Linux Kernels as Complex Networks: A Novel Method to Study Evolution

Lei Wang Zheng Wang Chen Yang Li Zhang
School of Computer Science, Beijing University of Aeronautics and Astronautics, Beijing, China
wanglei@buaa.edu.cn, {wangzheng, yangchen}@cse.buaa.edu.cn, lily@buaa.edu.cn

Qiang Ye
Dept. of Computer Science & IT, UPEI, Charlottetown, Canada
qye@upei.ca

Abstract

In recent years, many graphs have turned out to be complex networks. This paper presents a novel method to study Linux kernel evolution - using complex networks to understand how Linux kernel modules evolve over time. After studying the node degree distribution and average path length of the call graphs corresponding to the kernel modules of 223 different versions (V1.1.0 to V2.4.35), we found that the call graphs of the file system and drivers module are scale-free small-world complex networks. In addition, both of the file system and drivers module exhibit very strong preferential attachment tendency. Finally, we proposed a generic method that could be used to find major structural changes that occur during the evolution of software systems.

1. Introduction

There are all kinds of networks around us, such as social networks (e.g. friendship networks), technological networks (e.g. the Internet), and biological networks (e.g. neural networks). Regular graphs and random graphs were used to study these networks in the past [1][2]. However, many real networks have recently been found to be neither regular graphs nor random graphs. They belong to a new type of graph, complex networks, which has completely different statistical features than those of regular and random graphs [3]. Since complex networks can be used to study many different networks, the field of complex networks has been developing at a very fast pace and has brought together researchers from various areas such as computer science, mathematics, sociology, etc. In the domain of computer science, complex networks have been used to study many different systems including the Internet and software systems.

Software evolution has been an active area over the past decades. The study of software evolution has led to many important results that enable us to understand varied software systems more deeply. For example, Lehman et al. conducted a series of empirical studies that showed that the growth of the commercial systems under investigation is usually sub-linear. That is, the growth of these systems slows down as the systems become larger and more complex [4]. However, Godfrey et al. studied the evolution of an open source software system, Linux, and found that Linux grows at a super-linear rate, which contradicts the results corresponding to the commercial systems [5][6].

Code statistics is a widely used method to study software evolution. However, this method can only be used to predict the scale of software systems in terms of code size. It cannot be used to study the internal structures of software system. Complex networks have been used to investigate individual software systems [7]. But only few researchers have taken the advantages of complex networks to study operating system evolution. This paper presents a novel method to investigate the evolution of Linux kernels in a quantitative manner - using complex networks to understand how Linux kernel modules evolve over time.

In our research, we analyzed 223 Linux kernels (from V1.1.0 to V2.4.35) and generated a call graph for each module in the kernels. By comparing the graph of the same module in different kernel versions chronologically, we had a better understanding about Linux kernel evolution. Specifically, our results showed that all kernels under investigation share the scale-free, small-world, and preferential attachment features. In addition, we found a generic method that can be used to discover major structural changes that occur during Linux kernel evolution.

The rest of this paper is organized as follows. Section 2 discusses how the call graphs of the kernel modules were generated and Section 3 describes the evolution of nodes and edges in varied kernels. Degree distribution, average path length, and preferential attachment of the call graphs are presented in Section 4, 5, and 6 respectively. Section 7 introduces the related

978-1-4244-4828-9/09/$25.00 2009 IEEE Proc. ICSM 2009, Edmonton, Canada

work briefly. The paper closes with our conclusions in Section 8.

2. Call graphs of Linux kernel modules

A call graph is a directed graph that represents the calling relationship between functions in a program. Specifically, all functions in the program are modeled as graph nodes. When a function calls another function, one edge that points to the node corresponding to the callee function is added to the graph. Ultimately, the edges in the final call graph represent all function calls that appear in the program.

In a call graph, there are two types of node degrees: out-degree and in-degree. Out-degree and in-degree represent the number of edges that start from the node and end at the node respectively. The out-degree of a node indicates the number of functions that the function corresponding to the node calls and the in-degree of a node indicates the number of functions that call the function corresponding to the node.

In our research, we use the call graphs of Linux kernel modules to study Linux kernel evolution. Specifically, we generate a call graph for each kernel module. By comparing the graph of the same module in different kernel versions chronologically, we could understand how Linux kernel evolves over time.

To generate the call graph of a kernel module, we model each function in the module as a node. For each internal function call (i.e. the caller function and the callee function are both part of the module), one edge is added. For each cross-module function call (i.e. the caller function is in the module under investigation and the callee function is in another module), a new node representing the module containing the callee is added to the call graph, then an edge pointing to the new node is inserted into the graph.

After the call graph of a module is constructed using the above method, we need to find out the largest weakly connected component (WCC) of the graph. This is because the resulting graph could be either a single graph in which all nodes are connected or a set of non-connected sub-graphs. In the former case, the graph itself is the largest WCC for the module under investigation. And the graph will be analyzed thoroughly. In the latter case, the largest WCC is one of the sub-graphs. Since, the largest WCC includes more nodes than any other sub-graph, we use it as if it is the call graph of the module for analysis purposes. Hence, in either case, the largest WCC is used as the call graph for the module.

In our research, we gathered the original data about call graphs by compiling Linux kernels using a modified version of GCC 3.4.6 [8]. The modified compiler produced the function call relationship for each C file. Once the original data was available, we wrote a customized program to analyze the data and generate the call graphs of different Linux kernel modules.

We compiled 223 different Linux kernels that fall in the range of Version 1.1.0 to Linux 2.4.35 (all of them are from http://ftp.kernel.org). Among these kernels, there are 130 development versions and 93 stable versions. For clarity, we assigned sequence numbers to these kernels according to the chronological order. Namely, the kernel that appeared earliest is No. 1 kernel and the most recent kernel is No. 223 kernel. In our research, the sequence numbers, instead of the detailed version numbers, are used to distinguish varied kernel versions.

The modified compiler reported some errors due to incompatibility when compiling some old Linux kernels. In order to compile these kernels successfully, we modified these Linux kernels slightly. However, the modification was carried out in a way that the function call relationship was not changed.

3. Evolution of nodes and edges

A Linux kernel is composed of a few different modules. In this paper, we are focused on the file system module and drivers module. We present the evolution of the nodes and edges in the call graphs of these modules in this section.

The number of nodes is a very important metric of call graphs. It reflects the scale of a graph. Figure 1 shows the growth of the node number for the file system (fs for short) and drivers module over time. The vertical axis is the number of nodes and the horizontal axis indicates the sequence numbers of the kernels under investigation. As mentioned previously, 223 Linux kernels were studied in our project. Their sequence numbers start from 1 and ends at 223.

The number of edges reflects the denseness of a call graph. Figure 2 shows the changes of the edge number for the file system and drivers module as time goes by. The vertical axis is the number of edges and the horizontal axis indicates the sequence numbers of different kernels.

Figure 1 and Figure 2 show that the number of nodes and edges for the file system and drivers modules grow rapidly over time. This is especially true for recent kernels. In addition, in terms of node and edge number, the drivers module grows faster than the file system module. Actually, it has a nearly super-linear growth, which is similar to the results presented in [5].

In Figure 1, we noted that, for the drivers module, there are two special points (marked by two circles) at which the number of nodes drops dramatically. They actually indicate that the number of nodes for the former stable releases is greater than that for the latter development releases. Specifically, these two points correspond to kernel 2.1.0 and 2.3.0 respectively. Similarly, there are two special points for the drivers module in Figure 2. We think that the reason why the special points exist is that many drivers are not required in the development releases during the kernel development process.

4. Degree distribution and scale-free networks

In this section, we analyze the node degree distributions and confirm that the call graphs of the file system and drivers module in all kernels under investigation are scale-free networks.

Degree distribution of a call graph can be used to check whether the graph is a scale-free network. Specifically, if the degree distribution obeys the power-law [9], then the corresponding graph is a scale-free network. In our research, we studied the in- and out-degree distributions of the call graphs for all 223 Linux kernels and found out that power-law fits all degree distributions in the scenarios under investigation. Namely, the call graphs of the file system and drivers module in all kernels are scale-free networks. Due to length limitation, only the degree distributions corresponding to kernel 1.2.5 are presented as an example scenario in this paper.

Figure 3(a) and 3(b) are the out- and in-degree distributions of the file system module respectively. Figure 3(c) is the call graph of the file system module generated by GraphViz [10]. Similarly, Figure 4(a) and 4(b) are the out- and in-degree distributions of the drivers module respectively. Figure 4(c) is the call graph of the drivers module generated by GraphViz.

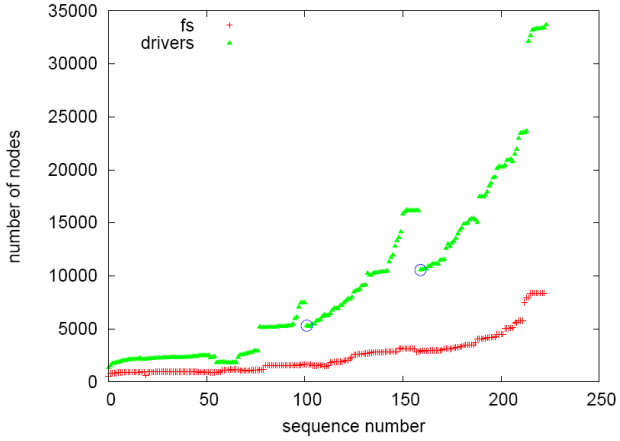

Figure 1 Number of nodes

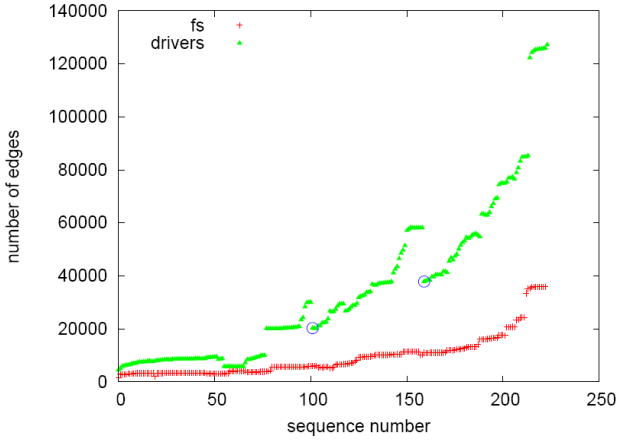

Figure 2 Number of edges

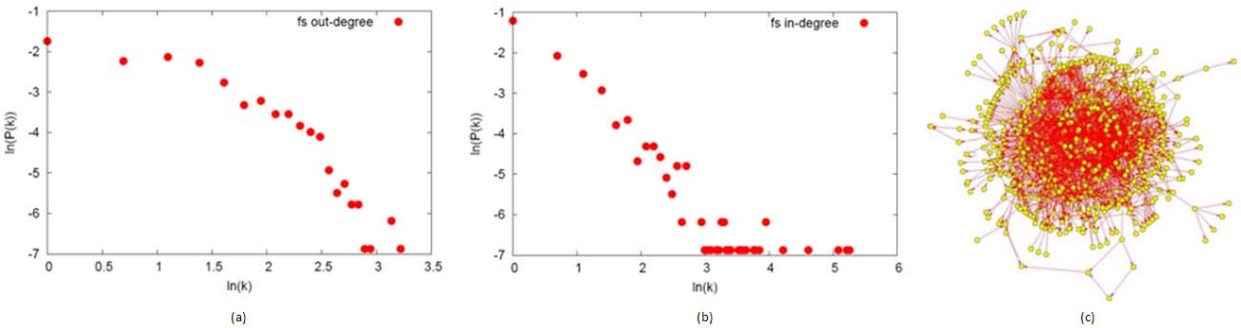

Figure 3 Out/in-degree distribution and function call graph of kernel 1.2.5 fs module

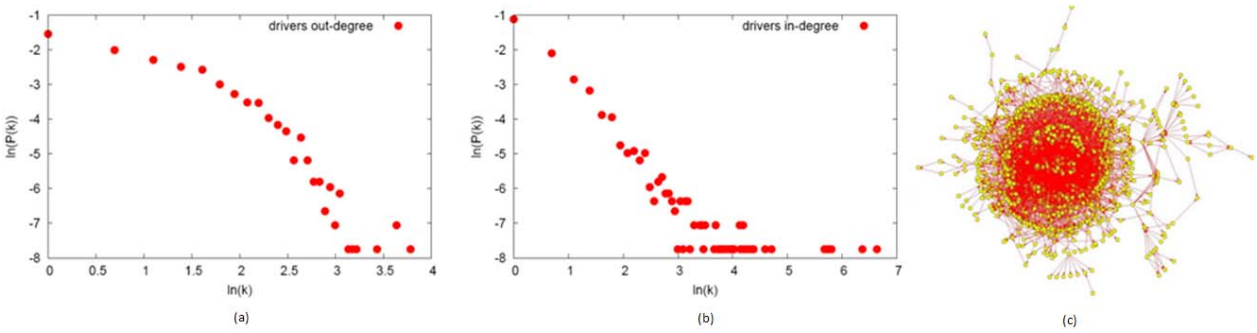

Figure 4 Out/in-degree distribution and function call graph of kernel 1.2.5 drivers module

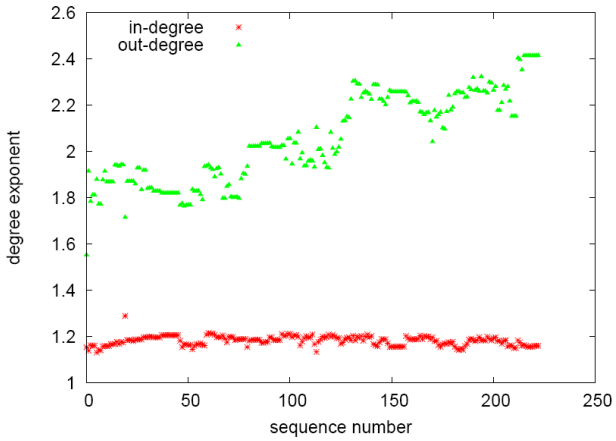

Figure 5 Degree exponent of fs module

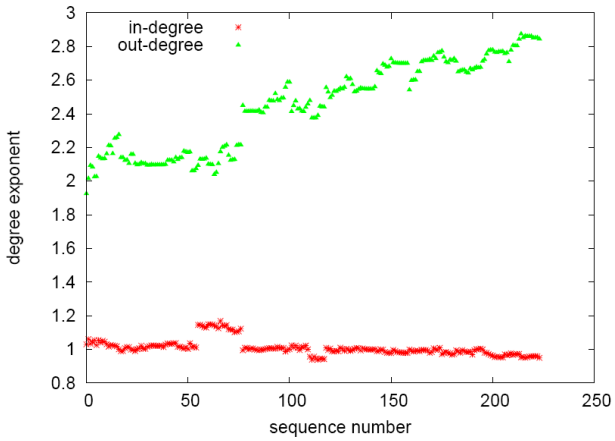

Figure 6 Degree exponent of drivers module

In Figure 3 and Figure 4, *P(k)* is the probability of finding a node with a specified degree *k* in the corresponding call graph. In Figure 3(a), 3(b), 4(a), and 4(b), the x axis is *ln(k)* and the y axis is *ln(p(k))*. A point *(x, y)* means that if we randomly select a node from the network, the probability of the node with the degree e^x is e^y. Obviously, in each of these four figures, the points have a tendency to lie on one straight line. This indicates the degree distributions should obey the power-law. Namely, if the degree distribution of a graph obeys the power-law, then the points following the distribution in the corresponding log-log plot will lie on a straight line.

We can also understand the power-law using the original node degree data. In the case that there are many nodes in a graph, if the node degree sum of few nodes account for a very large percentage of the node degree sum of all nodes in the graph, then the node degree distribution obeys the power-law. In Figure 3(c) and 4(c), a large portion of the edges involve only a few nodes, indicating that the call graphs of the file system and drivers module obey the power-law.

We also obtained some formal results about the node degree distributions. Theoretically, a straight line of points indicates that *P(k)* should be proportional to k^{-r} (*r* is a positive constant). Thus, we have:

$$P(k) = ak^{-r} \tag{1}$$

where *a* is a positive constant, then we can arrive at:

$$\ln(P(k)) = -r\ln(k) + \ln(a) \tag{2}$$

In the theory of complex networks, the constant r is called the degree exponent. It indicates the slope of the straight line. The out- and in-degree exponents of the file system module in all kernels under investigation are shown in Figure 5. Figure 6 shows the out- and in-degree exponents of the drivers module in different kernels. After formally verifying that the call graphs of the file system and drivers module of all kernels under investigation obey the power-law, we concluded that the call graphs are scale-free networks.

We noted that the out-degree exponents of the file system and drivers module are always greater than the in-degree exponent. Specifically, the average out-degree exponents of the file system and drivers module are 2.05 and 2.45 respectively. The average in-degree exponents of the file system and drivers module are 1.18 and 1.01 respectively.

5. Average path length and unusual evolutional points

Small-world networks is another much-studied class of complex networks. In this section, we analyze the average path length of the call graphs and confirm that the graphs under investigation are all small-world networks. In addition, we explain how an innovative method could be used to discover unusual evolutional points for Linux kernels.

5.1. Average path length and small-world networks

The distance between two nodes *i* and *j* in a graph is the number of edges on the shortest path between them. The average path length of a graph is defined as the average distance between any pair of nodes in the graph. Formally, it is defined using Eq. (3).

$$L = \frac{1}{\frac{1}{2}N(N+1)} \sum_{i \geq j} d_{ij} \tag{3}$$

where d_{ij} is the distance between node *i* and *j*, *N* is the number of nodes within the graph, and *L* is the average path length of the graph.

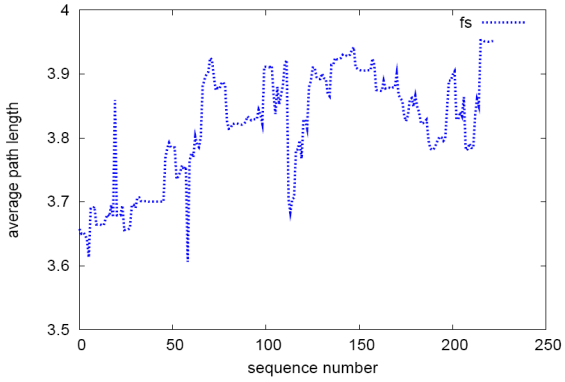

Figure 7 Average path length of fs module

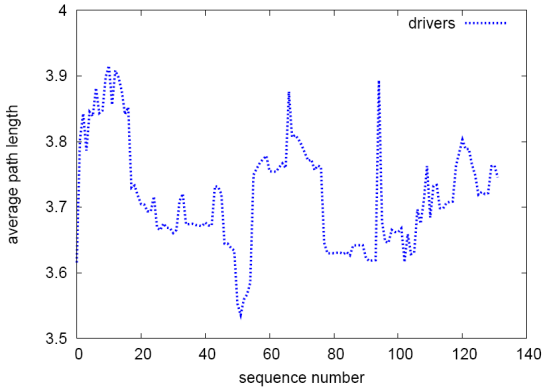

Figure 8 Average path length of drivers module

Figure 7 shows the average path length of the call graph for the file system module in the 223 kernels under investigation. Obviously, the average path length is between 3.6 and 4 during the kernel evolution. This indicates that the file system module in all 223 kernels has the small-world feature. The corresponding call graphs are small-word networks.

Figure 8 shows the evolution of the average path length for the drivers module. Because the computation complexity of the classic Dijkstra and Floy algorithm is $O(n^3)$ (*n* is the number of nodes) and the code size of the drivers module increases rapidly, it takes too long to calculate the average path length of the call graphs after kernel 2.2.0. Hence, in our research, we only obtained the results for 132 kernels that starts from kernel 1.1.0 and ends at kernel 2.1.117. We found that the largest average path length is less than 4, indicating that the call graphs corresponding to the 132 kernels are small-world networks.

5.2. Discovering unusual evolutional points

The theory of small-world model [11][12] shows that *L* scales as *ln(N)*, *N* is the number of nodes in the graph. This means that if the call graphs of Linux kernels conform to the small-world model, the average path length of a call graph should scale as *ln(N)* of the call graph. We checked whether this property holds for the call graphs of Linux kernels using the concept of the "slope" of the average path length. Given kernel version *i* and *j*, the slope of the average path length, k_{ij}, is formally defined as:

$$k_{ij} = \frac{L_j - L_i}{\ln(N_j) - \ln(N_i)} \tag{4}$$

According to *L* scales as *ln(N)*, k_{ij} should always be a positive number.

In our research, we calculated the slope corresponding to each pair of adjacent kernels. The results of the file system module are summarized in Figure 9. Overall, the slope values are positive or close to zero. However, there are a few unusual slope values that are large negative numbers. Considering that there might be some noise in real call graphs, it is better to use a threshold to filter out the noise. In our research, we used -20 as the threshold (the solid line in Figure 9). Then we found 4 unusual evolutional points corresponding to the 4 negative slope values that were not filtered out. The unusual evolutional points correspond to kernel 1.3.39, 2.1.29, 2.3.48, and 2.4.23.

The unusual evolutional points triggered us to think of an innovative method to discover structural changes of Linux kernels. The motivation was that since the proportional relationship between *L* and *ln(N)* is broken at the unusual evolutional points, some structural changes should have occurred in the kernels corresponding to the unusual evolutional points. The method is summarized as follows.

Method 1: Structural Change Discovery

1) Calculate the slope and find out the unusual evolutional points.
2) Compare the call graph of the kernel that leads to an unusual evolutional point with that of its previous kernel, and find out the edges that are added or deleted in the call graph of the later kernel.
3) Find out the nodes that are associated with the edges that were found in step 2 and count the number of appearances. This might result in a large number of nodes. However, 1 to 3 nodes usually account for more than 50% of the total number of appearances.
4) Finally, we need to analyze the Linux code of the functions corresponding to the nodes that account for more than 50% and find out the structural changes.

The analysis results of the first unusual evolutional point corresponding to kernel 1.3.39 are summarized in Table 1. In this case, the *smp_processor_id* function affects much more edges than other functions. Then we checked the source code related to this function and found out that kernel 1.3.39 introduced the support of SMP (Symmetric MultiProcessing). It was the introduction of this feature that resulted in a major structural change, leading to the first unusual evolutional point.

Table 1

Node	edge
smp_processor_id	175
sys_vhangup	2
mem_mmap	2
xiafs_unlink	1
…	…

Method 1 is a generic approach and can be used to find structural changes easily. Our analysis results show that the file system module is also affected by the SMP support at the second unusual point that corresponds to kernel 2.1.29. The unusual point corresponding to kernel 2.3.48 occurs because of the reconstruction of *hfs* file system and two functions in *vfs*, *block_read_full_page* and *block_write_full_page*. Finally, the *jfs* file system reconstruction triggers the last unusual point that corresponds to kernel 2.4.23.

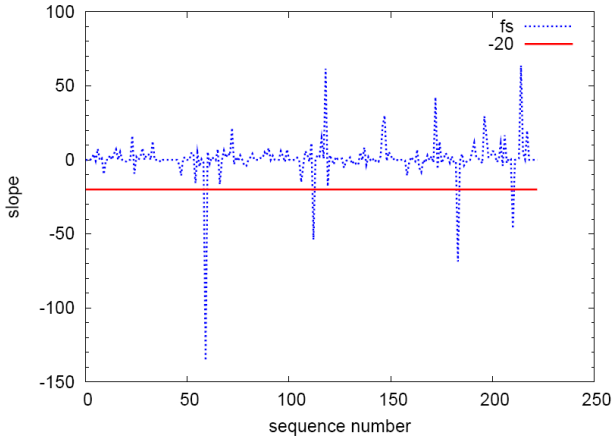

Figure 9 Evolution of the slope of fs module

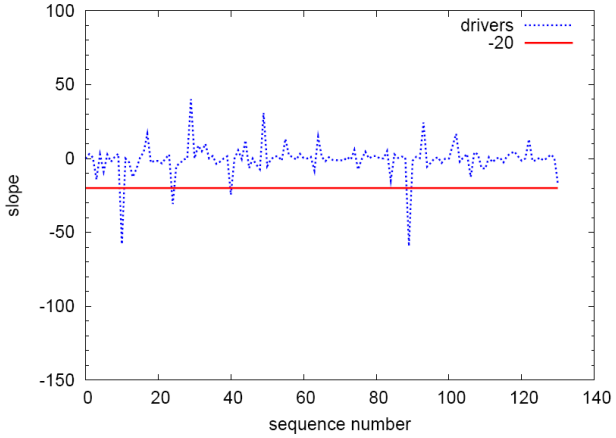

Figure 10 Evolution of the slope of drivers

Figure 10 shows the slope of the drivers module. There are 4 unusual points during the evolution of the drivers module, which correspond to kernel 1.1.64, 1.1.89, 1.2.9 and 2.0.22. After analyzing the code of the drivers module, we found that the reasons why these unusual point appear are:

- 1.1.64: the reconstruction of plip network driver
- 1.1.89: the reconstruction of scsi/u14-34f driver
- 1.2.9: the reconstruction of ide interface driver
- 2.0.22: the reconstruction of scsi/wd7000 driver

In addition, we noted that the swing of the slope of the file system module is much more serious than that of the drivers module, which shows that the drivers module is more stable than the file system module in terms of software structure although the code size of the drivers module increases faster than that of the file system module.

5.3. Clustering coefficient

Clustering, the tendency of a node's neighbors to be neighbors themselves in a graph, is a significant metric in directed graphs. We can calculate the clustering coefficient C_i of node i using Eq. (5).

$$C_i = 2E_i/(k_i(k_i-1)) \quad (5)$$

where E_i is the number of pairs of neighbors of node i, k_i is the degree of node i. The clustering coefficient C of a graph is the average clustering coefficient of the nodes in the graph.

The clustering coefficients of the file system and drivers module are presented in Figure 11 and Figure 12 respectively. Due to the same computation problem mentioned previously, we calculated the average clustering coefficient of the 132 kernels in the case of the drivers module.

Although the clustering coefficients of the file system and drivers module are both small, they tend to be constant as the size of the call graph increases. This shows the feature of clustering in the case of Linux kernels. The same phenomenon can be found in social networks.

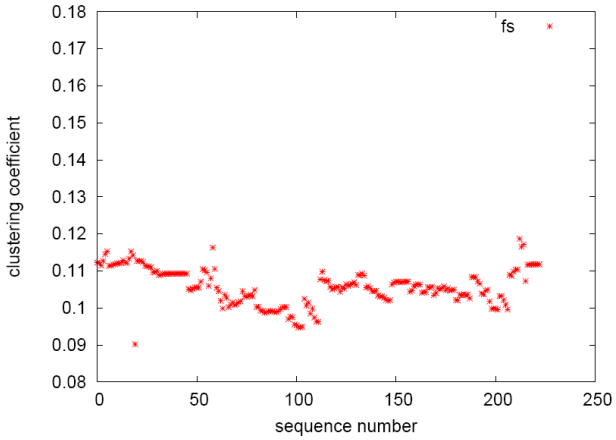

Figure 11 Clustering coefficient of fs module

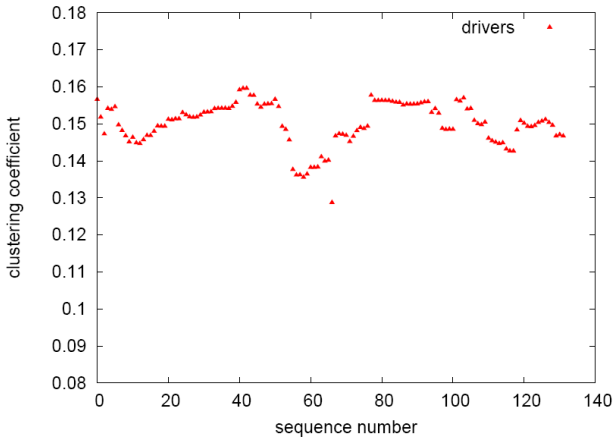

Figure 12 Clustering coefficient of drivers module

6. Preferential attachment and robust software

We mentioned previously that when unusual evolutional points appear, 1 to 3 nodes among many nodes usually account for more than 50% of the total number of appearances. This interesting phenomenon triggered us to think that, for Linux kernels, the newly added nodes might not be randomly connected to the old nodes in the previous version. Instead, it is likely that the new node tend to be attached to a few nodes in the call graph. This is one of the reasons why Linux has been very stable and robust.

6.1. Preferential attachment phenomenon

After studying how the newly added nodes are added, we found that the new nodes are very likely to be connected to the old nodes with high degrees. Namely, there exists a preferential attachment phenomenon. The details are presented as follows.

In our research, the ten nodes with the highest in-degrees and the ten nodes with highest out-degrees were first found out. Then we defined the concept of connecting probability (CP), the probability that a new node is connected to these 20 nodes. Formally, it was defined as:

$$CP = N_c/N_t \quad (6)$$

where N_c is the number of new nodes which connects to the 20 nodes with highest degree and N_t is the total number of new nodes.

Figure 13 shows the connecting probabilities of the file system module. In our research, we only considered the cases in which a new version added at least 10 new nodes to the previous version. For the scenarios in which the number of new nodes is small, preferential attachment could not be clearly observed even when there is a strong preferential attachment tendency. Finally, we studied 97 versions for the file system module. In Figure 13, the dotted line is the average connecting probability, which is about 40%. Since 20 nodes is a very small part of all the nodes in a kernel, the average of 40% indicates that there is a strong preferential attachment tendency.

Figure 14 illustrates the connecting probabilities of the drivers module. In this case, we found 162 versions that add at least 10 new nodes to the previous versions. Our results show that the average connecting probability is about 44%, which is greater than that in the case of the file system module. Because the number of nodes in the drivers module is greater than that in the file system module, the ratio of the number of the selected nodes (i.e. 20) to the total number of nodes in

the drivers module should be smaller than that in the case of the file system module. Hence, the preferential attachment tendency in the case of the drivers module is much stronger than that in the case of the file system module.

Preferential attachment can actually be used to design robust software. When there is preferential attachment in a program, the successful execution of a large portion of the function calls will depend on the correctness of the functions that are called often. These often-called functions correspond to the nodes with highest degrees in the call graph of the program, Hence, as long as the often-called functions are well designed and thoroughly checked, the execution of a large portion of the function calls will be more likely to be bug-free. Our results show that Linux kernels exhibits strong preferential attachment tendency. We believe that this is an important reason why Linux kernel stays so robust despite the rapid growth of the source code.

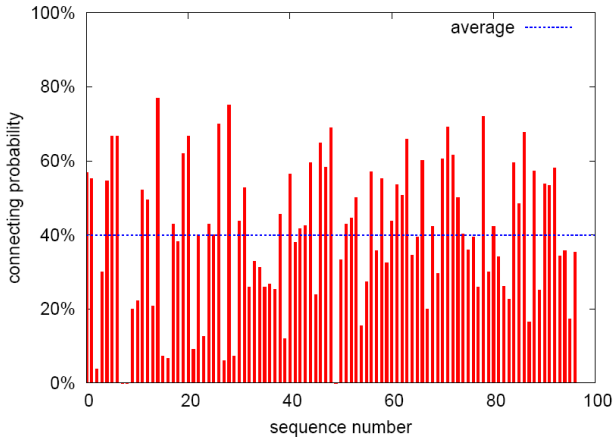

Figure 13 Connecting probability of fs module

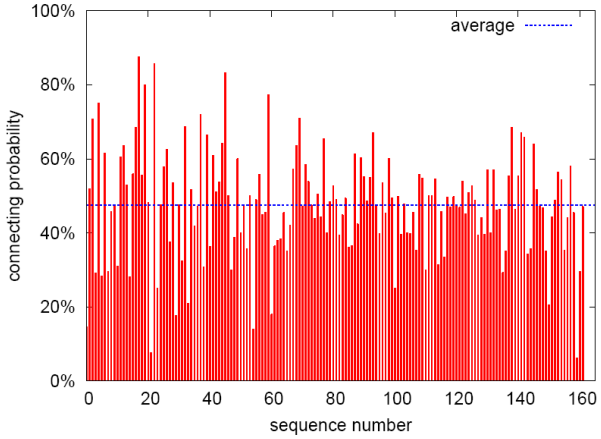

Figure 14 Connecting probability of drivers module

We also noted that the drivers module shows stronger preferential attaching tendency compared to the file system module. We think that this is one reason why the drivers module is more stable and its average path length almost remains the same despite the code size of drivers module keeps growing fast.

6.2. Degree probability

The preferential attachment is the main feature of BA scale-free model [13]. In this section, we check whether the call graphs of Linux kernels and the graph generated using the BA mode have the same preferential attachment tendency.

In our research, we defined a new concept, degree probability, to extend the probability used in the BA model.

Definition 1: The degree probability Π is the probability that a new node will be connected to the 20 nodes with highest degree, such that

$$\prod = k_{20} / \sum_i k_i \qquad (7)$$

where k_{20} is the degree sum of the 20 nodes with highest degrees and k_i is the degree of the node i. According to Definition 1, the greater the degree probability is, the larger the connecting probability is.

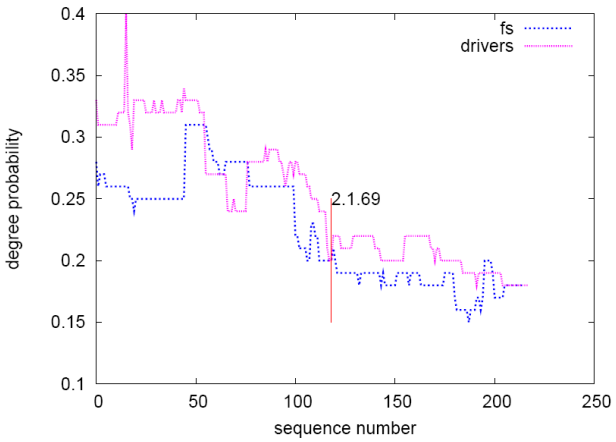

Figure 15 Degree probability

The degree probability of the file system and drivers module are presented in Figure 15. Our results show that there is a turning point, corresponding to kernel 2.1.69, that marks two different tendencies of the degree probability. Before kernel 2.1.69, the degree probability of the file system and drivers module decrease gradually, meaning the node degree becomes less and less concentrated as the number of nodes increases greatly. According to the BA model, as the degree probability decreases, the connecting probability should decrease too. However, the figures in Section 6.1 do not show a decreasing tendency of

the connecting probability at all. That shows that the call graphs of Linux kernels have a stronger preferential attachment tendency compared with the graphs generated using the BA model.

After kernel 2.1.69, the degree probabilities of the file system and drivers module start to decrease much slower. Actually, they have a tendency to remain above 0.15. This means that preferential attachment phenomenon exists in the kernels after version 2.1.69 and is likely to continue in the future versions.

7. Related work

In this section, we present the related work on software evolution and the application of complex networks to software systems.

7.1. Software evolution

Lehman and Ramil showed the clear linear growth of the OS/360 operating system over a number of releases [14]. Further, Lehman and Ramil established clear commonalities in the evolution of software through a series of empirical studies in the Feedback, Evolution And Software Technology (FEAST) [15] projects that showed the growth of these commercial systems is usually sub-linear, slowing down as the system becomes larger and more complex [16].

Software Architecture Group (SWAG) at the University of Waterloo is a well known group that is focused on the evolution of software systems. Godfrey et al. studied the evolution of Linux kernel using the method of code statistics. In their research, they expected that Linux grows slower as it becomes larger and more complex. However, in the end, they found evidence suggesting that Linux grows at a super-linear rate, which contradicts the empirical studies of the commercial systems [5].

Gall et al. implemented a release history database. In addition, they combined CVS and Bugzilla together into a whole system [17]. They concluded that all the modifications that the same developer submits within a couple of seconds present a logical coupling [18]. Using these logical couplings, the classes and modules that are modified together could be found out. The analysis of this kind of relationship revealed two types of structures, Man-in-the-middle and Data-container [19], which hammers the evolution of software. Finally, they introduced the method that can be used to change the architecture of these two types of structures.

Modeling software architecture is also a very useful method to study the structure evolution of software. The modeling language defines some elements, such as component, interface, and connector, with which the architecture of software can be depicted on different levels [20][21][22][23]. However, it is hard to use the method to analyze open source software because the design documents are not available.

7.2. Complex networks and software

Myers [7] studied the software collaboration graphs within several open source software systems, including Linux, and found that they are scale-free, small-world networks. He introduced a simple model of software evolution based on refactoring processes.

Jenkins et al. analyzed the stability of the structure of several software architecture graphs using the theory of complex networks and proposed a simple scheme to quantify the instability of the structure of software on the instability metric of Martin [24].

LaBelle et al. analyzed the complex networks of two open-source software repositories at the inter-package level and found that the package dependency networks share the small-world and near power-law distribution properties [25].

Valverde et al. studied software architecture using complex networks [26]. And they also analyzed the structure of open source communities by email exchanges, which is a complex weighted network with scaling laws at different levels [27].

8. Conclusions

In our research, we used complex networks to study the evolution of Linux kernel modules. Specifically, we generated the call graphs of 223 Linux kernels (Version 1.1.0 to Version 2.4.35) and investigated several aspects of these call graphs, such as degree distribution, average path length, preferential attachment, and clustering coefficient.

Our results show that the call graphs of the file system and driver module in all Linux kernels under investigation are scale-free small-world complex networks. In addition, both of the file system and drivers module exhibit very strong preferential attachment property. We believe this is why Linux kernels have been very robust despite the code size has increased rapidly. Finally, we proposed a generic method that could be used to find major structural changes that occur during the evolution of software systems.

There are a few areas that could be explored in the future research. Computing the average path length and clustering coefficient effectively is still a problem in the area of complex networks [28]. Currently, when the number of nodes is relatively large, it takes too long to complete the computation. In addition, we think that the relationship of the data structures within Linux

kernels should be a complex network. One of our research plans is to modify GCC in order to retrieve the relationship data and use complex networks to study the data structures.

Acknowledgments

This work was supported by Supported by National High Technology Research and Development Program of China (863)(No. 2007AA01A127) and Major State Basic Research Development Program of China (973) (No. 2007CB310803). Thanks for the anonymous reviewers' valuable comments.

References

[1] Chartrand G., *Introductory Graph Theory*, Published by Courier Dover Publications, 1985.

[2] P. Erdős, A. Rényi, "On Random Graphs", Publ. Math. Debrecen 6, 1959, pp. 290-297.

[3] M. E. J. Newman, "The structure and function of complex networks", *SIAM Review*, 2003, pp. 167-256.

[4] M. M. Lehman, "Laws of Software Evolution Revisited", *Lecture Notes in Computer Science*. 1996, pp. 108-124

[5] M. Godfrey, Q. Tu, "Evolution in Open Source Software: A Case Study", In Proceedings of the International Conference on Software Maintenance, 2000, pp. 131-142.

[6] M. Godfrey, Q. Tu. "Growth, evolution, and structural change in open source software". In Proceedings of the 4th International Workshop on Principles of Software Evolution, 2001, pp. 103-106.

[7] C. R. Myers. "Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs", *Physical review*, vol. 68 (2), no. 4, 2003, pp. 046116.1-046116.15.

[8] GCC 3.4 Release Series. http://gcc.gnu.org/gcc-3.4/

[9] M. E. J. Newman, "Power laws, Pareto distribution and Zipf's law", *Contemporary Physics* 46, 2005, pp. 323–351.

[10] Graphviz. http://www.graphviz.org/

[11] D. J. Watts, S. H. Strogatz, "Collective dynamics of 'small-world' networks", *Nature*, Volume 393, 1998, pp.440-442.

[12] M. E. Newman, D. J. Watts, "Renormalization group analysis of the small-world network model", *Physics Letters* A, Vol. 263, No. 4-6. 1999, pp. 341-346.

[13] AL Barabási, Reka Albert. "Emergence of scaling in random networks", *Science*(286), 1999, pp. 509-512.

[14] M. M. Lehman, J. F. Ramil, "Evolution in Software and Related Areas", In Proceedings of International Workshop on Principles of Software Evolution, 2001 pp. 1-16.

[15] Evolution And Software Technology (FEAST) web site. http://www.doc.ic.ac.uk/~mml/feast

[16] M. M. Lehman, D. E. Perry, J. F. Ramil, "Implications of evolution metrics on software maintenance", In Proceedings. International Conference on Software Maintenance, 1998, pp. 208-217.

[17] M. Fischer, M. Pinzger, H. Gall, "Populating a Release History Database from Version Control and Bug Tracking Systems" In Proceedings of the International Conference on Software Maintenance, 2003, pp. 23- 32.

[18] B. Fluri, H. Gall, M. Pinzger, "Fine-Grained Analysis of Change Couplings". In Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation, 2005, pp. 66-74.

[19] J. Ratzinger, M. Fischer, H. Gall. "Improving evolvability through refactoring", In Proceedings of the 2005 international workshop on Mining software repositories, 2005, pp. 1-5.

[20] N. Sadou, D. Tamzalit, M. Oussalah, "A unified approach for software architecture evolution at different abstraction levels", In Proceedings of the Eighth International Workshop on Principles of Software Evolution, 2005, pp. 65-70.

[21] C. H. Lung, S. Bot, K. Kalaichelvan, and R. Kazman, "An approach to software architecture analysis for evolution and reusability", In Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research, 1997, pp. 15-25.

[22] D. Garlan, B. Schmerl, "Architecture-driven modelling and analysis". In Proceedings of the Australian workshop on Safety critical systems and software, 2007, pp. 3-17.

[23] V. Issarny, T. Saridakis, A. Zarras, "Multi-view description of software architectures", In Proceedings of the third international workshop on Software architecture, 1998, pp. 81-84.

[24] S.Jenkins and S.R. Kirk, "Software architecture graphs as complex networks: A novel partitioning scheme to measure stability and evolution", *Information Sciences*, 177(12), 2007, pp. 2587-2601.

[25] N. Labelle and E. Wallingford, "Inter-Package Dependency Networks in Open-Source Software", CoRR cs.SE/0411096, 2004.

[26] S. Valverde and R. V. Solé, "Hierarchical small worlds in software architecture", *Dynamics of Continuous Discrete and Impulsive Systems*: Series B, 2007, pp. 1-11.

[27] S. Valverde and R. V. Solé, "Self-Organization versus Hierarchy in Open Source Social Networks", *Physical Review E* 76, 32767, 2007.

[28] D. Shi., L. Liu, "Evolving networks-models, measurements and methodologies", In Complex network, Shanghai Science and Technology Publication.