Linker-Based Program Extraction and Its Uses in Studying Software Evolution

Jingwei Wu [1] and Richard C. Holt [2]

School of Computer Science
University of Waterloo
Waterloo, Canada

Abstract

One of the problems of empirical studies of software evolution is the lack of an effective technique for extracting facts about very large software systems (millions of lines of code) over hundreds of versions. In this paper, we describe a linker-based approach to program extraction that is well suited for the study of large software system evolution. Our approach is particularly accurate, convenient and efficient. Its core component is a fact extractor, called *ldx*, which is a customized version of the GNU code linker *ld* and performs both code linking and fact extraction. We call a *ldx* output graph an *as-linked view* (ALV). A sequence of ALVs of successive versions of a software system can be utilized to help understand software evolution. We describe our preliminary empirical studies on the evolution of two large open source systems, Linux and PostgreSQL. We discuss several interesting results, which either validate results from earlier studies or suggest new concepts in studying software evolution.

Key words: Software Evolution, Linker-Based Program Extraction

1 Introduction

Modern software systems are extraordinarily large. For example, large commercial database systems and telephone systems have millions of lines of code and are highly complex. One of the problems of empirical studies of software evolution is the lack of an effective technique for extracting facts about very large software systems (millions of lines of code) over hundreds of versions. We argue that many current source code extractors are slow, heavyweight, and error-prone when being applied to extract facts across multiple versions for a long period. This restricts the use of program models in the study of large software system evolution.

[1] Email: j25wu@uwaterloo.ca
[2] Email: holt@plg.uwaterloo.ca

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Fig. 1. Lightweight Linker-Based Program Extraction

A sequence of program models of a large software system provide more information than simple metrics such as the number of lines of code and the number of modules. For example, function call graphs can be used to study how the interrelationships of functions evolve over time; and variable usage graphs can be utilized to assess the maintainability of a software system based on common coupling as discussed by Schach and Offutt [14]. We believe that the study of large software system evolution can benefit from program extraction techniques that are accurate, lightweight, and easy to automate.

We have developed a program extraction pipeline that is an adapted version of the program build pipeline. Figure 1 shows this pipeline. The core component of the pipeline is a linker-based fact extractor called *ldx*, which outputs a special kind of software view called *as-linked view* (ALV). An as-linked view is a graph of executable that comprises function calls and variable uses as well as object module dependencies perceived by the code linker as a program is being built. We have successfully applied this pipeline to study the evolution of many large software systems such as Linux [4] and PostgreSQL [5].

In this paper, we describe our linker-based approach to program extraction and discuss its uses in studying software evolution by examples. Our approach offers a practical means to facilitate research in the field of software evolution. The goal of this paper is not to enumerate all types of possible evolutionary analyses based on *ldx* facts, but to motivate ideas of studying software evolution using program models.

The rest of this paper is organized as follows. Section 2 presents our linker-based program extraction pipeline and discusses its benefits and limitations. Section 3 describes several empirical studies on the evolution of two large open source software systems, Linux and PostgreSQL. Section 4 reviews the related work. Section 5 concludes the paper.

2 Lightweight Linker Extraction

In the field of software evolution, an effective program extraction technique is not readily available without sacrificing completeness in favor of speed, convenience, and robustness. We made such a tradeoff in the development of our linker-based fact extractor *ldx*.

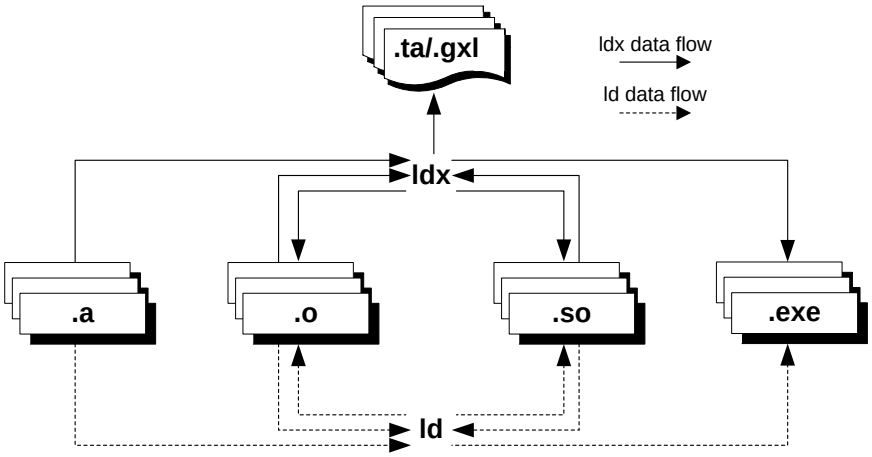


Fig. 2. Linker vs. Extractor

2.1 Linker Extractor

The *ldx* is a customized version of the GNU code linker *ld*. It can be used as a full substitute for *ld* during the extraction of as-linked views, thus making the extraction process the same as the normal program build process.

As shown in Figure 2, the extractor *ldx* operates directly on object code instead of source code. It has all the functionalities of GNU *ld*. In addition, it effectively leverages linker knowledge to produce useful relations (function calls and variable uses) as well as object module dependencies seen by the code linker during program building. The output of *ldx* is in the form of Graph Exchange Language (GXL) [2] or Tuple Attribute Language (TA) [10].

We use symbol linkage to refer to function calls and variable uses, which are most widely used relations in studying software. Extracting symbol linkage can be achieved using several different approaches such as the following:

(1) Customize an existing code linker;
(2) Customize the front end of an existing compiler;
(3) Develop a lightweight source extractor from scratch.

This list is by no means complete but gives three viable approaches. Our fact extractor *ldx* takes the first approach, and CPPX [1], CAN [7], and TkSee/SN [6] use the second approach. The idea of developing extractors through customization is to leverage knowledge and functionality of existing tools as much as possible. In the development of *ldx*, we made only small modifications to GNU *ld* to support fact extraction. The resulting fact extractor includes a abundant set of features of *ld* . For example, our extractor is available on most platforms and supports multiple binary code formats. Compared to the first two approaches, developing an extractor from scratch is more expensive and requires more maintenance.

2.2 ALV Example

We now examine a small C program called `Hello.c`. We explain how to extract its as-linked view and what information this view contains. The `Hello.c` program is shown as follows:

```
#include <stdio.h>
int main() {
  printf("Hello\n");
}
```

To extract `Hello.c` on a Linux machine, we can execute the following commands. Command 13 is used to substitute *ldx* for GNU *ld*. The compiler *gcc* will use *ldx* instead of *ld* as its code linker. Command 14 produces an object file `Hello.o`. Command 15 builds the final executable `Hello`. The file `Hello.ta` is generated as `Hello` is being created, and it stores facts produced by *ldx*.

```
[12] ls
Hello.c
[13] ln -s ‘which ldx‘ ~/bin/ld
[14] gcc -c Hello.c
[15] gcc Hello.o -o Hello
[16] ls
Hello  Hello.c  Hello.o  Hello.ta
```

Figure 3 shows a graph of the `Hello` program based on the data in `Hello.ta`. This graph is an as-linked view. The node at the top represents the executable program, which depends on one object module `Hello.o` and one dynamic library `libc.so.6`. At the bottom of the graph is a name reference `printf`, which is in the shape of diamond. The call between function `main` and function `printf` exists as a resolved cross-reference. Looking at the source code, we know that the function `main` is defined in `Hello.c`. Thus, it is contained by the object module `Hello.o`. The function `printf` is provided by the dynamic C programming library `libc.so.6`.

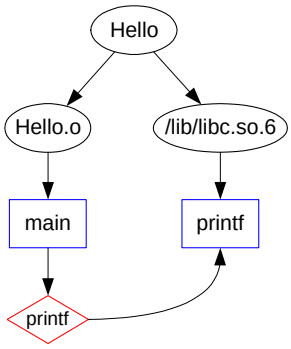


Fig. 3. ALV of Hello

2.3 Benefits and Limitations

The major benefits that a linker extractor like *ldx* offers include the following:

(i) It can deal with very large software systems at a reasonably fast speed.

We have observed that the average linking time accounts for about 1-2 percent of the total build time for a software system. The extractor *ldx* is about 10 times slower than the linker *ld*. Therefore, linker extraction only causes 10-20 percent overhead of the total build time. This performance figure was collected through extracting two large open source systems (Linux and PostgreSQL) as well as the DB2 database management system on an IBM PC with one Pentium 4 1.6GHz CPU and 1GB memory.

(ii) Its output is more than one order of magnitude smaller than CPPX output.

In comparison with CPPX, which outputs a complete abstract syntax graph (ASG) for each C/C++ source file, *ldx* only outputs facts about function calls and variable uses as well as object module dependencies. This benefits studies related to call graphs and variable usage by saving time in extracting and transforming data. The disadvantage is that *ldx* output is not useful for studies requiring details below the function level (e.g., program slicing).

(iii) It is less susceptible to failure than a general purpose source code extractor.

A binary format is less prone to change than syntax checking performed by a source extractor. A source extractor needs more frequent updates than an binary code extractor. For example, in order to build 368 releases (1996-2003) of the Linux Kernel from 2.0 to 2.5.75, we needed to download three different versions of GNU GCC: 2.91.66, 2.95.3, and 3.1. The GCC-3.0 based CPPX ran into problems when being used to extract older Linux versions before year 1999 since more strict syntax checking was performed by GCC-3.0. The *ldx* based on *ld-2.14* had no problems of extracting facts from all binary code produced by three different GCC compilers.

(iv) The intricacies of the build process become transparent during fact extraction.

The build scripts of a large software system normally include *configure*, *Makefile*(s), and shell scripts. The *configure* script probes the underlying system environment to generate appropriate structures, algorithms, and *Makefile*(s). Then, the *Makefile*(s) are used to build the entire system. Many intricacies occur in this process, and they may hinder fact extraction. By using *ldx*, the build process simply becomes the fact extraction process.

(v) It is easy to automate fact extraction over multiple versions.

The extractor *ldx* has all the functionality of GNU *ld*. By substituting *ldx* for *ld*, we can piggyback a program's build process to carry out fact extraction automatically. For example, we successfully extracted 85 monthly builds of PostgreSQL (from January 1997 to January 2004) by simply building them. The extraction took 10.5 hours on a Linux machine with Pentium 4 1.6 GHz CPU and 1 GB memory.

The decision to extract facts at link-time renders the analysis fast and straightforward. However, there are several drawbacks of the *ldx*-based extraction. First, it is not capable of producing detailed information below the function level and does not support analyses related to types and data structures. This can be complemented

by using other source code extractors, such as CPPX, when necessary. Second, it simply fails when syntax errors are present in the source code. We found that syntax errors did exist in large software systems such as Linux and PostgreSQL, but only in a negligible amount. We manually fixed those syntax errors. Third, it only considers configurations for a particular platform. If multiple platforms are involved, *ldx* has to be run on all those platforms. In this case, *ldx* probably is not a good choice for collecting program facts.

3 Empirical Studies

This section describes several empirical studies we have conducted. We show that linker-based program extraction produces useful data for studying the evolution of large software systems. We focus our discussions on two well-known large open source systems, PostgreSQL and Linux. PostgreSQL is a large Database Management System (DBMS), and Linux is a Unix-type operating system originally created by Linus Torvalds with the assistance of developers around the world.

We extracted as-linked views for 85 monthly builds of PostgreSQL (January 1997 – January 2004) and for 368 releases of the Linux kernel (2.0 – 2.5.75). Linux-2.0 was released on June 09 1996, and Linux-2.5.75 was released on July 13 2003. The 368 releases of Linux cover 86 months development. Due to the fact that the order of Linux releases by version numbers is different from their order by date, we studied Linux in two cases: (1) analyzing 324 releases ordered both by version numbers and date; (2) analyzing 79 representative (both stable and development) releases ordered by date, which were chosen on an approximately monthly basis. In the latter case, we did not choose any release from June 2000 to December 2000 since no development releases had ever been officially delivered during that time. Instead, we used the last development release in May 2000 to represent the development activities in those months.

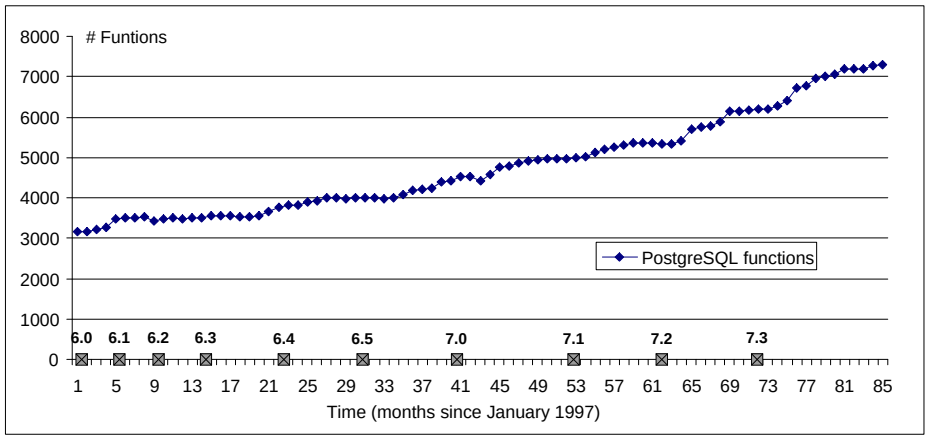
3.1 Evolution of Functions and Function Calls

Given the *ldx* output of a software system, we can easily collect various statistics of functions and function calls, for example, the number of functions, the number of function calls, and the average number of calls per function.

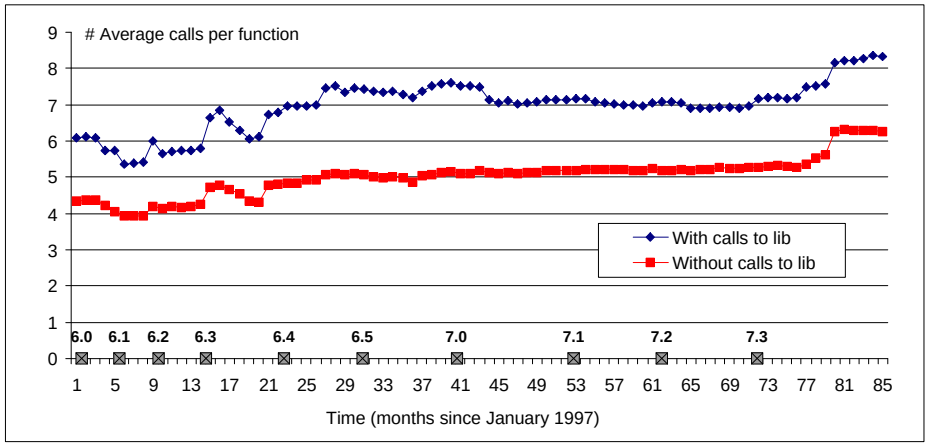
PostgreSQL

Figure 4 provides an evolutionary view of PostgreSQL. In Figure 4(a), we can see that PostgreSQL grows approximately linearly in terms of the number of functions defined. The growth rate is about 45-50 functions per month. Lehman stated that E-type systems grow continually [12]. It is interesting to see that PostgreSQL not only observes the law of *continuing growth* but also grows at a linear rate.

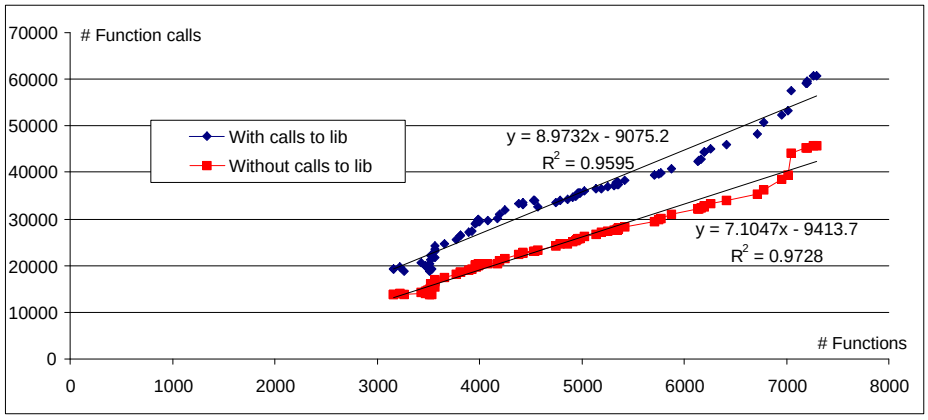
It is suspected that the average number of function calls per function stays constant as a software system evolves. To our knowledge, no explicit empirical studies have been conducted to verify this conjecture. Figure 4(b) plots the average number



(a) Evolution of the Number of Functions



(b) Evolution of the Average Number of Calls per Function



(c) Scatter Plot of Function Calls against Functions

Fig. 4. Evolution of Functions and Function Calls of PostgreSQL

of function calls per function in two cases: (1) counting all function calls including those to system library functions, and (2) ignoring calls to system library functions. Both cases show considerable instability in PostgreSQL in its early development period prior to its major release 6.5. Reading through PostgreSQL's release notes and change logs, we found that frequent restructurings were done during that period. Similar results were reported in [9]. A sudden rise in the number of calls per function appears after release 7.3. An examination of PostgreSQL ALVs revealed that this rise was caused by the rewrite of error reporting. The old `elog` function of `elog.c` was replaced by three new functions `errstart`, `errfinish` and `elog_finish`. This resulted in a widespread change across the system. The call graph of PostgreSQL is becoming larger and more complicated overall. This clearly shows Lehman's law of *growing complexity* [12].

Figure 4(c) shows a scatter plot of the number of function calls against the number of functions. Two trend lines based on linear regression are included in the figure. They explain the strong linear relationship between the number of function calls and the number of functions. The fit indexes of both, R^2, have very high values around 0.97. They indicate the high quality of the fit of the two linear regression models.

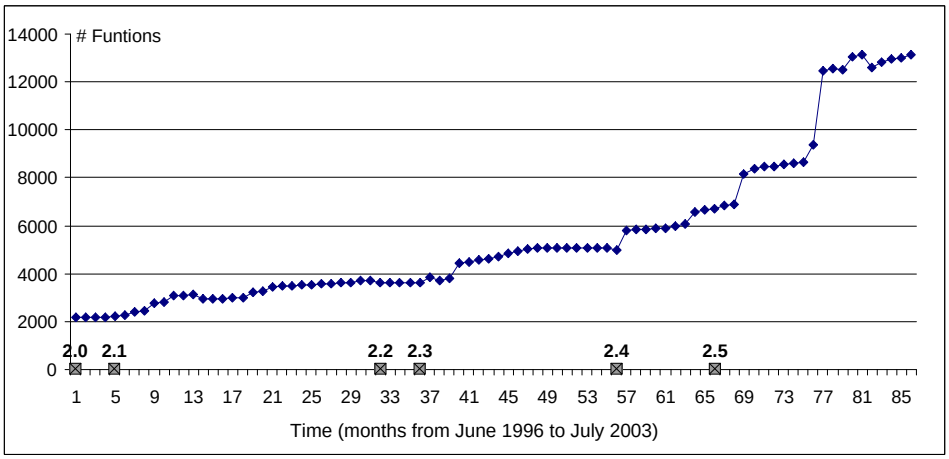
Linux Kernel

We found that the evolution of Linux was significantly different from that of PostgreSQL. This perhaps is because these two applications are targeted at different domains. One is a operating system, and the other is a database system.

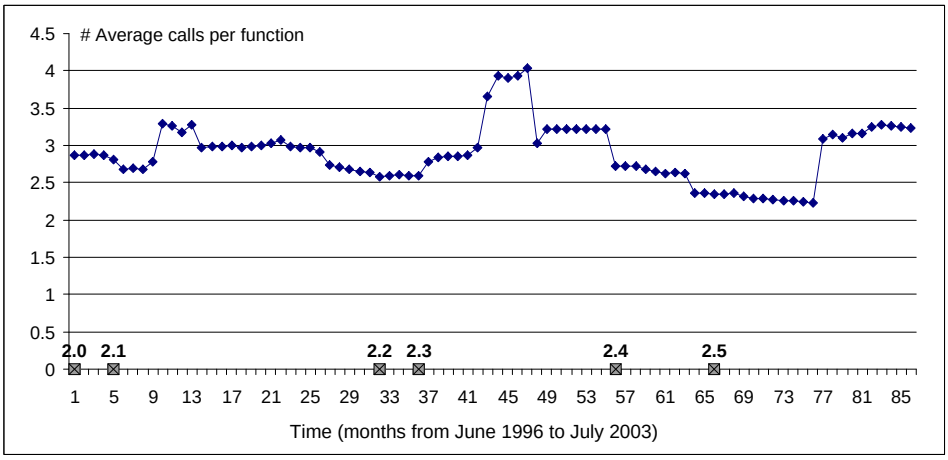
Figure 5(a) shows that the releases of Linux are growing at a super-linear rate over time. This further confirms the growth rate of Linux reported by Godfrey and Tu in year 2000 [8]. In addition, the growth of Linux shows irregularities such as jumps. All the jumps only appeared during the period of development releases that are odd-numbered. For example, from month 76 to month 78, the number of functions jumped from 8500 to 12500. This was apparently caused by integration effort.

As Linux evolves, each function makes three calls approximately. However, the average number of calls does not stay constant but oscillates. We can see from Figure 5(b) that a sudden rise occurred during each development period and then it was followed by continuous drops. Two drops are particularly interesting. The smaller drop in month 14 was related to release 2.1.44, which was extremely unstable and can cause filesystem corruption according to the online documentation of Linux [4]. The larger drop from month 47 to month 55 was related to the prereleases of Linux 2.4.0, which we suspect was intended to bring the system back to a more maintainable form. A significant amount effort was expended but no releases were delivered form June to December in 2000.

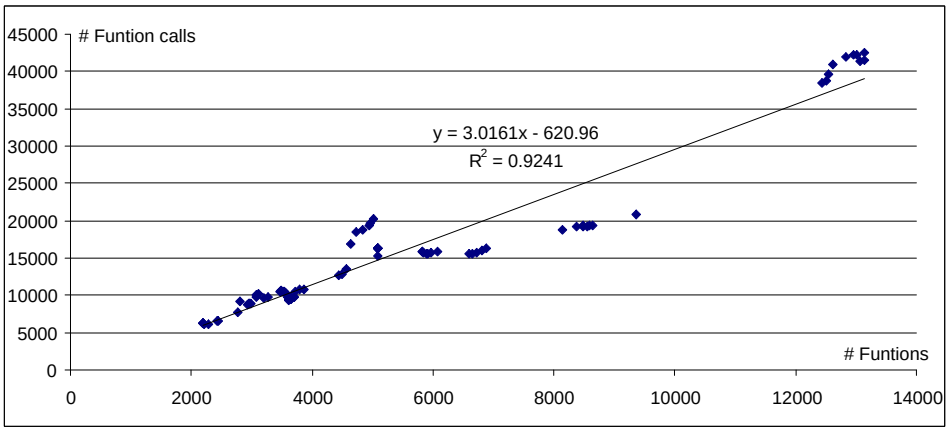
The scatter plot shown in Figure 5(c) has a linear regression trend line with a very high R^2 value. However, the linear model in this figure does not fit the data as well as the two models in Figure 4(c). There are more outliers not consistent with the linear regression model for Linux.



(a) Evolution of the Number of Functions

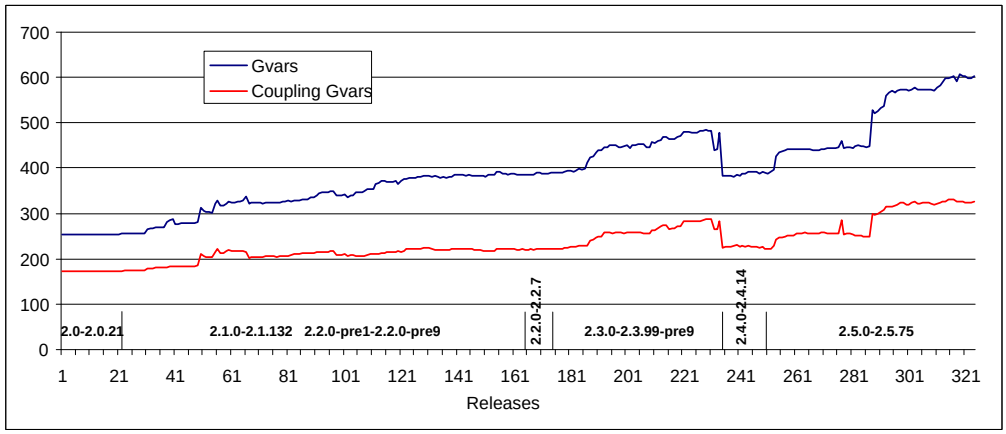


(b) Evolution of the Average Number of Calls per Function

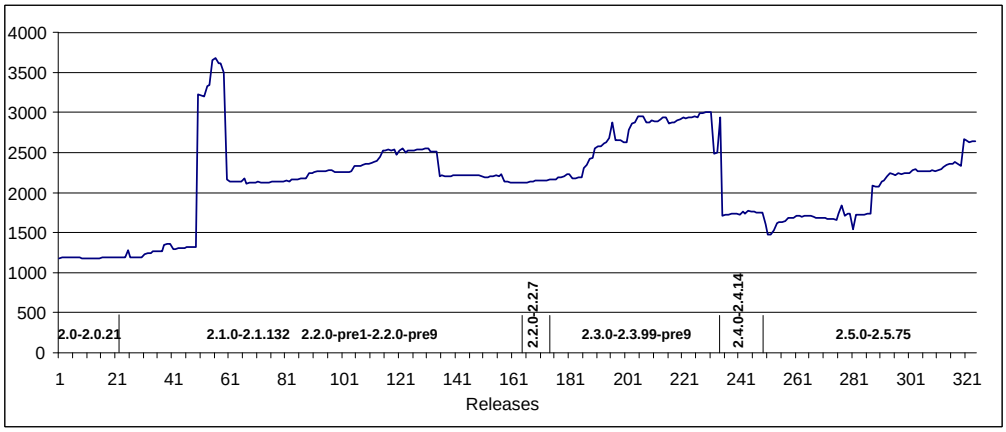


(c) Scatter Plot of Function Calls against Functions

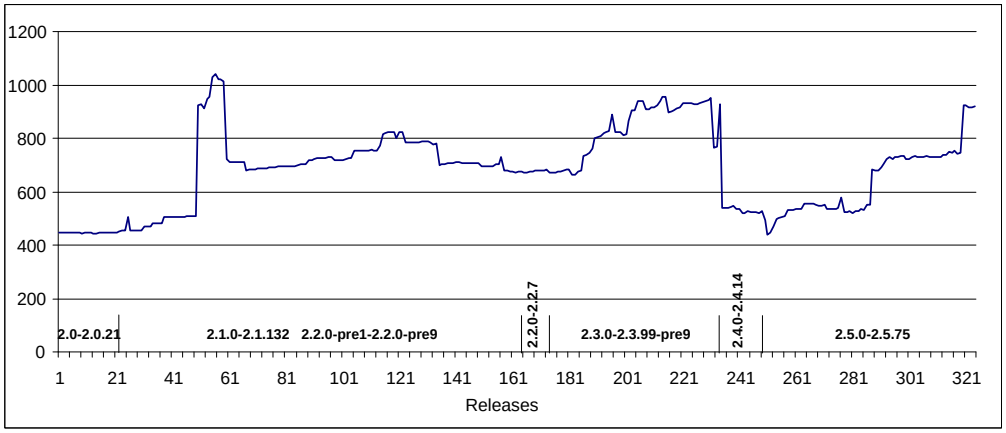
Fig. 5. Evolution of Functions and Function Calls of Linux



(a) Evolution of the Number of Global Variables



(b) Evolution of the Number of References to Global Variables



(c) Evolution of the Number of Common Couplings between Modules

Fig. 6. Evolution of References to Global Variables of Linux

3.2 Evolution of References to Global Variables

In this section, we discuss the evolution of references to global variables. We also study how coupling between modules grows over time. Due to the limited space, we only show our empirical results of the Linux kernel. The figures in this section used data from 324 kernel releases.

We only count global variables that are readable and writable, and we ignore all global constants. Figure 6(a) shows two growth lines of global variables in Linux. One is the growth of global variables that are referenced by functions, i.e., $Gvars$; the other captures the growth of global variables that cause common coupling between two different modules, i.e., $CouplingGvars$. Modules X and Y are common (global) *coupled* if they share references to the same global variable and one of them defines that global variable. The $Gvars$ and $CouplingGvars$ are both growing at a linear rate approximately and are strongly correlated.

Figure 6(b) plots the growth of the number of references to global variables. If a function makes several references to a global variable, we only count one reference to that variable. That is to say, references to global variables are lifted to the function level. The growth curve in the figure is a mixture of constant levels, sharp rises, and sudden drops. The sudden drop right before the milestone release 2.4.0 reflects that great effort was devoted to enhance the maintainability of the kernel at that time. Figure 6(c) shows the growth of common couplings between modules. It is interesting that the figures 6(b) and 6(c) share a very similar curve. This can be explained as that common coupling relationships between modules are actually lifted references to global variables at the module level.

In an earlier study on the maintainability of the Linux kernel, Schach and Offutt manually counted the number of instances of common coupling for all kernel releases before release 2.4 and they found that the exponential growth of common coupling is an inherent feature of successive versions of Linux [14]. They further claimed that the development of Linux would be slowed in the future unless the kernel is restructured with a bare minimum of common coupling. Our results suggest that restructuring efforts have reduced the instances of common coupling. We suspect that this restructuring process partially contributed to the delay of Linux releases from June 2000 to December 2000. Figure 6(c) shows a worrying trend toward more instances of common coupling as release 2.6 is coming close. This trend requires attention of Linux developers.

Lehman's law of *self regulation* states that the inherent stabilizing mechanisms of a software system yield regulations as the system evolves [12]. A superimposed ripple on the general growth patterns of a software system is a commonly observed phenomenon. The curves in Figure 6 and 6(c) strongly suggest that the development of Linux is self-regulated.

3.3 Evolutionary Change of Program Models

In order to effectively analyze software evolution, researchers need good models of change. In the following, we will describe a simple model of change based on the

concept of program models. Given two releases of a software system, X and Y, we use $M(X)$ and $M(Y)$ to represent their program models respectively. We assume that M is an algorithm for generating sets, for example, sets of lines of code and sets of function calls. We define $Addition(X, Y)$ and $Deletion(X, Y)$ to denote the differences between these two releases:

$$Addition(X, Y) = M(Y) - M(X)$$

$$Deletion(X, Y) = M(X) - M(Y)$$

We then define relative change rates of addition and deletion as:

$$Addition_R(X, Y) = \frac{|Addition(X, Y)|}{|M(X) \cup M(Y)|}$$

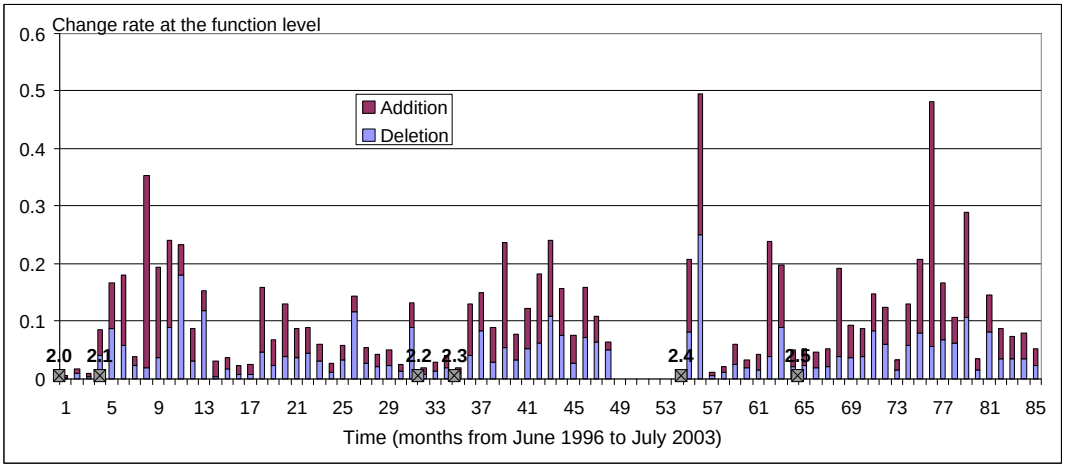
$$Deletion_R(X, Y) = \frac{|Deletion(X, Y)|}{|M(X) \cup M(Y)|}$$

A program model, in the simplest case, can be an ordered set (sequence) of lines of source code. A diff tool can be used to determine what is added or deleted for any two software releases. In the study of the change history of Linux, we used the graph of function calls and variable uses as the program model of Linux. We studied 79 Linux releases that were chosen on an approximately monthly basis.

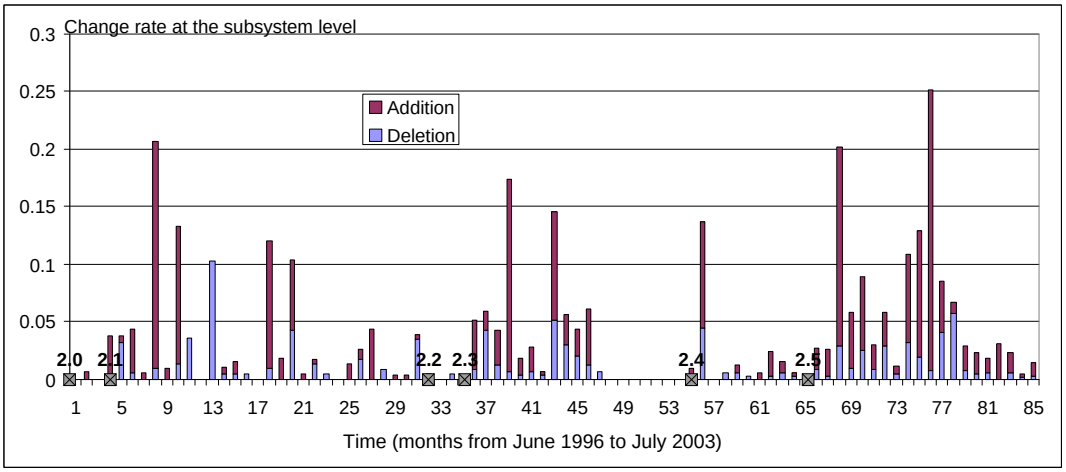
We calculated monthly change rates. We plotted them using stacked areas as shown in Figure 7. The light grey area represents deletion, and the dark grey area represents addition. Figure 7(a) displays a view of change rates at the function level for any two adjacent monthly releases, while Figure 7(b) provides a view of change rates at the subsystem level. We adopted the source code directory structure as the subsystem hierarchy. A subsystem is a directory that directly contains at least one source file. The dependencies between subsystems are calculated based on lower level function calls and variable uses. Both figures show that Linux is continually changed in order to meet new requirements. The law of *continuing change* [12] applies to Linux. However, the change was not evenly distributed. The two figures also suggest that the evolution of Linux is punctuated. Substantial changes were made during the periods of months 5-13, 40-49, and 68-78.

By Linux tradition, even-numbered kernel releases (e.g., 2.4) are stable releases for production systems, while odd numbered kernel releases (e.g., 2.5) are development releases. A notable spike appears between two stable release, Linux-2.4.0 and Linux-2.4.1. This spike is apparently caused by the integration of various features experimented during the development periods of releases 2.3.x.

Figure 7 shows that change of Linux is largely targeted at introducing new features (addition) rather than restructuring the system (deletion). For example, the spike in month 76 has a larger addition rate and a smaller deletion rate. This can be mapped to the sudden growth of functions shown in Figure 5(a). In month 26, the deletion rate is grater than the addition rate. The kernel was cleaned up by removing gratuitous function calls and variable uses during that time.



(a) Relative Change at the Function Level (Function Calls and Variable Uses)



(b) Relative Change at the Subsystem Level (Subsystem Dependencies)

Fig. 7. Evolutionary Change of Linux

4 Related Work

Our linker-based program extraction pipeline is a modified version of the SwagKit program comprehension pipeline [3,11]. Our pipeline extracts facts at the stage of code linking, while the SwagKit pipeline extracts facts at the stage of compilation. Our approach is more cost effective when being used to extract facts from hundreds of versions of a large software system.

Lehman's pioneering work on software evolution has resulted in eight laws on how E-type software systems evolve [12,13]. As shown in Section 3, as-linked views are useful for examining whether the evolution of large software systems conforms to these laws. In particular, we examined four laws: *continuing growth*, *growing complexity*, *self-regulation*, and *continuing change*. Lehman suggests using the number of modules to measure the size of a large software system, while

we used the number of functions.

Godfrey *et. al.* studied the evolution of the Linux kernel [8]. They used the number of uncommented lines of code (LOC) to measure various aspects of Linux. In contrast, our data of Linux provides a relational view of the system. We suspect that our data may be used to produce more interesting results to validate their views of Linux's evolution. Also, they studied all source files regardless of what architecture they are targeted at. We studied only compiled versions for the i386/i686 architecture.

Schach and Offutt studied the maintainability of Linux by counting the number of instances of common coupling [14]. They did the work manually. Our linker-based approach to fact extraction is easy to automate and more accurate. In addition, our studies on Linux's common coupling further extend their work and reveal that substantial restructuring has been done to reduce common couplings before release 2.4.

5 Conclusions

We presented a linker-based approach to program extraction. We customized the GNU linker *ld* into *ldx* to extract facts from binary code. The *ldx* fact extraction is accurate, convenient, and easy to automate. It is suitable for analyzing the evolution of very large software systems. We successfully applied *ldx* to extract hundreds of versions of Linux and PostgreSQL.

We conducted empirical studies on the evolution of PostgreSQL and Linux. Our studies confirmed four of Lehman's laws and showed several interesting results. For example, PostgreSQL is growing at a linear rate while Linux still at a super-linear rate. This is surprising given that Godfrey *et.al.* has shown the growth Linux is super-linear during its early development periods. The maintainability of Linux measured using common coupling gets improved before release 2.4 but shows a big growth as release 2.6 is approaching. We also presented a set-based model for studying software change based on *ldx* data.

Finally, we consider our method of linker-based program extraction provides a practical means to advance the study of large software system evolution. Now, we are preparing a web site to host all the data we have collected so far. We hope that our data can help the software evolution community to promote future research.

References

[1] "The CPPX Source Extractor," http://swag.uwaterloo.ca/∼cppx, 2002.

[2] "The Graph eXchange Language," http://www.gupro.de/GXL, 2002.

[3] "The SwagKit Software Analysis Toolkit," http://swag.uwaterloo.ca/swagkit, 2002.

[4] "The Linux Kernel Archives," http://www.kernel.org, 2003.

[5] "The PostgreSQL Database System," http://www.postgresql.org, 2003.

[6] "The TkSee/SN Source Code Extractor," http://www.site.uottawa.ca/ tcl/kbre, 2003.

[7] Ferenc, R., A. Beszedes, F. Magyar and T. Gyimothy, *A short introduction to columbus/can*, Technical Report (2001).

[8] Godfrey, M. W. and Q. Tu, *Evolution in open source software: A case study*, in: *Proceedings of the International Conference on Software Maintenance*, San Jose, California, 2000, pp. 131–142.

[9] Hassan, A. E. and R. C. Holt, *The chaos of software development*, in: *Proceedings of the International Workshop on Principles of Software Evolution*, Helsinki, Finland, 2003, pp. 84–94.

[10] Holt, R. C., *An introduction to ta: the tuple attribute language* (1996).

[11] Holt, R. C., M. W. Godfrey and A. J. Malton, *The build and comprehend pipeline*, in: *Proceedings of the Second ASERC Workshop on Software Architecture*, Banff, Alberta, Canada, 2003.

[12] Lehman, M. M. and J. F. Ramil, *Rules and tools for software evolution planning and management*, Annals of Software Engineering **11** (2001), pp. 15–44.

[13] Lehman, M. M., J. F. Ramil, P. D. Wernick, D. E. Perry and W. M. Turski, *Metrics and laws of software evolution - the nineties view*, in: *Proceedings of the 4th International Software Metrics Symposium*, Albuquerque, NM, 1997, pp. 20–32.

[14] Schach, S. R. and A. J. Offutt, *On the nonmaintainability of open source software*, in: *Proceedings of the 2nd Workshop on Open Source Software Engineering*, Orlando, Florida, 2002.