,882	318,941	46.2%	9
2.0.12	8/9/1996	689,891	318,890	46.2%	4
2.0.13	8/16/1996	693,000	321,607	46.4%	7
2.0.14	8/20/1996	693,523	322,017	46.4%	4
2.0.15	8/25/1996	693,651	322,075	46.4%	5
2.0.16	8/31/1996	693,971	322,184	46.4%	6
2.0.17	9/2/1996	694,179	322,278	46.4%	2
2.0.18	9/5/1996	694,240	322,312	46.4%	3
2.0.19	9/11/1996	694,461	322,431	46.4%	6
2.0.20	9/13/1996	694,499	322,448	46.4%	2
2.0.21	9/20/1996	694,965	322,705	46.4%	7
2.1.0	9/30/1996	698,027	324,136	46.4%	10
2.1.1	10/3/1996	698,288	324,255	46.4%	3
2.0.22	10/8/1996	695,662	323,052	46.4%	5
2.1.2	10/8/1996	699,010	324,583	46.4%	0
2.1.3	10/10/1996	699,735	324,913	46.4%	2
2.1.4	10/15/1996	700,804	325,399	46.4%	5
2.0.23	10/18/1996	695,866	323,136	46.4%	3
2.1.5	10/18/1996	718,955	333,653	46.4%	0
2.1.6	10/29/1996	723,214	335,590	46.4%	11
2.0.24	10/30/1996	702,315	324,685	46.2%	1
2.1.7	11/1/1996	727,498	337,538	46.4%	2
2.0.25	11/8/1996	702,402	324,771	46.2%	7
2.1.8	11/9/1996	731,806	339,497	46.4%	1
2.1.9	11/12/1996	736,140	341,468	46.4%	3

2.1.10	11/15/1996	740,498	343,450	46.4%	3
2.1.11	11/18/1996	744,881	345,443	46.4%	3
2.0.26	11/22/1996	703,495	325,392	46.3%	4
2.1.12	11/22/1996	749,290	347,448	46.4%	0
2.1.13	11/23/1996	753,725	349,465	46.4%	1
2.0.27	12/1/1996	703,885	325,545	46.2%	8
2.1.14	12/1/1996	758,187	351,494	46.4%	0
2.1.15	12/12/1996	762,673	353,534	46.4%	11
2.1.16	12/18/1996	767,185	355,586	46.3%	6
2.1.17	12/22/1996	771,724	357,650	46.3%	4
2.1.18	12/29/1996	776,289	359,726	46.3%	7
2.1.19	12/31/1996	780,881	361,814	46.3%	2
2.1.20	1/2/1997	785,499	363,914	46.3%	2
2.0.28	1/14/1997	704,209	325,483	46.2%	12
2.1.21	1/14/1997	790,143	366,026	46.3%	0
2.1.22	1/23/1997	794,816	368,151	46.3%	9
2.1.23	1/26/1997	799,515	370,288	46.3%	3
2.1.24	1/28/1997	804,241	372,437	46.3%	2
2.0.29	2/7/1997	704,276	325,503	46.2%	10
2.1.27	2/26/1997	818,587	378,961	46.3%	19
2.1.28	3/3/1997	823,423	381,160	46.3%	5
2.1.29	3/10/1997	828,289	383,373	46.3%	7
2.1.30	3/26/1997	833,182	385,598	46.3%	16
2.1.31	4/3/1997	838,103	387,836	46.3%	8
2.1.32	4/5/1997	843,053	390,087	46.3%	2
2.0.30	4/8/1997	724,756	335,543	46.3%	3
2.1.33	4/10/1997	848,034	392,352	46.3%	2
2.1.34	4/14/1997	853,041	394,629	46.3%	4
2.1.35	4/15/1997	858,079	396,920	46.3%	1
2.1.36	4/23/1997	863,145	399,224	46.3%	8
2.1.37	5/14/1997	868,240	401,541	46.2%	21
2.1.38	5/18/1997	873,366	403,872	46.2%	4
2.1.39	5/18/1997	878,521	406,216	46.2%	0
2.1.40	5/22/1997	883,706	408,574	46.2%	4
2.1.41	5/28/1997	888,920	410,945	46.2%	6
2.1.42	5/29/1997	894,167	413,331	46.2%	1
2.1.43	6/16/1997	899,442	415,730	46.2%	18
2.1.44	7/7/1997	904,748	418,143	46.2%	21

2.1.45	7/17/1997	910,085	420,570	46.2%	10
2.1.46	7/19/1997	915,453	423,011	46.2%	2
2.1.47	7/24/1997	920,854	425,467	46.2%	5
2.1.48	8/4/1997	926,283	427,936	46.2%	11
2.1.49	8/11/1997	931,745	430,420	46.2%	7
2.1.50	8/14/1997	937,241	432,919	46.2%	3
2.1.51	8/19/1997	942,767	435,432	46.2%	5
2.1.52	9/3/1997	948,324	437,959	46.2%	15
2.1.53	9/4/1997	953,913	440,501	46.2%	1
2.1.54	9/6/1997	959,536	443,058	46.2%	2
2.1.55	9/9/1997	965,192	445,630	46.2%	3
2.1.56	9/20/1997	970,879	448,216	46.2%	11
2.1.25	9/23/1997	808,995	374,599	46.3%	3
2.1.26	9/23/1997	813,778	376,774	46.3%	0
2.1.57	9/25/1997	976,600	450,818	46.2%	2
2.1.58	10/15/1997	982,355	453,435	46.2%	20
2.0.31	10/17/1997	772,428	360,444	46.7%	2
2.1.59	10/17/1997	988,143	456,067	46.2%	0
2.1.60	10/25/1997	993,964	458,714	46.1%	8
2.1.61	10/31/1997	999,820	461,377	46.1%	6
2.1.62	11/3/1997	1,005,709	464,055	46.1%	3
2.1.63	11/12/1997	1,011,631	466,748	46.1%	9
2.1.64	11/15/1997	1,017,590	469,458	46.1%	3
2.0.32	11/18/1997	773,460	361,024	46.7%	3
2.1.65	11/18/1997	1,023,582	472,183	46.1%	0
2.1.66	11/26/1997	1,029,607	474,923	46.1%	8
2.1.67	11/29/1997	1,035,670	477,680	46.1%	3
2.1.68	11/30/1997	1,041,768	480,453	46.1%	1
2.1.69	12/1/1997	1,047,901	483,242	46.1%	1
2.1.70	12/3/1997	1,054,069	486,047	46.1%	2
2.1.71	12/4/1997	1,060,272	488,868	46.1%	1
2.1.72	12/9/1997	1,066,513	491,706	46.1%	5
2.0.33	12/16/1997	774,036	361,641	46.7%	7
2.1.73	12/19/1997	1,072,789	494,560	46.1%	3
2.1.74	12/20/1997	1,079,100	497,430	46.1%	1
2.1.75	12/22/1997	1,085,451	500,318	46.1%	2
2.1.76	12/24/1997	1,091,837	503,222	46.1%	2
2.1.77	1/2/1998	1,098,260	506,143	46.1%	9

2.1.78	1/6/1998	1,104,721	509,081	46.1%	4
2.1.79	1/13/1998	1,111,219	512,036	46.1%	7
2.1.80	1/21/1998	1,117,754	515,008	46.1%	8
2.1.81	1/24/1998	1,124,327	517,997	46.1%	3
2.1.82	1/26/1998	1,130,939	521,004	46.1%	2
2.1.83	1/30/1998	1,137,589	524,028	46.1%	4
2.1.84	1/31/1998	1,144,279	527,070	46.1%	1
2.1.85	2/4/1998	1,151,005	530,129	46.1%	4
2.1.86	2/11/1998	1,157,772	533,206	46.1%	7
2.1.87	2/17/1998	1,164,578	536,301	46.1%	6
2.1.88	2/21/1998	1,171,423	539,414	46.0%	4
2.1.89	3/7/1998	1,178,308	542,545	46.0%	14
2.1.90	3/18/1998	1,185,235	545,695	46.0%	11
2.1.91	3/26/1998	1,192,199	548,862	46.0%	8
2.1.92	4/2/1998	1,199,205	552,048	46.0%	7
2.1.93	4/7/1998	1,206,251	555,252	46.0%	5
2.1.94	4/9/1998	1,213,338	558,475	46.0%	2
2.1.95	4/10/1998	1,220,467	561,717	46.0%	1
2.1.96	4/14/1998	1,227,636	564,977	46.0%	4
2.1.97	4/18/1998	1,234,849	568,257	46.0%	4
2.1.98	4/24/1998	1,242,101	571,555	46.0%	6
2.1.99	5/1/1998	1,249,397	574,873	46.0%	7
2.1.100	5/8/1998	1,256,735	578,210	46.0%	7
2.1.101	5/9/1998	1,264,115	581,566	46.0%	1
2.1.102	5/14/1998	1,271,539	584,942	46.0%	5
2.1.103	5/21/1998	1,279,005	588,337	46.0%	7
2.0.34	6/4/1998	816,287	374,819	45.9%	14
2.1.104	6/5/1998	1,286,514	591,752	46.0%	1
2.1.105	6/7/1998	1,294,068	595,187	46.0%	2
2.1.106	6/13/1998	1,301,665	598,642	46.0%	6
2.1.107	6/25/1998	1,309,305	602,116	46.0%	12
2.1.108	7/2/1998	1,316,990	605,611	46.0%	7
2.0.35	7/13/1998	844,848	391,762	46.4%	11
2.1.109	7/17/1998	1,324,722	609,127	46.0%	4
2.1.110	7/21/1998	1,332,495	612,662	46.0%	4
2.1.111	7/25/1998	1,340,317	616,219	46.0%	4
2.1.112	7/28/1998	1,348,181	619,795	46.0%	3
2.1.113	8/1/1998	1,356,093	623,393	46.0%	4

2.1.114	8/3/1998	1,364,051	627,012	46.0%	2
2.1.115	8/6/1998	1,372,053	630,651	46.0%	3
2.1.116	8/19/1998	1,380,104	634,312	46.0%	13
2.1.117	8/20/1998	1,388,200	637,994	46.0%	1
2.1.118	8/26/1998	1,396,343	641,697	46.0%	6
2.1.119	8/27/1998	1,404,535	645,422	46.0%	1
2.1.120	9/5/1998	1,412,772	649,168	45.9%	9
2.1.121	9/9/1998	1,421,058	652,936	45.9%	4
2.1.122	9/16/1998	1,429,392	656,726	45.9%	7
2.1.123	9/28/1998	1,437,775	660,538	45.9%	12
2.1.124	10/4/1998	1,446,206	664,372	45.9%	6
2.1.125	10/9/1998	1,454,685	668,228	45.9%	5
2.1.126	10/24/1998	1,463,215	672,107	45.9%	15
2.1.127	11/7/1998	1,471,793	676,008	45.9%	14
2.1.128	11/12/1998	1,480,422	679,932	45.9%	5
2.0.36	11/16/1998	879,296	406,679	46.3%	4
2.1.129	11/19/1998	1,489,102	683,879	45.9%	3
2.1.130	11/26/1998	1,497,832	687,849	45.9%	7
2.1.131	12/3/1998	1,506,610	691,841	45.9%	7
2.1.132	12/22/1998	1,515,441	695,857	45.9%	19
2.2.0	1/26/1999	1,534,289	704,428	45.9%	35
2.2.1	1/28/1999	1,537,319	705,806	45.9%	2
2.2.2	2/23/1999	1,540,356	707,187	45.9%	26
2.2.3	3/9/1999	1,543,399	708,571	45.9%	14
2.2.4	3/23/1999	1,546,449	709,958	45.9%	14
2.2.5	3/29/1999	1,549,504	711,347	45.9%	6
2.2.6	4/16/1999	1,552,565	712,739	45.9%	18
2.2.7	4/28/1999	1,555,632	714,134	45.9%	12
2.2.8	5/11/1999	1,558,704	715,531	45.9%	13
2.3.0	5/11/1999	1,621,440	744,060	45.9%	0
2.2.9	5/13/1999	1,561,783	716,931	45.9%	2
2.3.1	5/14/1999	1,640,195	752,589	45.9%	1
2.3.2	5/15/1999	1,659,163	761,215	45.9%	1
2.3.3	5/17/1999	1,678,350	769,940	45.9%	2
2.3.4	6/1/1999	1,697,758	778,766	45.9%	15
2.3.5	6/2/1999	1,717,386	787,692	45.9%	1
2.3.6	6/10/1999	1,737,241	796,721	45.9%	8
2.0.37	6/14/1999	909,460	420,679	46.3%	4

2.2.10	6/14/1999	1,564,868	718,334	45.9%	0
2.3.7	6/21/1999	1,757,322	805,853	45.9%	7
2.3.8	6/22/1999	1,777,635	815,090	45.9%	1
2.3.9	6/30/1999	1,798,180	824,433	45.8%	8
2.3.10	7/8/1999	1,818,960	833,883	45.8%	8
2.3.11	7/21/1999	1,839,978	843,441	45.8%	13
2.3.12	7/28/1999	1,861,238	853,109	45.8%	7
2.2.11	8/9/1999	1,567,960	719,740	45.9%	12
2.3.13	8/9/1999	1,882,740	862,887	45.8%	0
2.3.14	8/19/1999	1,904,491	872,778	45.8%	10
2.0.38	8/25/1999	909,473	420,684	46.3%	6
2.3.15	8/25/1999	1,926,489	882,782	45.8%	0
2.2.12	8/26/1999	1,571,056	721,148	45.9%	1
2.3.16	9/1/1999	1,948,739	892,900	45.8%	6
2.3.17	9/7/1999	1,971,246	903,135	45.8%	6
2.3.18	9/10/1999	1,994,010	913,487	45.8%	3
2.3.19	10/4/1999	2,017,035	923,958	45.8%	24
2.3.20	10/9/1999	2,040,323	934,548	45.8%	5
2.3.21	10/11/1999	2,063,881	945,261	45.8%	2
2.3.22	10/15/1999	2,087,705	956,095	45.8%	4
2.2.13	10/20/1999	1,574,159	722,559	45.9%	5
2.3.23	10/22/1999	2,111,803	967,054	45.8%	2
2.3.24	10/27/1999	2,136,179	978,139	45.8%	5
2.3.25	11/1/1999	2,160,835	989,351	45.8%	5
2.3.26	11/7/1999	2,185,771	1,000,691	45.8%	6
2.3.27	11/12/1999	2,210,994	1,012,161	45.8%	5
2.3.28	11/12/1999	2,236,507	1,023,763	45.8%	0
2.3.29	11/24/1999	2,262,312	1,035,498	45.8%	12
2.3.30	12/7/1999	2,288,412	1,047,367	45.8%	13
2.3.31	12/8/1999	2,314,811	1,059,372	45.8%	1
2.3.32	12/14/1999	2,341,513	1,071,515	45.8%	6
2.3.33	12/14/1999	2,368,521	1,083,797	45.8%	0
2.3.34	12/21/1999	2,395,839	1,096,220	45.8%	7
2.3.35	12/29/1999	2,423,470	1,108,785	45.8%	8
2.2.14	1/4/2000	1,577,268	723,973	45.9%	6
2.3.36	1/4/2000	2,451,417	1,121,494	45.7%	0
2.3.37	1/6/2000	2,479,685	1,134,349	45.7%	2
2.3.38	1/8/2000	2,508,277	1,147,351	45.7%	2

2.3.39	1/11/2000	2,537,196	1,160,502	45.7%	3
2.3.40	1/21/2000	2,566,447	1,173,804	45.7%	10
2.3.41	1/28/2000	2,596,034	1,187,259	45.7%	7
2.3.42	2/1/2000	2,625,958	1,200,867	45.7%	4
2.3.43	2/10/2000	2,656,227	1,214,632	45.7%	9
2.3.44	2/12/2000	2,686,844	1,228,555	45.7%	2
2.3.45	2/14/2000	2,717,810	1,242,637	45.7%	2
2.3.46	2/17/2000	2,749,131	1,256,880	45.7%	3
2.3.47	2/21/2000	2,780,812	1,271,287	45.7%	4
2.3.48	2/27/2000	2,812,856	1,285,859	45.7%	6
2.3.49	3/2/2000	2,845,267	1,300,598	45.7%	4
2.3.50	3/7/2000	2,878,047	1,315,505	45.7%	5
2.3.51	3/11/2000	2,911,206	1,330,584	45.7%	4
2.2.15	5/4/2000	1,580,382	725,389	45.9%	54
2.2.16	6/7/2000	1,583,505	726,809	45.9%	34
2.2.17	9/4/2000	1,586,632	728,231	45.9%	89
2.2.18	12/11/2000	1,589,765	729,656	45.9%	98
2.4.0	1/4/2001	2,978,667	1,361,262	45.7%	24
2.0.39	1/9/2001	912,577	422,335	46.3%	5
2.4.1	1/30/2001	2,986,363	1,364,762	45.7%	21
2.4.2	2/22/2001	2,994,080	1,368,271	45.7%	23
2.2.19	3/25/2001	1,592,905	731,084	45.9%	31
2.4.3	3/30/2001	3,001,816	1,371,789	45.7%	5
2.4.4	4/28/2001	3,009,569	1,375,315	45.7%	29
2.4.5	5/26/2001	3,017,345	1,378,851	45.7%	28
2.4.6	7/4/2001	3,025,140	1,382,396	45.7%	39
2.4.7	7/20/2001	3,032,958	1,385,951	45.7%	16
2.4.8	8/11/2001	3,040,793	1,389,514	45.7%	22
2.4.9	8/16/2001	3,048,648	1,393,086	45.7%	5
2.4.10	9/23/2001	3,056,525	1,396,668	45.7%	38
2.4.11	10/9/2001	3,064,421	1,400,259	45.7%	16
2.4.12	10/11/2001	3,072,338	1,403,859	45.7%	2
2.4.13	10/24/2001	3,080,274	1,407,468	45.7%	13
2.2.20	11/2/2001	1,596,050	732,514	45.9%	9
2.4.14	11/5/2001	3,088,232	1,411,087	45.7%	3
2.4.15	11/23/2001	3,096,210	1,414,715	45.7%	18
2.5.0	11/23/2001	3,633,072	1,658,854	45.7%	0
2.4.16	11/26/2001	3,104,208	1,418,352	45.7%	3

2.5.1	12/17/2001	3,642,939	1,663,341	45.7%	21
2.4.17	12/21/2001	3,112,227	1,421,999	45.7%	4
2.5.2	1/15/2002	3,652,834	1,667,841	45.7%	25
2.5.3	1/30/2002	3,662,754	1,672,352	45.7%	15
2.5.4	2/11/2002	3,672,702	1,676,876	45.7%	12
2.5.5	2/20/2002	3,682,677	1,681,412	45.7%	9
2.4.18	2/25/2002	3,120,267	1,425,655	45.7%	5
2.5.6	3/8/2002	3,692,680	1,685,961	45.7%	11
2.5.7	3/18/2002	3,702,707	1,690,521	45.7%	10
2.5.8	4/14/2002	3,712,763	1,695,094	45.7%	27
2.5.9	4/22/2002	3,722,848	1,699,680	45.7%	8
2.5.10	4/24/2002	3,732,959	1,704,278	45.7%	2
2.5.11	4/29/2002	3,743,096	1,708,888	45.7%	5
2.5.12	5/1/2002	3,753,262	1,713,511	45.7%	2
2.5.13	5/3/2002	3,763,455	1,718,146	45.7%	2
2.5.14	5/6/2002	3,773,673	1,722,793	45.7%	3
2.5.15	5/9/2002	3,783,923	1,727,454	45.7%	3
2.5.16	5/18/2002	3,794,199	1,732,127	45.7%	9
2.2.21	5/20/2002	1,599,203	733,948	45.9%	2
2.5.17	5/21/2002	3,804,501	1,736,812	45.7%	1
2.5.18	5/25/2002	3,814,832	1,741,510	45.7%	4
2.5.19	5/29/2002	3,825,192	1,746,221	45.7%	4
2.5.20	6/3/2002	3,835,580	1,750,945	45.7%	5
2.5.21	6/9/2002	3,845,994	1,755,681	45.6%	6
2.5.22	6/17/2002	3,856,439	1,760,431	45.6%	8
2.5.23	6/19/2002	3,866,911	1,765,193	45.6%	2
2.5.24	6/20/2002	3,877,411	1,769,968	45.6%	1
2.5.25	7/5/2002	3,887,940	1,774,756	45.6%	15
2.5.26	7/16/2002	3,898,498	1,779,557	45.6%	11
2.5.27	7/20/2002	3,909,081	1,784,370	45.6%	4
2.5.28	7/24/2002	3,919,696	1,789,197	45.6%	4
2.5.29	7/27/2002	3,930,339	1,794,037	45.6%	3
2.5.30	8/1/2002	3,941,011	1,798,890	45.6%	5
2.4.19	8/3/2002	3,128,326	1,429,320	45.7%	2
2.5.31	8/11/2002	3,951,711	1,803,756	45.6%	8
2.5.32	8/27/2002	3,962,442	1,808,636	45.6%	16
2.5.33	8/31/2002	3,973,200	1,813,528	45.6%	4
2.5.34	9/9/2002	3,983,988	1,818,434	45.6%	9

2.2.22	9/16/2002	1,602,361	735,384	45.9%	7
2.5.35	9/16/2002	3,994,805	1,823,353	45.6%	0
2.5.36	9/18/2002	4,005,650	1,828,285	45.6%	2
2.5.37	9/20/2002	4,016,527	1,833,231	45.6%	2
2.5.38	9/22/2002	4,027,431	1,838,190	45.6%	2
2.5.39	9/27/2002	4,038,367	1,843,163	45.6%	5
2.5.40	10/1/2002	4,049,329	1,848,148	45.6%	4
2.5.41	10/7/2002	4,060,324	1,853,148	45.6%	6
2.5.42	10/12/2002	4,071,348	1,858,161	45.6%	5
2.5.43	10/16/2002	4,082,400	1,863,187	45.6%	4
2.5.44	10/19/2002	4,093,483	1,868,227	45.6%	3
2.5.45	10/31/2002	4,104,597	1,873,281	45.6%	12
2.5.46	11/4/2002	4,115,739	1,878,348	45.6%	4
2.5.47	11/11/2002	4,126,914	1,883,430	45.6%	7
2.5.48	11/18/2002	4,138,116	1,888,524	45.6%	7
2.5.49	11/22/2002	4,149,351	1,893,633	45.6%	4
2.5.50	11/27/2002	4,160,614	1,898,755	45.6%	5
2.4.20	11/28/2002	3,136,408	1,432,995	45.7%	1
2.2.23	11/29/2002	1,605,525	736,823	45.9%	1
2.5.51	12/10/2002	4,171,910	1,903,892	45.6%	11
2.5.52	12/16/2002	4,183,235	1,909,042	45.6%	6
2.5.53	12/24/2002	4,194,591	1,914,206	45.6%	8
2.5.54	1/2/2003	4,205,977	1,919,384	45.6%	9
2.5.55	1/9/2003	4,217,394	1,924,576	45.6%	7
2.5.56	1/10/2003	4,228,842	1,929,782	45.6%	1
2.5.57	1/13/2003	4,240,323	1,935,003	45.6%	3
2.5.58	1/14/2003	4,251,833	1,940,237	45.6%	1
2.5.59	1/17/2003	4,263,373	1,945,485	45.6%	3
2.5.60	2/10/2003	4,274,947	1,950,748	45.6%	24
2.5.61	2/15/2003	4,286,551	1,956,025	45.6%	5
2.5.62	2/17/2003	4,298,186	1,961,316	45.6%	2
2.5.63	2/24/2003	4,309,853	1,966,622	45.6%	7
2.2.24	3/5/2003	1,608,696	738,265	45.9%	9
2.5.64	3/5/2003	4,321,552	1,971,942	45.6%	0
2.2.25	3/17/2003	1,611,872	739,709	45.9%	12
2.5.65	3/17/2003	4,333,282	1,977,276	45.6%	0
2.5.66	3/24/2003	4,345,044	1,982,625	45.6%	7
2.5.67	4/7/2003	4,356,837	1,987,988	45.6%	14

2.5.68	4/20/2003	4,368,664	1,993,366	45.6%	13
2.5.69	5/5/2003	4,380,521	1,998,758	45.6%	15
2.5.70	5/27/2003	4,392,411	2,004,165	45.6%	22
2.4.21	6/13/2003	3,144,509	1,436,679	45.7%	17
2.5.71	6/14/2003	4,404,331	2,009,586	45.6%	1
2.5.72	6/17/2003	4,416,285	2,015,022	45.6%	3
2.5.73	6/22/2003	4,428,272	2,020,473	45.6%	5
2.5.74	7/2/2003	4,440,289	2,025,938	45.6%	10
2.5.75	7/10/2003	4,452,342	2,031,419	45.6%	8
2.4.22	8/25/2003	3,152,632	1,440,373	45.7%	46
2.4.23	11/28/2003	3,160,775	1,444,076	45.7%	95
2.6.0	12/18/2003	4,476,542	2,042,424	45.6%	20
2.4.24	1/5/2004	3,168,940	1,447,789	45.7%	18
2.6.1	1/9/2004	4,488,692	2,047,949	45.6%	4
2.6.2	2/4/2004	4,500,874	2,053,489	45.6%	26
2.0.40	2/8/2004	913,716	423,324	46.3%	4
2.4.25	2/18/2004	3,177,124	1,451,511	45.7%	10
2.6.3	2/18/2004	4,513,089	2,059,044	45.6%	0
2.2.26	2/24/2004	1,615,056	741,157	45.9%	6
2.6.4	3/11/2004	4,525,338	2,064,614	45.6%	16
2.6.5	4/4/2004	4,537,617	2,070,198	45.6%	24
2.4.26	4/14/2004	3,185,331	1,455,243	45.7%	10
2.6.6	5/10/2004	4,549,934	2,075,799	45.6%	26
2.6.7	6/16/2004	4,562,281	2,081,414	45.6%	37
2.4.27	8/7/2004	3,193,560	1,458,985	45.7%	52
2.6.8	8/14/2004	4,574,661	2,087,044	45.6%	7
2.6.8.1	8/14/2004	4,587,077	2,092,690	45.6%	0
2.6.9	10/18/2004	4,599,526	2,098,351	45.6%	65
2.4.28	11/17/2004	3,201,808	1,462,736	45.7%	30
2.6.10	12/24/2004	4,612,007	2,104,027	45.6%	37
2.4.29	1/19/2005	3,210,079	1,466,497	45.7%	26
2.6.11	3/2/2005	4,624,522	2,109,718	45.6%	42
2.6.11.1	3/9/2005	4,637,071	2,115,425	45.6%	7
2.6.11.2	3/9/2005	4,649,656	2,121,148	45.6%	0
2.6.11.3	3/13/2005	4,662,274	2,126,886	45.6%	4
2.6.11.4	3/16/2005	4,674,925	2,132,639	45.6%	3
2.6.11.5	3/19/2005	4,687,611	2,138,408	45.6%	3
2.6.11.6	3/26/2005	4,700,332	2,144,193	45.6%	7

2.4.30	4/4/2005	3,218,369	1,470,267	45.7%	9
2.6.11.7	4/7/2005	4,713,086	2,149,993	45.6%	3
2.6.11.8	4/30/2005	4,725,876	2,155,809	45.6%	23
2.6.11.9	5/11/2005	4,738,698	2,161,640	45.6%	11
2.6.11.10	5/16/2005	4,751,558	2,167,488	45.6%	5
2.6.11.11	5/27/2005	4,764,451	2,173,351	45.6%	11
2.4.31	6/1/2005	3,226,681	1,474,047	45.7%	5
2.6.11.12	6/12/2005	4,777,378	2,179,230	45.6%	11
2.6.12	6/17/2005	4,790,342	2,185,125	45.6%	5
2.6.12.1	6/22/2005	4,803,340	2,191,036	45.6%	5
2.6.12.2	6/30/2005	4,816,373	2,196,963	45.6%	8
2.6.12.3	7/15/2005	4,829,442	2,202,906	45.6%	15
2.6.12.4	8/5/2005	4,842,546	2,208,865	45.6%	21
2.6.12.5	8/15/2005	4,855,685	2,214,840	45.6%	10
2.6.12.6	8/29/2005	4,868,861	2,220,832	45.6%	14
2.6.13	8/29/2005	4,882,071	2,226,839	45.6%	0
2.6.13.1	9/10/2005	4,895,317	2,232,863	45.6%	12
2.6.13.2	9/18/2005	4,908,599	2,238,903	45.6%	8
2.6.13.3	10/3/2005	4,921,917	2,244,959	45.6%	15
2.6.13.4	10/10/2005	4,935,271	2,251,032	45.6%	7
2.6.14	10/28/2005	4,962,088	2,263,227	45.6%	18
2.6.14.1	11/9/2005	4,975,550	2,269,349	45.6%	12
2.6.14.2	11/11/2005	4,989,050	2,275,488	45.6%	2
2.4.32	11/16/2005	3,235,015	1,477,837	45.7%	5
2.6.14.3	11/24/2005	5,002,587	2,281,644	45.6%	8
2.6.13.5	12/15/2005	4,948,661	2,257,121	45.6%	21
2.6.14.4	12/15/2005	5,016,159	2,287,816	45.6%	0
2.6.14.5	12/27/2005	5,029,766	2,294,004	45.6%	12
2.6.15	1/3/2006	5,070,815	2,312,671	45.6%	7
2.6.14.6	1/8/2006	5,043,413	2,300,210	45.6%	5
2.6.15.1	1/15/2006	5,084,572	2,318,927	45.6%	7
2.6.15.2	1/31/2006	5,098,366	2,325,200	45.6%	16
2.6.14.7	2/2/2006	5,057,096	2,306,432	45.6%	2
2.6.15.3	2/6/2006	5,112,198	2,331,490	45.6%	4
2.6.15.4	2/10/2006	5,126,067	2,337,797	45.6%	4
2.6.15.5	3/1/2006	5,139,974	2,344,121	45.6%	19
2.6.15.6	3/5/2006	5,153,918	2,350,462	45.6%	4
2.6.16	3/20/2006	5,181,917	2,363,195	45.6%	15

2.6.15.7	3/28/2006	5,167,899	2,356,820	45.6%	8
2.6.16.1	3/28/2006	5,195,976	2,369,588	45.6%	0
2.6.16.2	4/7/2006	5,210,071	2,375,998	45.6%	10
2.6.16.3	4/11/2006	5,224,204	2,382,425	45.6%	4
2.6.16.4	4/11/2006	5,238,377	2,388,870	45.6%	0
2.6.16.5	4/12/2006	5,252,587	2,395,332	45.6%	1
2.6.16.6	4/17/2006	5,266,836	2,401,812	45.6%	5
2.6.16.7	4/17/2006	5,281,123	2,408,309	45.6%	0
2.6.16.8	4/18/2006	5,295,447	2,414,823	45.6%	1
2.6.16.9	4/19/2006	5,309,814	2,421,356	45.6%	1
2.6.16.10	4/24/2006	5,324,217	2,427,906	45.6%	5
2.6.16.11	4/24/2006	5,338,658	2,434,473	45.6%	0
2.6.16.12	5/1/2006	5,353,140	2,441,059	45.6%	7
2.6.16.13	5/2/2006	5,367,660	2,447,662	45.6%	1
2.6.16.14	5/5/2006	5,382,220	2,454,283	45.6%	3
2.6.16.15	5/9/2006	5,396,819	2,460,922	45.6%	4
2.6.16.16	5/11/2006	5,411,458	2,467,579	45.6%	2
2.6.16.17	5/20/2006	5,426,136	2,474,254	45.6%	9
2.6.16.18	5/22/2006	5,440,854	2,480,947	45.6%	2
2.6.16.19	5/31/2006	5,455,612	2,487,658	45.6%	9
2.6.16.20	6/5/2006	5,470,411	2,494,388	45.6%	5
2.6.17	6/18/2006	6,146,058	2,801,640	45.6%	13
2.6.16.21	6/20/2006	5,485,248	2,501,135	45.6%	2
2.6.17.1	6/20/2006	6,162,724	2,809,219	45.6%	0
2.6.16.22	6/22/2006	5,500,126	2,507,901	45.6%	2
2.6.16.23	6/30/2006	5,515,044	2,514,685	45.6%	8
2.6.17.2	6/30/2006	6,179,434	2,816,818	45.6%	0
2.6.17.3	6/30/2006	6,196,189	2,824,437	45.6%	0
2.6.16.24	7/6/2006	5,530,004	2,521,488	45.6%	6
2.6.17.4	7/6/2006	6,212,991	2,832,078	45.6%	0
2.6.16.25	7/15/2006	5,545,001	2,528,308	45.6%	9
2.6.16.26	7/15/2006	5,560,042	2,535,148	45.6%	0
2.6.17.5	7/15/2006	6,229,838	2,839,739	45.6%	0
2.6.17.6	7/15/2006	6,246,730	2,847,421	45.6%	0
2.6.16.27	7/17/2006	5,575,123	2,542,006	45.6%	2
2.6.17.7	7/25/2006	6,263,667	2,855,123	45.6%	8
2.6.17.8	8/7/2006	6,280,650	2,862,846	45.6%	13
2.4.33	8/11/2006	3,243,369	1,481,636	45.7%	4

2.6.17.9	8/18/2006	6,297,681	2,870,591	45.6%	7
2.4.33.1	8/19/2006	3,251,747	1,485,446	45.7%	1
2.4.33.2	8/22/2006	3,260,145	1,489,265	45.7%	3
2.6.17.10	8/22/2006	6,314,757	2,878,356	45.6%	0
2.6.17.11	8/23/2006	6,331,878	2,886,142	45.6%	1
2.6.16.28	8/25/2006	5,590,243	2,548,882	45.6%	2
2.4.33.3	8/31/2006	3,268,565	1,493,094	45.7%	6
2.6.17.12	9/8/2006	6,349,046	2,893,949	45.6%	8
2.6.17.13	9/9/2006	6,366,262	2,901,778	45.6%	1
2.6.16.29	9/13/2006	5,605,405	2,555,777	45.6%	4
2.6.18	9/20/2006	6,400,830	2,917,498	45.6%	7
2.6.17.14	10/13/2006	6,383,521	2,909,627	45.6%	23
2.6.18.1	10/14/2006	6,418,184	2,925,390	45.6%	1
2.6.16.30	11/2/2006	5,620,607	2,562,690	45.6%	19
2.6.18.2	11/4/2006	6,435,587	2,933,304	45.6%	2
2.6.16.31	11/7/2006	5,635,853	2,569,623	45.6%	3
2.6.16.32	11/15/2006	5,651,138	2,576,574	45.6%	8
2.4.33.4	11/19/2006	3,277,007	1,496,933	45.7%	4
2.6.18.3	11/19/2006	6,453,034	2,941,238	45.6%	0
2.6.16.33	11/27/2006	5,666,465	2,583,544	45.6%	8
2.6.16.34	11/29/2006	5,681,832	2,590,532	45.6%	2
2.6.18.4	11/29/2006	6,470,531	2,949,195	45.6%	0
2.6.19	11/29/2006	6,558,722	2,989,300	45.6%	0
2.6.18.5	12/2/2006	6,488,075	2,957,173	45.6%	3
2.6.16.35	12/6/2006	5,697,242	2,597,540	45.6%	4
2.6.19.1	12/11/2006	6,576,506	2,997,387	45.6%	5
2.6.16.36	12/13/2006	5,712,692	2,604,566	45.6%	2
2.4.33.5	12/14/2006	3,285,469	1,500,781	45.7%	1
2.6.18.6	12/17/2006	6,505,665	2,965,172	45.6%	3
2.4.33.6	12/18/2006	3,293,955	1,504,640	45.7%	1
2.4.33.7	12/22/2006	3,302,461	1,508,508	45.7%	4
2.4.34	12/23/2006	3,310,991	1,512,387	45.7%	1
2.6.16.37	12/28/2006	5,728,186	2,611,612	45.6%	5
2.6.19.2	1/10/2007	6,594,335	3,005,495	45.6%	13
2.6.16.38	1/20/2007	5,743,722	2,618,677	45.6%	10
2.6.16.39	1/30/2007	5,759,298	2,625,760	45.6%	10
2.4.34.1	2/3/2007	3,319,540	1,516,275	45.7%	4
2.6.20	2/4/2007	6,702,330	3,054,606	45.6%	1

2.6.19.3	2/5/2007	6,612,213	3,013,625	45.6%	1
2.6.16.40	2/10/2007	5,774,917	2,632,863	45.6%	5
2.6.16.41	2/17/2007	5,790,579	2,639,985	45.6%	7
2.6.18.7	2/20/2007	6,523,303	2,973,193	45.6%	3
2.6.19.4	2/20/2007	6,630,139	3,021,777	45.6%	0
2.6.20.1	2/20/2007	6,720,501	3,062,869	45.6%	0
2.6.18.8	2/23/2007	6,540,990	2,981,236	45.6%	3
2.6.19.5	2/24/2007	6,648,114	3,029,951	45.6%	1
2.6.16.42	2/25/2007	5,806,284	2,647,127	45.6%	1
2.6.16.43	3/2/2007	5,822,031	2,654,288	45.6%	5
2.6.19.6	3/3/2007	6,666,137	3,038,147	45.6%	1
2.6.19.7	3/3/2007	6,684,211	3,046,366	45.6%	0
2.6.20.2	3/9/2007	6,738,722	3,071,155	45.6%	6
2.6.16.44	3/20/2007	5,837,820	2,661,468	45.6%	11
2.6.20.4	3/23/2007	6,775,308	3,087,793	45.6%	3
2.4.34.2	3/24/2007	3,328,112	1,520,173	45.7%	1
2.6.16.45	3/24/2007	5,853,650	2,668,667	45.6%	0
2.6.16.46	3/31/2007	5,869,525	2,675,886	45.6%	7
2.6.20.5	4/6/2007	6,793,675	3,096,145	45.6%	6
2.6.20.6	4/6/2007	6,812,093	3,104,521	45.6%	0
2.6.16.47	4/13/2007	5,885,444	2,683,125	45.6%	7
2.6.20.7	4/13/2007	6,830,561	3,112,919	45.6%	0
2.6.16.48	4/15/2007	5,901,404	2,690,383	45.6%	2
2.4.34.3	4/22/2007	3,336,708	1,524,082	45.7%	7
2.4.34.4	4/22/2007	3,345,324	1,528,000	45.7%	0
2.6.16.49	4/22/2007	5,917,406	2,697,660	45.6%	0
2.6.20.8	4/25/2007	6,849,076	3,121,339	45.6%	3
2.6.20.9	4/26/2007	6,867,645	3,129,783	45.6%	1
2.6.21	4/26/2007	7,113,638	3,241,649	45.6%	0
2.6.20.10	4/27/2007	6,886,261	3,138,249	45.6%	1
2.6.21.1	4/27/2007	7,132,921	3,250,418	45.6%	0
2.6.20.11	5/2/2007	6,904,931	3,146,739	45.6%	5
2.6.16.50	5/3/2007	5,933,454	2,704,958	45.6%	1
2.6.16.51	5/9/2007	5,949,544	2,712,275	45.6%	6
2.6.21.2	5/23/2007	7,152,257	3,259,211	45.6%	14
2.6.20.12	5/24/2007	6,923,649	3,155,251	45.6%	1
2.6.21.3	5/24/2007	7,171,643	3,268,027	45.6%	0
2.6.16.52	5/30/2007	5,965,678	2,719,612	45.6%	6

2.4.34.5	6/6/2007	3,353,964	1,531,929	45.7%	7
2.6.20.13	6/7/2007	6,942,417	3,163,786	45.6%	1
2.6.21.4	6/7/2007	7,191,084	3,276,868	45.6%	0
2.6.20.14	6/11/2007	6,961,236	3,172,344	45.6%	4
2.6.21.5	6/11/2007	7,210,576	3,285,732	45.6%	0
2.6.20.15	7/7/2007	6,980,108	3,180,926	45.6%	26
2.6.21.6	7/7/2007	7,230,121	3,294,620	45.6%	0
2.6.22	7/8/2007	7,269,371	3,312,469	45.6%	1
2.6.22.1	7/10/2007	7,289,074	3,321,429	45.6%	2
2.4.34.6	7/22/2007	3,362,623	1,535,867	45.7%	12
2.6.16.53	7/25/2007	5,981,857	2,726,969	45.6%	3
2.4.35	7/26/2007	3,371,307	1,539,816	45.7%	1
2.6.21.7	8/4/2007	7,249,719	3,303,532	45.6%	9
2.6.22.2	8/9/2007	7,308,832	3,330,414	45.6%	5
2.4.35.1	8/15/2007	3,380,013	1,543,775	45.7%	6
2.6.22.3	8/15/2007	7,328,643	3,339,423	45.6%	0
2.6.20.16	8/16/2007	6,999,028	3,189,530	45.6%	1
2.6.22.4	8/21/2007	7,348,506	3,348,456	45.6%	5
2.6.22.5	8/22/2007	7,368,425	3,357,514	45.6%	1
2.6.20.17	8/25/2007	7,018,001	3,198,158	45.6%	3
2.6.20.18	8/28/2007	7,037,027	3,206,810	45.6%	3
2.6.22.6	8/31/2007	7,388,398	3,366,597	45.6%	3
2.4.35.2	9/8/2007	3,388,741	1,547,744	45.7%	8
2.6.20.19	9/8/2007	7,056,101	3,215,484	45.6%	0
2.6.22.7	9/21/2007	7,408,425	3,375,704	45.6%	13
2.4.35.3	9/23/2007	3,397,491	1,551,723	45.7%	2
2.6.20.20	9/23/2007	7,075,228	3,224,182	45.6%	0
2.6.22.8	9/25/2007	7,428,504	3,384,835	45.6%	2
2.6.22.9	9/26/2007	7,448,638	3,393,991	45.6%	1
2.6.16.54	10/6/2007	5,998,076	2,734,345	45.6%	10
2.6.23	10/9/2007	7,673,747	3,496,360	45.6%	3
2.6.22.10	10/10/2007	7,468,827	3,403,172	45.6%	1
2.6.16.55	10/12/2007	6,014,342	2,741,742	45.6%	2
2.6.23.1	10/12/2007	7,694,545	3,505,818	45.6%	0
2.6.20.3	10/17/2007	6,756,991	3,079,463	45.6%	5
2.6.20.21	10/17/2007	7,094,408	3,232,904	45.6%	0
2.6.16.56	11/1/2007	6,030,652	2,749,159	45.6%	15
2.6.22.11	11/2/2007	7,489,071	3,412,378	45.6%	1

2.6.16.57	11/5/2007	6,047,004	2,756,595	45.6%	3
2.6.22.12	11/5/2007	7,509,370	3,421,609	45.6%	0
2.6.22.13	11/16/2007	7,529,724	3,430,865	45.6%	11
2.6.23.2	11/16/2007	7,715,399	3,515,301	45.6%	0
2.6.23.3	11/16/2007	7,736,311	3,524,811	45.6%	0
2.6.23.4	11/16/2007	7,757,276	3,534,345	45.6%	0
2.6.23.5	11/16/2007	7,778,301	3,543,906	45.6%	0
2.6.23.6	11/16/2007	7,799,383	3,553,493	45.6%	0
2.6.23.7	11/16/2007	7,820,520	3,563,105	45.6%	0
2.6.23.8	11/16/2007	7,841,716	3,572,744	45.6%	0
2.4.35.4	11/17/2007	3,406,265	1,555,713	45.7%	1
2.6.22.14	11/21/2007	7,550,133	3,440,146	45.6%	4
2.6.23.9	11/26/2007	7,862,967	3,582,408	45.6%	5
2.6.22.15	12/14/2007	7,570,597	3,449,452	45.6%	18
2.6.23.10	12/14/2007	7,884,277	3,592,099	45.6%	0
2.6.23.11	12/15/2007	7,905,645	3,601,816	45.6%	1
2.4.35.5	12/16/2007	3,415,061	1,559,713	45.7%	1
2.6.23.12	12/18/2007	7,927,070	3,611,559	45.6%	2
2.4.36	1/1/2008	3,423,879	1,563,723	45.7%	14
2.6.16.58	1/6/2008	6,063,402	2,764,052	45.6%	5
2.6.23.13	1/9/2008	7,948,554	3,621,329	45.6%	3
2.6.22.16	1/14/2008	7,591,116	3,458,783	45.6%	5
2.6.23.14	1/14/2008	7,970,096	3,631,125	45.6%	0
2.6.16.59	1/19/2008	6,079,844	2,771,529	45.6%	5
2.6.24	1/24/2008	8,056,844	3,670,574	45.6%	5
2.6.16.60	1/27/2008	6,096,330	2,779,026	45.6%	3
2.6.22.17	2/6/2008	7,611,689	3,468,139	45.6%	10
2.6.23.15	2/8/2008	7,991,694	3,640,947	45.6%	2
2.6.24.1	2/8/2008	8,078,680	3,680,504	45.6%	0
2.6.22.18	2/11/2008	7,632,318	3,477,520	45.6%	3
2.6.23.16	2/11/2008	8,013,352	3,650,796	45.6%	0
2.6.24.2	2/11/2008	8,100,573	3,690,460	45.6%	0
2.4.36.1	2/16/2008	3,432,719	1,567,743	45.7%	5
2.4.36.2	2/24/2008	3,441,583	1,571,774	45.7%	8
2.6.22.19	2/26/2008	7,653,004	3,486,927	45.6%	2
2.6.23.17	2/26/2008	8,035,069	3,660,672	45.6%	0
2.6.24.3	2/26/2008	8,122,526	3,700,443	45.6%	0
2.6.24.4	3/24/2008	8,144,538	3,710,453	45.6%	27

2.6.25	4/17/2008	8,233,182	3,750,764	45.6%	24
2.4.36.3	4/19/2008	3,450,469	1,575,815	45.7%	2
2.6.24.5	4/19/2008	8,166,609	3,720,490	45.6%	0
2.6.24.6	5/1/2008	8,188,740	3,730,554	45.6%	12
2.6.25.1	5/1/2008	8,255,493	3,760,910	45.6%	0
2.4.36.4	5/7/2008	3,459,377	1,579,866	45.7%	6
2.6.24.7	5/7/2008	8,210,932	3,740,646	45.6%	0
2.6.25.2	5/7/2008	8,277,865	3,771,084	45.6%	0
2.6.25.3	5/10/2008	8,300,297	3,781,285	45.6%	3
2.6.25.4	5/15/2008	8,322,791	3,791,514	45.6%	5
2.4.36.5	6/1/2008	3,468,309	1,583,928	45.7%	17
2.4.36.6	6/6/2008	3,477,266	1,588,001	45.7%	5
2.6.25.5	6/6/2008	8,345,344	3,801,770	45.6%	0
2.6.25.6	6/9/2008	8,367,958	3,812,054	45.6%	3
2.6.25.7	6/16/2008	8,390,635	3,822,366	45.6%	7
2.6.25.8	6/22/2008	8,413,372	3,832,706	45.6%	6
2.6.25.9	6/24/2008	8,436,171	3,843,074	45.6%	2
2.6.25.10	7/3/2008	8,459,032	3,853,470	45.6%	9
2.6.25.11	7/13/2008	8,481,955	3,863,894	45.6%	10
2.6.26	7/13/2008	8,714,615	3,969,697	45.6%	0
2.6.16.61	7/16/2008	6,112,862	2,786,544	45.6%	3
2.6.16.62	7/21/2008	6,129,438	2,794,082	45.6%	5
2.6.25.12	7/24/2008	8,504,939	3,874,346	45.6%	3
2.6.25.13	7/28/2008	8,527,984	3,884,826	45.6%	4
2.6.25.14	8/1/2008	8,551,093	3,895,335	45.6%	4
2.6.26.1	8/1/2008	8,738,228	3,980,435	45.6%	0
2.6.25.15	8/6/2008	8,574,264	3,905,872	45.6%	5
2.6.26.2	8/6/2008	8,761,907	3,991,203	45.6%	0
2.6.25.16	8/20/2008	8,597,499	3,916,438	45.6%	14
2.6.26.3	8/20/2008	8,785,647	4,001,999	45.6%	0
2.4.36.7	9/7/2008	3,486,242	1,592,083	45.7%	18
2.6.25.17	9/8/2008	8,620,795	3,927,032	45.6%	1
2.6.26.4	9/8/2008	8,809,454	4,012,825	45.6%	0
2.6.26.5	9/8/2008	8,833,324	4,023,680	45.6%	0
2.6.25.18	10/9/2008	8,644,155	3,937,655	45.6%	31
2.6.26.6	10/9/2008	8,857,258	4,034,564	45.6%	0
2.6.27	10/9/2008	8,929,453	4,067,395	45.6%	0
2.6.27.1	10/15/2008	8,953,647	4,078,397	45.6%	6

2.6.27.2	10/18/2008	8,977,908	4,089,430	45.5%	3
2.4.36.8	10/19/2008	3,495,245	1,596,177	45.7%	1
2.6.25.19	10/22/2008	8,667,579	3,948,307	45.6%	3
2.6.26.7	10/22/2008	8,881,258	4,045,478	45.6%	0
2.6.27.3	10/22/2008	9,002,234	4,100,492	45.5%	0
2.6.27.4	10/26/2008	9,026,625	4,111,584	45.5%	4
2.6.27.5	11/7/2008	9,051,082	4,122,706	45.5%	12
2.4.36.9	11/9/2008	3,504,267	1,600,280	45.7%	2
2.6.25.20	11/10/2008	8,691,064	3,958,987	45.6%	1
2.6.26.8	11/10/2008	8,905,324	4,056,422	45.6%	0
2.6.27.6	11/13/2008	9,075,605	4,133,858	45.5%	3
2.6.27.7	11/20/2008	9,100,197	4,145,041	45.5%	7
2.4.37	12/2/2008	3,513,316	1,604,395	45.7%	12
2.6.27.8	12/5/2008	9,124,854	4,156,254	45.5%	3
2.6.27.9	12/14/2008	9,149,578	4,167,497	45.5%	9
2.6.27.10	12/18/2008	9,174,367	4,178,770	45.5%	4
2.6.28	12/24/2008	10,446,612	4,757,326	45.5%	6
2.6.27.11	1/14/2009	9,199,224	4,190,074	45.5%	21
2.6.27.12	1/18/2009	9,224,148	4,201,408	45.5%	4
2.6.28.1	1/18/2009	10,474,910	4,770,195	45.5%	0
2.6.27.13	1/25/2009	9,249,142	4,212,774	45.5%	7
2.6.28.2	1/25/2009	10,503,286	4,783,099	45.5%	0
2.6.27.14	2/2/2009	9,274,202	4,224,170	45.5%	8
2.6.28.3	2/2/2009	10,531,737	4,796,037	45.5%	0
2.6.27.15	2/6/2009	9,299,327	4,235,596	45.5%	4
2.6.28.4	2/6/2009	10,560,267	4,809,011	45.5%	0
2.6.27.16	2/12/2009	9,324,523	4,247,054	45.5%	6
2.6.28.5	2/12/2009	10,588,874	4,822,020	45.5%	0
2.6.27.17	2/13/2009	9,349,788	4,258,543	45.5%	1
2.6.27.18	2/17/2009	9,375,118	4,270,062	45.5%	4
2.6.28.6	2/17/2009	10,617,557	4,835,064	45.5%	0
2.6.27.19	2/20/2009	9,400,519	4,281,613	45.5%	3
2.6.28.7	2/20/2009	10,646,318	4,848,143	45.5%	0
2.6.27.20	3/17/2009	9,425,987	4,293,195	45.5%	25
2.6.28.8	3/17/2009	10,675,158	4,861,258	45.5%	0
2.6.27.21	3/23/2009	9,451,527	4,304,809	45.5%	6
2.6.28.9	3/23/2009	10,704,075	4,874,408	45.5%	0
2.6.29	3/23/2009	10,762,144	4,900,815	45.5%	0

2.6.29.1	4/2/2009	10,791,296	4,914,072	45.5%	10
2.4.37.1	4/19/2009	3,522,387	1,608,520	45.7%	17
2.6.29.2	4/27/2009	10,820,527	4,927,365	45.5%	8
2.6.27.22	5/2/2009	9,477,134	4,316,454	45.5%	5
2.6.28.10	5/2/2009	10,733,069	4,887,593	45.5%	0
2.6.27.23	5/8/2009	9,502,810	4,328,130	45.5%	6
2.6.29.3	5/8/2009	10,849,838	4,940,694	45.5%	0
2.6.27.24	5/20/2009	9,528,555	4,339,838	45.5%	12
2.6.29.4	5/20/2009	10,879,227	4,954,059	45.5%	0
2.4.37.2	6/7/2009	3,531,480	1,612,655	45.7%	18
2.6.30	6/10/2009	10,967,874	4,994,371	45.5%	3
2.6.27.25	6/12/2009	9,554,372	4,351,578	45.5%	2
2.6.29.5	6/15/2009	10,908,696	4,967,460	45.5%	3
2.6.27.26	7/2/2009	9,580,256	4,363,349	45.5%	17
2.6.29.6	7/2/2009	10,938,244	4,980,897	45.5%	0
2.6.30.1	7/2/2009	10,997,582	5,007,881	45.5%	0
2.4.37.3	7/19/2009	3,540,597	1,616,801	45.7%	17
2.6.27.27	7/20/2009	9,606,211	4,375,152	45.5%	1
2.6.30.2	7/20/2009	11,027,372	5,021,428	45.5%	0
2.6.27.28	7/24/2009	9,632,236	4,386,987	45.5%	4
2.6.30.3	7/24/2009	11,057,241	5,035,011	45.5%	0
2.4.37.4	7/26/2009	3,549,738	1,620,958	45.7%	2
2.6.27.29	7/30/2009	9,658,334	4,398,855	45.5%	4
2.6.30.4	7/30/2009	11,087,193	5,048,632	45.5%	0
2.4.37.5	8/13/2009	3,558,904	1,625,126	45.7%	14
2.6.27.30	8/16/2009	9,684,500	4,410,754	45.5%	3
2.6.30.5	8/16/2009	11,117,225	5,062,289	45.5%	0
2.6.27.31	8/17/2009	9,710,736	4,422,685	45.5%	1
2.6.27.32	9/9/2009	9,737,045	4,434,649	45.5%	23
2.6.27.33	9/9/2009	9,763,424	4,446,645	45.5%	0
2.6.30.6	9/9/2009	11,147,336	5,075,982	45.5%	0
2.6.31	9/9/2009	11,299,129	5,145,010	45.5%	0
2.4.37.6	9/13/2009	3,568,091	1,629,304	45.7%	4
2.6.27.34	9/15/2009	9,789,876	4,458,674	45.5%	2
2.6.30.7	9/15/2009	11,177,531	5,089,713	45.5%	0
2.6.27.35	9/24/2009	9,816,398	4,470,735	45.5%	9
2.6.30.8	9/24/2009	11,207,809	5,103,482	45.5%	0
2.6.31.1	9/24/2009	11,329,732	5,158,927	45.5%	0

2.6.27.36	10/5/2009	9,842,990	4,482,828	45.5%	11
2.6.30.9	10/5/2009	11,238,166	5,117,287	45.5%	0
2.6.31.2	10/5/2009	11,360,421	5,172,883	45.5%	0
2.6.31.3	10/7/2009	11,391,192	5,186,876	45.5%	2
2.6.27.37	10/12/2009	9,869,658	4,494,955	45.5%	5
2.6.31.4	10/12/2009	11,422,046	5,200,907	45.5%	0
2.6.27.38	10/22/2009	9,896,395	4,507,114	45.5%	10
2.6.31.5	10/22/2009	11,452,984	5,214,976	45.5%	0
2.4.37.7	11/7/2009	3,577,303	1,633,493	45.7%	16
2.6.27.39	11/10/2009	9,923,206	4,519,306	45.5%	3
2.6.31.6	11/10/2009	11,484,005	5,229,083	45.5%	0
2.6.32	12/3/2009	11,766,999	5,357,775	45.5%	23
2.6.30.10	12/4/2009	11,268,607	5,131,130	45.5%	1
2.6.27.40	12/8/2009	9,950,088	4,531,531	45.5%	4
2.6.27.41	12/8/2009	9,977,046	4,543,790	45.5%	0
2.6.27.42	12/8/2009	10,004,074	4,556,081	45.5%	0
2.6.31.7	12/8/2009	11,515,110	5,243,228	45.5%	0
2.6.31.8	12/14/2009	11,546,298	5,257,411	45.5%	6
2.6.32.1	12/14/2009	11,798,869	5,372,268	45.5%	0
2.6.31.9	12/18/2009	11,577,573	5,271,633	45.5%	4
2.6.32.2	12/18/2009	11,830,827	5,386,801	45.5%	0
2.6.27.43	1/6/2010	10,031,174	4,568,405	45.5%	19
2.6.31.10	1/6/2010	11,608,930	5,285,893	45.5%	0
2.6.32.3	1/6/2010	11,862,869	5,401,372	45.5%	0
2.6.31.11	1/7/2010	11,640,374	5,300,192	45.5%	1
2.6.27.44	1/18/2010	10,058,350	4,580,763	45.5%	11
2.6.31.12	1/18/2010	11,671,901	5,314,529	45.5%	0
2.6.32.4	1/18/2010	11,894,998	5,415,983	45.5%	0
2.6.32.5	1/22/2010	11,927,216	5,430,634	45.5%	4
2.6.32.6	1/25/2010	11,959,519	5,445,324	45.5%	3
2.6.27.45	1/28/2010	10,085,600	4,593,155	45.5%	3
2.6.32.7	1/28/2010	11,991,913	5,460,055	45.5%	0
2.4.37.8	1/31/2010	3,586,539	1,637,693	45.7%	3
2.4.37.9	2/1/2010	3,595,796	1,641,903	45.7%	1
2.6.32.8	2/9/2010	12,024,390	5,474,824	45.5%	8
2.6.32.9	2/23/2010	12,056,957	5,489,634	45.5%	14
2.6.33	2/24/2010	12,692,741	5,778,758	45.5%	1
2.6.32.10	3/15/2010	12,089,612	5,504,484	45.5%	19

2.6.33.1	3/15/2010	12,727,117	5,794,391	45.5%	0
2.6.27.46	4/1/2010	10,112,922	4,605,580	45.5%	17
2.6.31.13	4/1/2010	11,703,516	5,328,906	45.5%	0
2.6.32.11	4/1/2010	12,122,355	5,519,374	45.5%	0
2.6.33.2	4/1/2010	12,761,585	5,810,065	45.5%	0
2.6.32.12	4/26/2010	12,155,188	5,534,305	45.5%	25
2.6.33.3	4/26/2010	12,796,146	5,825,782	45.5%	0
2.6.32.13	5/12/2010	12,188,107	5,549,275	45.5%	16
2.6.33.4	5/12/2010	12,830,800	5,841,541	45.5%	0
2.6.34	5/16/2010	12,970,358	5,905,005	45.5%	4
2.6.27.47	5/26/2010	10,140,317	4,618,038	45.5%	10
2.6.32.14	5/26/2010	12,221,119	5,564,287	45.5%	0
2.6.33.5	5/26/2010	12,865,549	5,857,343	45.5%	0
2.6.32.15	6/1/2010	12,254,218	5,579,339	45.5%	6
2.6.27.48	7/5/2010	10,167,787	4,630,530	45.5%	34
2.6.31.14	7/5/2010	11,735,215	5,343,321	45.5%	0
2.6.32.16	7/5/2010	12,287,405	5,594,431	45.5%	0
2.6.33.6	7/5/2010	12,900,390	5,873,187	45.5%	0
2.6.34.1	7/5/2010	13,005,485	5,920,979	45.5%	0
2.6.35	8/1/2010	13,254,040	6,034,010	45.5%	27
2.6.27.49	8/2/2010	10,195,332	4,643,056	45.5%	1
2.6.32.17	8/2/2010	12,320,685	5,609,565	45.5%	0
2.6.33.7	8/2/2010	12,935,328	5,889,075	45.5%	0
2.6.34.2	8/2/2010	13,040,704	5,936,995	45.5%	0
2.6.27.50	8/10/2010	10,222,951	4,655,616	45.5%	8
2.6.32.18	8/10/2010	12,354,053	5,624,739	45.5%	0
2.6.34.3	8/10/2010	13,076,022	5,953,056	45.5%	0
2.6.35.1	8/10/2010	13,289,932	6,050,332	45.5%	0
2.6.27.51	8/13/2010	10,250,645	4,668,210	45.5%	3
2.6.32.19	8/13/2010	12,387,511	5,639,954	45.5%	0
2.6.34.4	8/13/2010	13,111,432	5,969,159	45.5%	0
2.6.35.2	8/13/2010	13,325,923	6,066,699	45.5%	0
2.6.27.52	8/20/2010	10,278,414	4,680,838	45.5%	7
2.6.32.20	8/20/2010	12,421,061	5,655,211	45.5%	0
2.6.34.5	8/20/2010	13,146,940	5,985,306	45.5%	0
2.6.35.3	8/20/2010	13,362,011	6,083,110	45.5%	0
2.6.27.53	8/27/2010	10,306,258	4,693,500	45.5%	7
2.6.32.21	8/27/2010	12,454,701	5,670,509	45.5%	0

2.6.34.6	8/27/2010	13,182,544	6,001,497	45.5%	0
2.6.35.4	8/27/2010	13,398,195	6,099,565	45.5%	0
2.4.37.10	9/6/2010	3,605,081	1,646,125	45.7%	10
2.6.34.7	9/13/2010	13,218,242	6,017,731	45.5%	7
2.6.27.54	9/20/2010	10,334,177	4,706,196	45.5%	7
2.6.32.22	9/20/2010	12,488,431	5,685,848	45.5%	0
2.6.35.5	9/20/2010	13,434,479	6,116,065	45.5%	0
2.6.32.23	9/27/2010	12,522,254	5,701,229	45.5%	7
2.6.35.6	9/27/2010	13,470,861	6,132,610	45.5%	0
2.6.35.7	9/29/2010	13,507,340	6,149,199	45.5%	2
2.6.32.24	10/1/2010	12,556,167	5,716,651	45.5%	2
2.6.36	10/20/2010	13,617,372	6,199,236	45.5%	19
2.6.27.55	10/29/2010	10,362,172	4,718,927	45.5%	9
2.6.32.25	10/29/2010	12,590,173	5,732,115	45.5%	0
2.6.35.8	10/29/2010	13,543,918	6,165,833	45.5%	0
2.6.27.56	11/22/2010	10,390,242	4,731,692	45.5%	24
2.6.32.26	11/22/2010	12,624,270	5,747,621	45.5%	0
2.6.35.9	11/22/2010	13,580,596	6,182,512	45.5%	0
2.6.36.1	11/22/2010	13,654,249	6,216,006	45.5%	0
2.6.27.57	12/9/2010	10,418,390	4,744,492	45.5%	17
2.6.32.27	12/9/2010	12,658,460	5,763,169	45.5%	0
2.6.36.2	12/9/2010	13,691,225	6,232,821	45.5%	0
2.4.37.11	12/18/2010	3,614,387	1,650,357	45.7%	9
2.6.37	1/5/2011	13,802,752	6,283,538	45.5%	18
2.6.36.3	1/7/2011	13,728,300	6,249,681	45.5%	2
2.6.36.4	2/17/2011	13,765,476	6,266,587	45.5%	41
2.6.37.1	2/17/2011	13,840,130	6,300,536	45.5%	0
Total		4,410,970,884	2,012,427,320		6185
Average		4,528,718	2,066,147	46.0%	6.4
Min		165,768	83,696	38.2%	0
Max		13,840,130	6,300,536	50.5%	98

APPENDIX D

SCHEME LANGUAGE AND EXCEL MACRO SCRIPTS

This appendix gives the source code scripts used to obtain the slice size of the *CodeSurfer* tool. The slicing process was required to collect the slice size in order to perform slice intersection calculations. The *srcSlice* tool was designed to generate the slice size automatically in number of statements.

A.1 Slice Size Data Generation Script (Scheme Code)

CodeSurfer has API which allows functional programming to be used as an extension of the capabilities of the tool and also to allow user to perform user-defined operation and obtain user-defined results. The script to obtain the slice size data from *CodeSurfer* was written in functional programming language *Scheme*.

The algorithm of the following script is as follows:

1. Get a list of all program points in the PDG
2. Perform slicing using the nodes from the list
3. Calculate the number of program points in the slice
4. Output the size of the slice into a specified file
5. Repeat from step 2 to 4 until end of list.

To perform slicing on a particular code, the project for that program needs to be open in *CodeSurfer*. Once *CodeSurfer* has completed the slicing process it will output a list of statement numbers that included in each file in the system. It should be noted that the slice sizes are not calculated automatically and will need removing duplicate lines before the slice size for each file can be calculated. The filtering is done using EXCEL automated analysis macros developed which will be detailed in Appendix A.2.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;
;; Name: (ex:get-displayed-set name)
;; Args: name : SYMBOL
;; Returns: PDG_VERTEX_SET | #f
;; Action:
;; Returns the points associated with the Displayed Set specified
;; by name. If the there is no Displayed Set with this name,
returns
;; #f.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;
;; example: (ex:get-displayed-set 'query-results)
(define (ex:get-displayed-set name)
(dht:get-pdgvs ss:main-project name))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;
;; Name: (ex:set-displayed-set name vs)
;; Args: name : SYMBOL
;; vs : PDG_VERTEX_SET
;; PreCond: name must be an existing displayed set.
;; Returns: UNDEFINED
;; Action:
;; Sets vs to be the points associated with the Displayed Set
specified
;; by name.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;
;; example: (ex:set-displayed-set 'query-results (pdg-vertex-set-
create))
;; clears the query results.
(define (ex:set-displayed-set name vs)
(vi:set-displayed-set name vs))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;
;; Name: (ex:sdg-sourcefile-vertex vs)
;; Args: vs: an empty pdg-vertex-set
;; Returns: PDG_VERTEX_SET
;; Action:
;; Return all the vertex in the source file.
```

```
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;
(define (ex:sdg-sourcefile-vertex)
(define vs (pdg-vertex-set-create))
(let ((pdg-list (ex:prj-sourcefile-pdglist)))
(for-each
(lambda (pdg)
(pdg-vertex-set-union! vs (pdg-vertices pdg))
)
pdg-list))
vs)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;
;; Name: (ex:prj-sourcefile-pdglist)
;; Returns: a list of user defined pdgs in source files
;; Action:
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;
(define (ex:prj-sourcefile-pdglist)
(let ((pdg-list '( ))
(file-list (prj:contribute-nonlib-source-files ss:main-project)))
(for-each (lambda (file)
(let ((filepdgs (file:functions-in-file file)))
(set! pdg-list (append pdg-list filepdgs))))
file-list)
pdg-list))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;
;; Name: (ex:sdg-source-vertex-byprj vset)
;; Args: vset : vertex-set of sdg
;; prjname : project name
;; Returns: all vertices of prjname which corresponding the
source code
;; Action:
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;
(define (ex:sdg-source-vertex-byprj vset)
(let ((file-list (prj:contribute-nonlib-source-files ss:main-
project)))
;(let ((file-list (prj:topincludes ss:main-project)))
;(let ((file-list (prj:source-file-names-no-duplicate ss:main-
project)))
(for-each
(lambda (files)
;;count the pLOC for each source file
; (let* ((fileuid (file:uid files))
; (fileloc (file-get-linecount fileuid))
; (filename (file:fname-base-only (file:name files))))
; (format #t "~a:~a:~a lines\n" fileuid filename fileloc))
(define line-number 1)
(let* ((infilename (file:name files))
(in-port (open-input-file infilename))
(prj ss:main-project) ;; there is only one project
)
(define oneline-of-code (read-line in-port))
(while (not (eof-object? oneline-of-code))
(pdg-vertex-set-union! vset (file:vertices-in-line files line-
number))
(set! oneline-of-code (read-line in-port))
(set! line-number (+ 1 line-number))
)
(close-input-port in-port)
)
)
file-list)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;
;; Name: (ex:prj-vertex-b-slice-source-numberofvertex
outfilename)
;; Args: outfilename :
;; Returns: number of vertex of backward slices
;; Action:
;; backward slicing every program point that corresponding the
source code in a project
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;
(define (ex:prj-vertex-b-slice-source-numberofvertex outfilename)
(let ((out-port (open-output-file outfilename )))
 (define vset-sourcefile (pdg-vertex-set-create))
 (ex:sdg-source-vertex-byprj vset-sourcefile)
 (define vset-list (pdg-vertex-set-sort vset-sourcefile ""))
  (for-each
   (lambda (vertex)
           (let* ((vs (pdg-vertex-set-create)))
            (pdg-vertex-set-put! vs vertex)
;            (vi:forward-slice ss:main-project-viewer vs)
            (vi:slice ss:main-project-viewer vs)
            (let* ((querypoints (set-cardinality (ex:get-
displayed-set 'query-points)))
;               (vset-queryresults (ex:get-displayed-set 'query-
results))
              (queryresults (set-cardinality (pdg-vertex-set-
intersect
                                        vset-sourcefile
```

```
                                                    (ex:get-displayed-set
'query-results))))
                      )
;                      (write querypoints out-port)
;                      (write-char #\tab out-port)
                      (write queryresults out-port)
                      (newline out-port)
                )
            (ex:set-displayed-set 'query-points (pdg-vertex-set-
create))
                (ex:set-displayed-set 'query-results (pdg-vertex-set-
create))
           )
      )
    vset-list)
 (close-output-port out-port)))
```

A.2 Filter Automated Analysis Script (Excel Macro)

The Excel macro script used to remove duplicated lines and calculate the slice size for each file in the system in the *CodeSurfer* is as follows:

```
' Filter for each file the slice lines
'''''''''''''''''''''''''''''''''''''''''''''''''''''''''
Sub Slice-Size()
Dim myRange As Variant
Dim row As Integer
Dim cell As Range, myBin As Range, myData As Range, newrange As
Range
Dim sh As Worksheet
Dim Rang As Range, UniqueValues As New Collection
Set sh = Sheets("filter")
With sh
Set cell = Cells(1, 2)
Set cellset = .Range("F1:ZZ2000")
Set mycount = .Range("D1:D30000")
' this is the name of the source file
myRange = .Range("A1:A30000").Value
var1 = cell.Address
var2 = cell.Address
Counter1 = 2
Counter2 = 1
```

```
.Range("D1").Value = "File Name"
For row = 1 To 24999
If myRange(row, 1) = myRange(row + 1, 1) Then
var2 = Cells (row + 1, 2).Address
Else
Set newrange = .Range (var1, var2)
var2 = Cells (row + 1, 2).Address
var1 = var2
On Error Resume Next
' ignore any subsequent error
For Each Rang In newrange
UniqueValues.Add Rang.Value, CStr(Rang.Value)
' add the unique item
Next Rang
On Error GoTo 0
countuniquevalues = UniqueValues.Count
mycount (Counter1, 1) = myRange(row, 1)
.Range("e" & Counter1).Value = countuniquevalues
Counter1 = Counter1 + 1
cellset(1, Counter2) = myRange(row, 1)
For w = 1 To countuniquevalues
cellset(w + 1, Counter2) = UniqueValues(w)
Next w
Counter2 = Counter2 + 1
Set UniqueValues = Nothing
End If
Next row
End With
End Sub
```

RERERENCES

[Alomari, Collard, Maletic 2012] Alomari, H. W., Collard, M. L., and Maletic, J. I., (2012), "Lightweight Forward Static Slicing".unpublished.

[Alomari, Collard, Maletic 2012] Alomari, H. W., Collard, M. L., and Maletic, J. I., (2012), "A Slice-Based Estimation Approach for Maintenance Effort".unpublished.

[Asundi 2005] Asundi, J., (2005), "The Need for Effort Estimation Models for Open Source Software Projects", SIGSOFT Software Engineering Notes, vol. 30, no. 4, pp. 1-3.

[Belady, Lehman 1972] Belady, L., and Lehman, M. M., (1972), "An Introduction to Program Growth Dynamics", Academic Press, New York, pp. 503-511.

[Bent, Atkinson, Griswold 2000] Bent, L., Atkinson, D. C., and Griswold, W. G., (2000), "A Qualitative Study of Two Whole-Program Slicers for C", University of California at San Diego. A preliminary version appeared at FSE '00, Technical Report CS20000643.

[Bergeretti, Carre' 1985] Bergeretti, J.-F., and Carre', B. A., (1985), "Information-Flow and Data-Flow Analysis of While-Programs", ACM Transactions on Programming Languages and Systems (TOPLAS'85), vol. 7, no. 1, pp. 37-61.

[Binkley, Schach 1997] Binkley, A. B., and Schach, S. R., (1997), "Inheritance-Based Metrics for Predicting Maintenance Effort: An Empirical Study": C. S. Department, Vanderbilt University, Technical Report 97-05.

[Binkley 1993] Binkley, D., (1993), "Slicing in the Presence of Parameter Aliasing", In Proceedings of the Third Software Engineering Research Forum, Orlando, Florida, pp. 261-268.

[Binkley, Gallagher 1996] Binkley, D., and Gallagher, K. B., (1996), "Program Slicing", Advances in Computers, vol. 43, no., pp. 1-50.

[Binkley, Gold, Harman 2007] Binkley, D., Gold, N., and Harman, M., (2007), "An Empirical Study of Static Program Slice Size", ACM Transactions on Software Engineering and Methodology, vol. 16, no. 2, pp. 1-32.

[Binkley, Gold, Harman, Li, Mahdavi 2006] Binkley, D., Gold, N., Harman, M., Li, Z., and Mahdavi, K., (2006), "An Empirical Study of Executable Concept Slice Size", in Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06), Benevento, Italy, pp. 103-114.

[Binkley, Harman 2003] Binkley, D., and Harman, M., (2003), "A Large-Scale Empirical Study of Forward and Backward Static Slice Size and Context Sensitivity ", in Proceedings of the International Conference on Software Maintenance (ICSM'03), September 22-26, pp. 44-53.

[Binkley, Harman 2004] Binkley, D., and Harman, M., (2004), "A Survey of Empirical Results on Program Slicing", Advances in Computers, vol. 62, no., pp. 105-178.

[Binkley, Harman, Hassoun, Islam, Li 2010] Binkley, D., Harman, M., Hassoun, Y., Islam, S., and Li, Z., (2010), "Assessing the Impact of Global Variables on Program Dependence and Dependence Clusters", Journal of Software Systems, vol. 83, no. 1, pp. 96-107.

[Boehm, Clark, Horowitz, Westland, Madachy, Selby 1995] Boehm, B., Clark, B., Horowitz, E., Westland, C., Madachy, R., and Selby, R., (1995), "Cost Models for Future Software Life Cycle Processes: COCOMO 2.0", Annals of Software Engineering, vol. 1, no. 1, pp. 57-94.

[Boehm 2002] Boehm, B. W., (2002). Software Engineering Economics. Book "Software Engineering Economics", Springer-Verlag New York, Inc.: 641-686.

[Capiluppi, Lago, Morisio 2003] Capiluppi, A., Lago, P., and Morisio, M., (2003), "Characteristics of Open Source Projects", in Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR'03), pp. 317-327.

[Chapin, Hale, Kham, Ramil, Tan 2001] Chapin, N., Hale, J. E., Kham, K. M., Ramil, J. F., and Tan, W.-G., (2001), "Types of software evolution and software maintenance", Journal of Software Maintenance, vol. 13, no., pp. 3-30.

[Chen, Schach, Yu, Offutt, Heller 2004] Chen, K., Schach, S. R., Yu, L., Offutt, J., and Heller, G. Z., (2004), "Open-Source Change Logs", Empirical Software Engining, vol. 9, no. 3, pp. 197-210.

[CodeSurfer] CodeSurfer, "CodeSurfer R. Grammatech Inc":http://www.grammatech.com/products/codesurfer.

[Collard 2003] Collard, M. L., (2003), "An Infrastructure to Support Meta-Differencing and Refactoring of Source Code", in Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE'03), Montreal, Quebec, Canada, pp. 377-380

[Collard, Decker, Maletic 2011] Collard, M. L., Decker, M. J., and Maletic, J. I., (2011), "Lightweight Transformation and Fact Extraction with the srcML Toolkit", in Proceedings of the 11th International Working Conference on Source Code Analysis and Manipulation (SCAM'11), Wiliamsburg, VA, USA, pp. 173-184.

[Collard, Kagdi, Maletic 2003] Collard, M. L., Kagdi, H. H., and Maletic, J. I., (2003), "An XML-Based Lightweight C++ Fact Extractor", in Proceedings of the IEEE International Workshop on Program Comprehension (IWPC03), Portland, OR, USA, May 10-11, pp. 134-143.

[Collard, Maletic, Robinson 2010] Collard, M. L., Maletic, J. I., and Robinson, B. P., (2010), "A Lightweight Transformational Approach to Support Large Scale Adaptive Changes", in Proceedings of the International Conference on Software Maintenance (ICSM'10), Timisoara, Romania, pp. 1-10.

[Conte, Dunsmore, Shen 1986] Conte, S. D., Dunsmore, H. E., and Shen, V. Y., (1986), Software Engineering Metrics and Models. Redwood City, CA, USA, Benjamin-Cummings Publishing Co., Inc.

[Cordy, Eliot, Robertson 1990] Cordy, J. R., Eliot, N. L., and Robertson, M. G., (1990), "TuringTool: A User Interface to Aid in the Software Maintenance Task", IEEE Transactions on Software Engineering, vol. 16, no. 3, pp. 294-301.

[Danicic, Fox, Harman, Hierons, Howroyd, Laurence 2005] Danicic, S., Fox, C., Harman, M., Hierons, R., Howroyd, J., and Laurence, M. R., (2005), "Static Program Slicing Algorithms are Minimal for Free Liberal Program Schemas", Computer Journal, vol. 48, no. 6, pp. 737-748.

[De Lucia, Pompella 2002] De Lucia, A., and Pompella, E., (2002), "Effort Estimation for Corrective Software Maintenance", in Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE'02), Ischia, Italy, pp. 409-416.

[De Lucia, Pompella, Stefanucci 2005] De Lucia, A., Pompella, E., and Stefanucci, S., (2005), "Assessing effort estimation models for corrective maintenance through empirical studies", Information and Software Technology, vol. 47, no. 1, pp. 3-15.

[Dragan, Collard, Maletic 2006] Dragan, N., Collard, M. L., and Maletic, J. I., (2006), "Reverse Engineering Method Stereotypes", 22nd IEEE International Conference on Software Maintenance (ICSM'06), Philadelphia, Pennsylvania USA, September 25-27, pp. 24-34.

[Eick, Graves, Karr, Marron, Mockus 2001] Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., and Mockus, A., (2001), "Does Code Decay? Assessing the Evidence from Change Management Data", IEEE Transactions on Software Engineering, vol. 27, no. 1, pp. 1-12.

[Feng, Maletic 2006] Feng, T., and Maletic, J. I., (2006), "Using Dynamic Slicing to Analyze Change Impact on Role Type based Component Composition Model", in Proceedings of the 5th IEEE/ACIS International Conference on Computer and

Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering,Software Architecture and Reuse, pp. 103-108.

[Gallagher, Binkley 2008] Gallagher, K., and Binkley, D. W., (2008), "Program Slicing", Frontiers of Software Maintenance (FoSM '08), Beijing, China, pp. 58-67.

[Gallagher 2004] Gallagher, K. B., (2004), "Some Notes on Interprocedural Program Slicing", in Proceedings of the 4th International Workshop on Source Code Analysis and Manipulation (SCAM'04), Chicago, illinois, USA, 15-16 Sept. 2004, pp. 36-42.

[Gallagher, Lyle 1991] Gallagher, K. B., and Lyle, J. R., (1991), "Using Program Slicing in Software Maintenance", IEEE Transactions on Software Engineering, vol. 17, no., pp. 751-761.

[Gallagher, O'Brien 2001] Gallagher, K. B., and O'Brien, L., (2001), "Analyzing Programs via Decomposition Slicing: Initial Data and Observations", in Proceeding of 7th Workshop on Empirical Studies of Software Maintenance, Florence, Italy, pp. 1-13.

[Godfrey, Qiang 2000] Godfrey, M. W., and Qiang, T., (2000), "Evolution in Open Source Software: A Case Study", in Proceedings of the International Conference on Software Maintenance (ICSM'00), Los Alamitos, CA, USA, pp. 131-142.

[Godfrey, Qiang 2001] Godfrey, M. W., and Qiang, T., (2001), "Growth, Evolution, and Structural Change in Open Source Software", In Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE'01), Vienna, Austria, pp. 103-106.

[Hayes, Patel, Zhao 2004] Hayes, J. H., Patel, S. C., and Zhao, L., (2004), "A Metrics-Based Software Maintenance Effort Model", in Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04), Tampere, Finland, pp. 254-259.

[Hoffner 1995] Hoffner, T., (1995), "Evaluation and Comparison of Program Slicing Tools": D. o. C. a. I. Science, Linkping University, Technical Report LiTH-IDA-R-95-01.

[Horwitz, Reps 1992] Horwitz, S., and Reps, T., (1992), "The Use of Program Dependence Graphs in Software Engineering", in Proceedings of International Conference on Software Engineering (ICSE'92), Melbourne, Australia, pp. 392-411.

[Horwitz, Reps, Binkley 1988] Horwitz, S., Reps, T., and Binkley, D., (1988), "Interprocedural slicing using dependence graphs", SIGPLAN Not., vol. 23, no., pp. 35-46.

[Ishio, Kusumoto, Inoue 2003] Ishio, T., Kusumoto, S., and Inoue, K., (2003), "Program Slicing Tool for Effective Software Evolution Using Aspect-Oriented Technique", in Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE'03), pp. 3-12.

[ISO/IEC 1999] ISO/IEC, (1999). Software Engineering - Software Maintenance. I. S. Organization, ISO/IEC FDIS 14764:1999(E): pp.

[Israeli, Feitelson 2010] Israeli, A., and Feitelson, D. G., (2010), "The Linux kernel as a Case Study in Software Evolution", Journal of Systems and Software, vol. 83, no. 3, pp. 485-501.

[Israeli, Feitelson 2010] Israeli, A., and Feitelson, D. G., (2010), "The Linux kernel as a case study in software evolution", J. Syst. Softw., vol. 83, no., pp. 485-501.

[Jorgensen 1995] Jorgensen, M., (1995), "Experience With the Accuracy of Software Maintenance Task Effort Prediction Models", IEEE Transactions on Software Engineering, vol. 21, no. 8, August, pp. 674-681.

[Kendall, Stuart, Ord 1987] Kendall, M. G., Stuart, A., and Ord, J. K., (1987), Kendall's Advanced Theory of Statistics. New York, Oxford University Press, Inc.

[Koren 2006] Koren, O., (2006), "A Study of the Linux kernel Evolution", ACM SIGOPS Operating Systems Review, vol. 40, no. 2, pp. 110-112.

[Kumar, Horwitz 2002] Kumar, S., and Horwitz, S., (2002), "Better Slicing of Programs with Jumps and Switches", Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE'02), Springer-Verlag, London, UK, April 08 - 12, pp. 96-112

[Lehman 1980] Lehman, M. M., (1980), "Programs, Life Cycles, and Laws of Software Evolution", Proceedings of the IEEE, vol. 68, no. 9, pp. 1060-1076.

[Lehman 1996] Lehman, M. M., (1996), "Laws of Software Evolution Revisited", in Proceedings of the 5th European Workshop on Software Process Technology (EWSPT'96), pp. 108-124.

[Lehman, Perry, Ramil 1998] Lehman, M. M., Perry, D. E., and Ramil, J. F., (1998), "Implications of Evolution Metrics on Software Maintenance", in Proceedings of the International Conference on Software Maintenance (ICSM'98), pp. 208-217.

[Lehman, Ramil 2003] Lehman, M. M., and Ramil, J. F., (2003), "Software Evolution: Background, Theory, Practice", Information Processing Letters, vol. 88, no. 1-2, pp. 33-44.

[Lehman, Ramil, Perry 1998] Lehman, M. M., Ramil, J. F., and Perry, D. E., (1998), "On Evidence Supporting the FEAST Hypothesis and the Laws of Software Evolution", in Proceedings of the 5th International Symposium on Software Metrics, pp. 84-88.

[Lehman, Ramil, Wernick, Perry, Turski 1997] Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., and Turski, W. M., (1997), "Metrics and Laws of Software Evolution - The Nineties View", in Proceedings of the 4th International Symposium on Software Metrics (METRICS'97), pp. 20-32.

[Liang, Harrold 1998] Liang, D., and Harrold, M. J., (1998), "Slicing Objects Using System Dependence Graphs", in Proceedings of the International Conference on Software Maintenance (ICSM'98), Bethesda, MD, USA, pp. 358-367.

[Lindvall 1998] Lindvall, M., (1998), "Monitoring and Measuring the Change-Prediction Process at Different Granularity Levels: An Empirical Study", Software Process Improvement and Practice 4, vol., no., pp. 3-10.

[Maletic, Collard 2004] Maletic, J. I., and Collard, M. L., (2004), "Supporting Source Code Difference Analysis", in Proceedings of the International Conference on Software Maintenance (ICSM'04), Chicago, IL, USA, pp. 210-219.

[Maletic, Collard, Marcus 2002] Maletic, J. I., Collard, M. L., and Marcus, A., (2002), "Source Code Files as Structured Documents", in Proceeding of the 10th IEEE International Workshop on Program Comprehension (IWPC'02), Paris, France, June 27-29, pp. 289-292.

[McCabe 1976] McCabe, T. J., (1976), "A complexity Measure", in Proceedings of the 2nd International Conference on Software Engineering (ICSE'76), San Francisco, California, USA, pp. 308-320.

[Murphy, Notkin 1998] Murphy, G. C., and Notkin, D., (1998), "An Empirical Study of Static Call Graph Extractors", ACM Transactions on Software Engineering and Methodology, vol. 7, no. 2, pp. 158-191.

[Niessink, Vliet 1997] Niessink, F., and Vliet, H. V., (1997), "Predicting Maintenance Effort with Function Points", in Proceedings of the International Conference on Software Maintenance (ICSM'97), Bari, Italy, pp. 32-39.

[Niessink, Vliet 1998] Niessink, F., and Vliet, H. V., (1998), "Two Case Studies in Measuring Software Maintenance Effort", in Proceedings of the International Conference on Software Maintenance (ICSM'98), Bethesda, Maryland, USA, pp. 76-86.

[Ohata, Hirose, Fujii, Inoue 2001] Ohata, F., Hirose, K., Fujii, M., and Inoue, K., (2001), "A Slicing Method for Object-Oriented Programs Using Lightweight Dynamic

Information", in Proceedings of the Eighth Asia-Pacific on Software Engineering Conference (APSEC'01), Macau, China, pp. 273-283.

[Ott, Thuss 1993] Ott, L. M., and Thuss, J. J., (1993), "Slice Based Metrics for Estimating Cohesion", In Proceedings of the IEEE-CS International Metrics Symposium, pp. 71-81.

[Ottenstein, Ottenstein 1984] Ottenstein, K. J., and Ottenstein, L. M., (1984), "The Program Dependence Graph in a Software Development Environment", ACM SIGSOFT Software Engineering Notes, vol. 9, no. 3, pp. 177-184.

[Pan, Kim, Whitehead 2006] Pan, K., Kim, S., and Whitehead, J., E.,James, (2006), "Bug Classification Using Program Slicing Metrics", in Proceedings of International Workshop on Source Code Analysis and Manipulation (SCAM'06), Philadelphia, PA, USA, pp. 31-42.

[Ramil, Lehman 2000] Ramil, J. F., and Lehman, M. M., (2000), "Effort Estimation from Change Records of Evolving Software", in Proceedings of International Conference on Software Engineering (ICSE'00), Limerick, Ireland, pp. 777-787.

[Sage, Palmer 1990] Sage, A. P., and Palmer, J. D., (1990), Software systems engineering, Wiley-Interscience.

[Shepperd, Schofield, Kitchenham 1996] Shepperd, M., Schofield, C., and Kitchenham, B., (1996), "Effort Estimation Using Analogy", in Proceedings of the International Conference on Software Engineering (ICSE'96), Berlin, Germany, pp. 170-178.

[Swanson 1976] Swanson, E. B., (1976), "The dimensions of maintenance", ICSE, pp. 492-497.

[Tip 1995] Tip, F., (1995), "A Survey of Program Slicing Techniques", Journal of Programming Language, vol. 3, no. 0, pp. 121-189.

[Tonella 2003] Tonella, P., (2003), "Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis", IEEE Transactions on Software Engineering, vol. 29, no. 6, pp. 495-509.

[Venkatesh 1991] Venkatesh, G. A., (1991), "The Semantic Approach to Program Slicing", ACM SIGPLAN Notices, vol. 26, no. 6, pp. 107-119.

[Weiser 1981] Weiser, M., (1981), "Program Slicing", Proceedings of the International Conference on Software Engineering (ICSE'81), San Diego, California, USA, pp. 439-449.

[Weiser 1984] Weiser, M., (1984), "Program Slicing", IEEE Transactions on Software Engineering, vol. 10, no. 4, pp. 352-357.

[Weiser 1979] Weiser, M. D., (1979). Program slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method, University of Michigan, Ann Arbor, MI, USA, Ph.D. Dissertation Thesis.

[Wisconsin] Wisconsin, "Wisconsin Program-Slicing Project":http://www.cs.wisc.edu/wpis/html/#codesurfer.

[Xu, Qian, Zhang, Wu, Chen 2005] Xu, B., Qian, J., Zhang, X., Wu, Z., and Chen, L., (2005), "A Brief Survey of Program Slicing", ACM SIGSOFT Software Engineering Notes, vol. 30, no. 2, pp. 1-36.

[Yu 2006] Yu, L., (2006), "Indirectly Predicting the Maintenance Effort of Open-Source Software", Journal of Software Maintenance and Evolution, vol. 18, no. 5, September, pp. 311-332.

[Yu, Schach, Chen 2005] Yu, L., Schach, S. R., and Chen, K., (2005), "Measuring the Maintainability of Open-Source Software", International Symposium on Empirical Software Engineering (ISESE'05), Nossa Heads, Australia, 17-18 Nov. 2005, pp. 297-303.

[Zhang, Gupta, Gupta 2007] Zhang, X., Gupta, N., and Gupta, R., (2007), "A Study of Effectiveness of Dynamic Slicing in Locating Real Faults", Empirical Software Engineering, vol. 12, no. 2, pp. 143-160.