

Power Laws in Software

PANAGIOTIS LOURIDAS, DIOMIDIS SPINELLIS, and VASILEIOS VLACHOS
Athens University of Economics and Business

2

A single statistical framework, comprising power law distributions and scale-free networks, seems to fit a wide variety of phenomena. There is evidence that power laws appear in software at the class and function level. We show that distributions with long, fat tails in software are much more pervasive than previously established, appearing at various levels of abstraction, in diverse systems and languages. The implications of this phenomenon cover various aspects of software engineering research and practice.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.8 [**Software Engineering**]: Metrics—*Product metrics*; D.2.9 [**Software Engineering**]: Management—*Software quality assurance (SQA)*; D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries; reuse models*; G.3 [**Probability and Statistics**]:

General Terms: Measurement, Languages

Additional Key Words and Phrases: Scale-free networks, power laws

ACM Reference Format:

Louridas, P., Spinellis, D., and Vlachos, V. 2008. Power laws in software. *ACM Trans. Softw. Engin. Method.* 18, 1, Article 2 (September 2008), 26 pages. DOI = 10.1145/1391984.1391986 <http://doi.acm.org/10.1145/1391984.1391986>

1. INTRODUCTION

Phenomena in a campaign covering diverse areas, from Internet topology to human acquaintance networks, can be described by the same statistical framework. It appears that so-called power law distributions are found anywhere that researchers care to look. This observation has escaped the confines of academia and has been popularized by among others, a leading scientist in the field [Barabási 2002; Barabási and Bonabeau 2003].

In this context, some aspects of software have been studied and found to obey the same laws. When software is seen as a network of interconnected and

This work was partially funded by the European Community's Sixth Framework Programme under the contract IST-2005-033331 "Software Quality Observatory for Open Source Software (SQO-OSS)."

Authors' addresses: Patission 76, GR-104 34 Athens, Greece; email: louridas@aueb.gr; dds@aueb.gr; vbill@aueb.gr.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org. © 2008 ACM 1049-331X/2008/09-ART2 \$5.00 DOI 10.1145/1391984.1391986 <http://doi.acm.org/10.1145/1391984.1391986>

ACM Transactions on Software Engineering and Methodology, Vol. 18, No. 1, Article 2, Pub. date: September 2008.

cooperating components, the network is of a specific kind, a scale-free network, with important characteristics. The idea that software forms a network is not new: twenty years ago Knuth asserted that “a complex piece of software is, indeed, best regarded as a *web* that has been delicately pieced together from simple materials” [Knuth 1984b]; what is important is the recognition of similarities between this sort of web and corresponding webs in other fields.

In this work, we examine power laws in software from a software engineering point of view; if software forms structures with predictable characteristics, we may be able to explain earlier empirical findings in software engineering research, exploit the structures in current practice, and also provide directions for further research.

The remainder of this article is structured as follows. We introduce power laws in Section 2. There is already evidence of power laws in software at a microscopic level, for example, at the level of method calls or class references [Wheeldon and Counsell 2003; Baxter et al. 2006]. Our main contribution, in Section 3, is to show that the phenomenon is far more pervasive than has been established. We find that software follows power laws not only at the level of fine-grained constructs, but at various levels of abstraction, from the microscopic to the macroscopic. Moreover, we find that power laws emerge both in products developed and released by software organizations, and as a result of ad hoc contributions from around the world; and, predictably, the pattern remains the same when we examine both source code and compiled code.

We present some implications of our findings for software engineering in Section 4. The implications are two-fold: existing observations parallel our findings, while our findings may guide software development research and practice. Details on the research methodology and the tools used can be found in the Appendix.

2. POWER LAWS

The notion of power laws as a descriptive device has been around for more than a century [Mitzenmacher 2004]. During this period, power laws have cropped up in different guises in various contexts. The Italian economist Vilfredo Pareto described a power law distribution (although nobody called them that way back then) in the 19th century [Pareto 1897]. In the early 20th century, G. Udney Yule also came upon power laws in his study of the creation of biological species [Yule 1925]. A bit later, Harvard linguist George Kingsley Zipf “observed that the n th most common word in natural language texts seems to occur with a frequency approximately proportional to $1/n$ ” [Knuth 1998; Zipf 1935, 1949]. Yule’s work was visited by Herbert Simon [Simon 1955] (who named the related distribution after Yule), while Benoit Mandelbrot came upon power laws in a theory of word frequencies in 1951 [Mandelbrot 1951a, 1951b]; he went on to show that they occur as a result of an optimization process [Mandelbrot 1953].

Mathematically, a power law is a probability distribution function in which the probability that a random variable takes a value is proportional to a negative power of that value:

$$P(X = x) \propto cx^{-k} \quad \text{where } c > 0, k > 0. \quad (1)$$

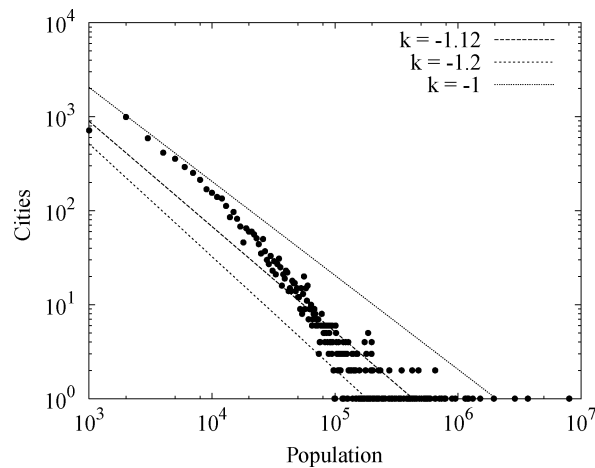


Fig. 1. US city sizes, 2000 US Census.

Sometimes power laws are defined by an equivalent, complementary cumulative definition, where the probability that a random variable takes at least a value is proportional to a negative power of that value, that is:

$$P(X \geq x) \propto cx^{-k'}. \quad (2)$$

The two definitions, however, are essentially equivalent, with $k = k' + 1$.

A well known example of a power law is the distribution of city sizes. If we plot the histogram of the distribution of city populations, we see that while the bulk of the distribution occurs for fairly small sizes (most US cities have small populations), there is a small number of cities with a population much higher than the typical value [Newman 2005]. Figure 1 is such a plot, in logarithmic scale, where a power law traces a straight line. If we fit the data to a line, we find that the slope of the line is $k = -1.12$ ($r^2 = 0.82$; we show other values of k for illustration purposes).

Power laws also appear in a slightly different form, as *rank-size* distributions, where n items are ranked according to their sizes, the probability p_r of the r th sized item being inversely proportional to its rank. Examples of such distributions are the Zipf law [Knuth 1998, p. 400] and the Zipf-Mandelbrot law [Mandelbrot 1983, p. 344]. In general, distributions such as $p_r \propto cr^{-k}$ can be converted to power laws [Adamic 2000].

In everyday life, we encounter power laws in the Pareto principle, or 80-20 rule (taking its name from the power law Pareto distribution). According to this principle, 20 percent of some input is generally responsible for 80 percent of some result. The observation that 20 percent of a group's effort needs to be expended for accomplishing 80 percent of a project, or that "80 percent of the contribution comes from 20 percent of the contributors" [Boehm 1987] are popular instances of the rule.

The 80-20 rule can, in fact, be derived analytically. A power law of the form (2) can be converted to a rank-size distribution [Adamic and Huberman 2002] where the probabilities are ranked as p_1, p_2, \dots, p_n (p_1 being the most frequent

occurrence, p_2 the second most frequent, and so on). That distribution is approximated by another rank-size distribution [Knuth 1998, p. 400] for which we have, for the first a percent of the ranked probabilities,

$$\frac{p_1 + p_2 + \cdots + p_{an}}{p_1 + p_2 + \cdots + p_n} \approx b \quad \text{for all } n \quad (3)$$

where

$$\theta = \frac{\log b}{\log a} = 1 - \frac{1}{k'}. \quad (4)$$

For $b = 0.80$ and $a = 0.20$ we arrive at the Pareto principle: the richest 20 percent of the items have 80 percent of the resources. As this holds for any n , the principle applies in a fractal, self-similar fashion to the top 20 percent, so that the topmost 4 percent of the items have 64 percent of the resources, and so on. Note that there is no need for $b = 0.80$ and $a = 0.20$; nor is there a need for $a + b = 1$.

The rise of power laws to ubiquity status came when it was observed that networks seem to exhibit such laws. A large variety of structures, ranging from linguistic word nets to biological metabolic pathways, can be modeled by networks of nodes that communicate via links. In many of these structures, the distribution of links to nodes follows a power law distribution. Such networks follow fat-tailed distributions that incarnate rules of the “winner takes all” type. Nodes that are heavily linked, *hubs*, are not improbable and they do occur.

If we take the web as an example of a power law distribution, with sites as vertices and links as edges, the resulting network exhibits a structure where, like US city sizes, most sites are linked from a small number of sites; there are some sites, however, that have a much larger number of incoming links [Albert et al. 1999; Huberman and Adamic 1999]. Such hubs of connectivity are characteristic of scale-free networks. Moreover, the ratio of hubs to the total number of nodes in the network remains constant, irrespective of the network size; networks that have this structure are therefore called *scale-free* networks.

Scale-free networks seem to appear everywhere, from the Internet [Faloutsos et al. 1999] to actor collaborations [Barabási and Albert 1999]; a guide to literature, as of 2003, runs to four pages of author-date citations [Dorogovtsev and Mendes 2003, pp. 237–240]. Since a common law seems to describe all kinds of different phenomena, the research agenda turned into seeking an explanation for the ubiquity of power laws, and, in fact, various models have been proposed. These can be divided into two groups [Mitzenmacher 2004]: treating power laws as the result of an optimization process, or as a result of a growth model, the most popular of which is the *preferential attachment* model [Barabási et al. 1999; Barabási and Albert 1999].

3. SOFTWARE POWER LAWS

We studied the existence of scale-free networks in modules comprising software systems. We chose modules of varying size and functionality, ranging from simple Java classes to systems using self-contained libraries written in C, Perl, and Ruby. For our purposes, the links connecting the modules are given by

Table I. Software Power Laws.

Dataset	size	k		r^2
		in/out	in/out	in/out
J2SE SDK	13,055	2.09/3.12		.99/.86
Eclipse	22,001	2.02/3.15		.99/.87
OpenOffice	3,019	1.93/2.87		.99/.94
BEA WebLogic	80,095	2.01/3.52		.99/.86
CPAN packages	27,895	1.93/3.70		.98/.95
Linux libraries	4,047	1.68/2.56		.92/.62
FreeBSD libraries	2,682	1.68/2.56		.91/.58
MS-Windows binaries	1,355	1.66/3.14		.98/.76
FreeBSD ports, libraries deps	5,104	1.75/2.97		.94/.76
FreeBSD ports, build deps	8,494	1.82/3.50		.99/.98
FreeBSD ports, runtime deps	7,816	1.96/3.18		.99/.99
T _E X	1,364	2.00/2.84		.91/.85
METAFONT	1,189	1.94/2.85		.96/.85
Ruby	603	2.62/3.14		.97/.95
The errors of T _E X	1,229	3.36		.94
Linux system calls (242)	3,908	1.40		.89
Linux C libraries functions (135)	3,908	1.37		.84
FreeBSD system calls (295)	3,103	1.59		.81
FreeBSD C libraries functions (135)	3,103	1.22		.80

their dependencies. For two modules, A and B , we add a directed link from B to A when B depends on A . This produces a directed graph. We explore the structure of both the incoming links and the outgoing links.

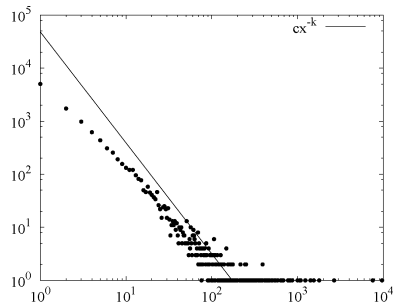
Note that measuring fan-in and fan-out is not new, and has been used as a measure for procedural complexity [Henry and Kafura 1981]. Here we are not interested in measuring complexity, but in seeing whether incoming and outgoing links in different levels of abstraction show similar patterns.

A summary of our findings is shown in Table I. In each row we list the number of nodes, the exponent for the incoming links and outgoing links, where applicable, and the corresponding correlation coefficient.

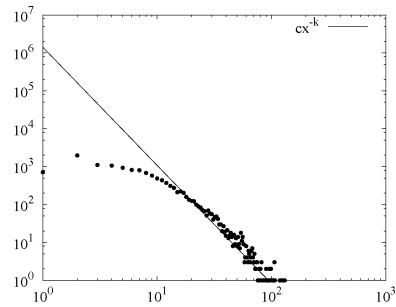
3.1 Java

In our Java study, network nodes are classes and the links represent the use of a class by another class. We say that class B uses class or interface A when the code of B references A . This includes class extension and interface implementation, as well as member variables of type A , the presence of type A in method signatures, and local (method) variables of type A . This list is not exhaustive: class B may use A during program execution by using run-time reflection. This however cannot be determined by statically examining the program code.

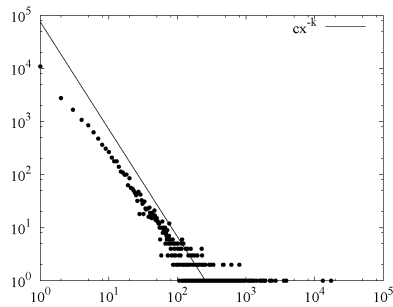
We studied the Java J2SE Software Development Kit (SDK) version 1.4.2.05, two large Java applications, the Eclipse platform, version 3.1, the OpenOffice suite, and a big Java middleware product, the BEA WebLogic platform, version 8.1 (Figure 2, one row per dataset). The dependencies were extracted by trawling through the compiled bytecode of Java classes. A class's bytecode contains all necessary information pertaining to the use of other classes [Lindholm and Yellin 1999], as defined above, and examining compiled code is much faster



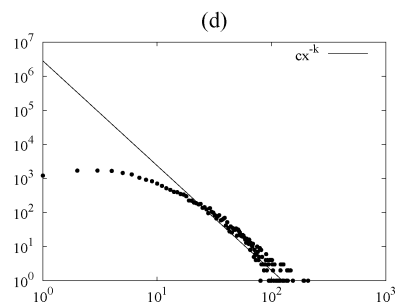
(a) J2SDK 1.4.2_05 in ($k = 2.09$)



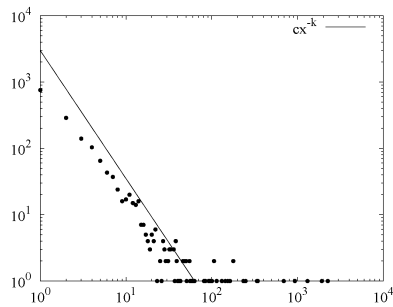
(b) J2SDK 1.4.2_05 out ($k = 3.12$)



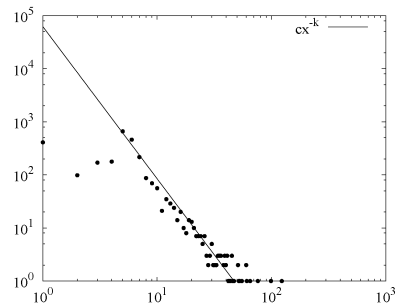
(c) Eclipse 3.1 in ($k = 2.02$)



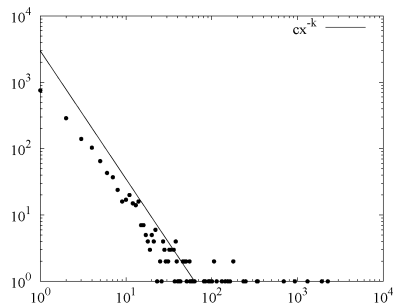
(d) Eclipse 3.1 out ($k = 3.15$)



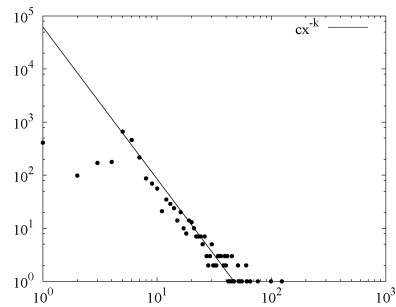
(e) OpenOffice in ($k = 1.93$)



(f) OpenOffice out ($k = 2.87$)



(g) WebLogic 8.1 in ($k = 2.01$)



(h) WebLogic 8.1 out ($k = 3.52$)

Fig. 2. Java dependencies.

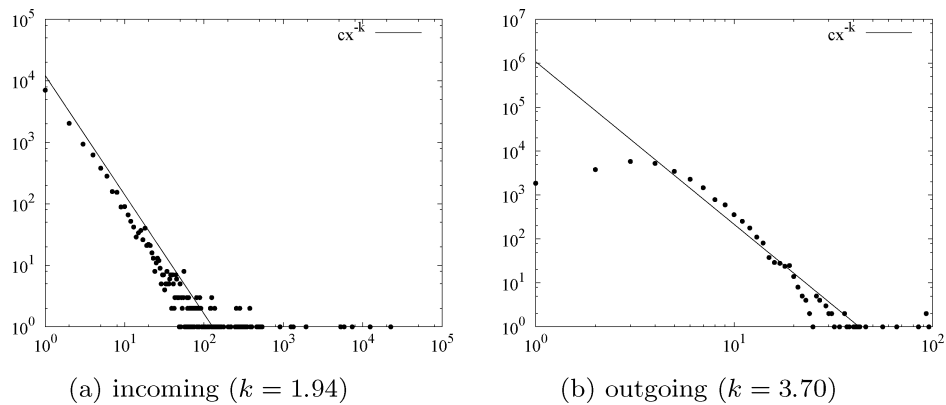


Fig. 3. Perl CPAN packages dependencies.

than parsing source code. The Java compiler may optimize away some references, and therefore dependencies, but the effects of such optimization cannot alter the overall picture.

3.2 Perl CPAN Packages

Perl is an interpreted language, probably the most popular among scripting languages. A reason for its current popularity is the vast number and variety of available libraries, called *packages*. Perl packages are namespaces that define program entities. The correspondence between files and packages is one to many, since one file may define multiple namespaces. A package B uses a package A when the code of B imports A to its own namespace; B then depends on A .

Perl packages are archived and published by developers all over the world in the Comprehensive Perl Archive Network (CPAN), where they are available as open source software. An archived CPAN package may contain a number of files, and therefore a number of Perl packages. CPAN also contains the Perl language itself, so that the entire Perl corpus can be studied by studying CPAN.

We built the dependency graph for Perl packages by going through all CPAN packages and collecting references among them (Figure 3). The number of CPAN packages increases daily; on October 21, 2004, CPAN contained 8,128 archives, making a total of 27,895 packages.

3.3 Shared Libraries in Open Source Unix Distributions

Libraries in the Unix operating system and its intellectual offsprings (e.g., Linux, FreeBSD—in what follows Unix refers to the whole family) come in two kinds. *Static libraries* are statically linked to programs at build time; the programs include the library code in their code, so that they are self-sufficient at runtime, at the cost of code bloat. *Shared libraries* are used by programs at runtime; programs are not encumbered with library code, but the program must find the required libraries in the executing system.

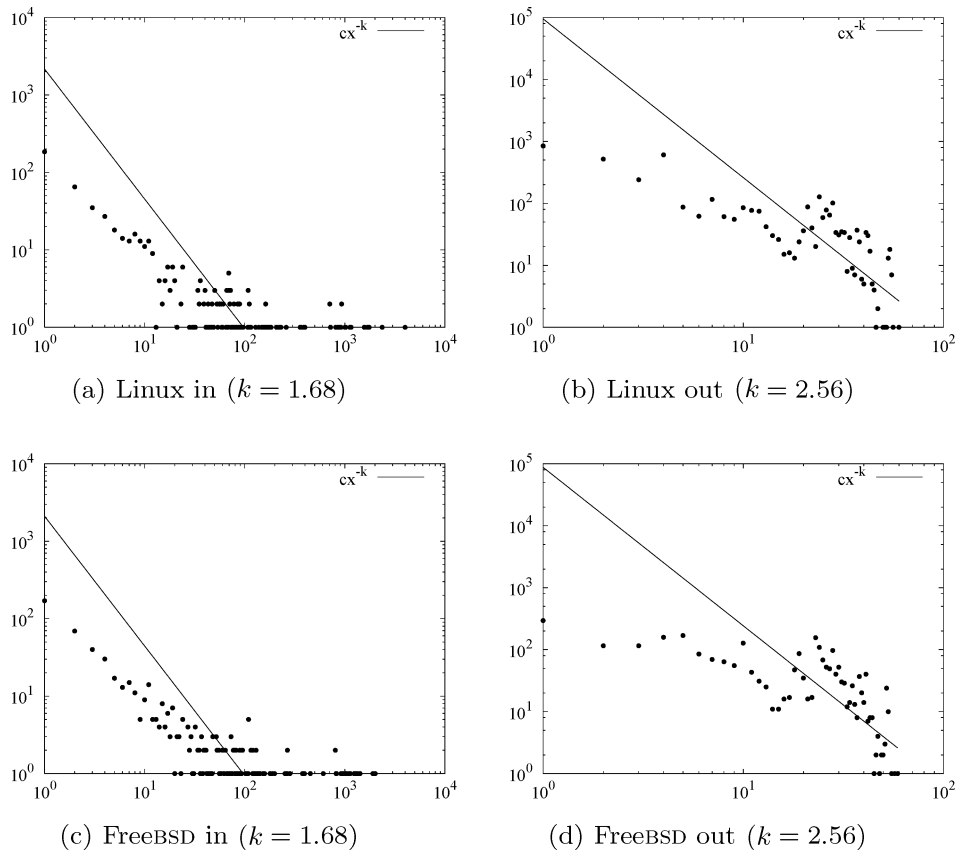


Fig. 4. Unix libraries dependencies.

Shared libraries have flourished to the point that an average Unix installation contains several hundreds of them. Shared libraries depend on one another when one uses the services of another, so a dependency graph emerges.

In Unix, these libraries are stored in a specific binary format called Executable and Linking Format (ELF); an ELF file contains several sections, one of which, the *dynamic section*, contains information pertaining to dynamic linking [TIS Committee 1995]. We examined a Fedora Linux Core 2 and a FreeBSD 5.21 system. By reading the dynamic section, we were able to build the dependency graph (Figure 4).

3.4 Microsoft Windows Executable and Dynamic Link Library Dependencies

There are three kinds of libraries in Microsoft Windows. Dynamically Linked Libraries (DLLs) are used for dynamic linking at runtime; static libraries are used for static linking, and import libraries are used at build time as stubs for DLLs to orchestrate the dynamic linking process.

Like shared libraries on Unix, the use of DLLs has flourished so that an average Windows installation contains several thousand of them. An executable

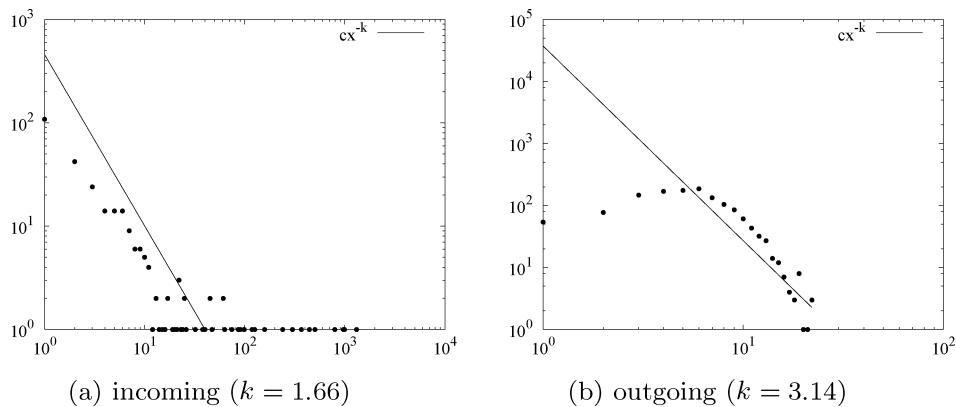


Fig. 5. Windows dependencies.

program, or a DLL, may depend on many DLLs; the evocative term “DLL hell” refers to the problems starting from broken dependencies, when a different version of a DLL is found in lieu of the required one.

Windows executables and dynamic link libraries are stored in a format called Common Object File Format (COFF), which also contains the information pertaining to dynamic linking. It is therefore possible to construct a dependency graph for Windows by examining COFF files, which we did for a system running Windows 2000 (Figure 5).

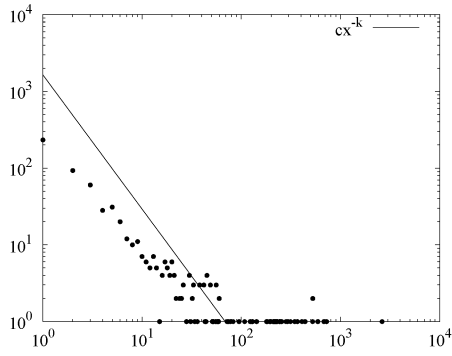
3.5 FreeBSD Ports

Users of the FreeBSD operating system can run on their systems thousands of applications ported, to it. When an application is ported, it enters the FreeBSD ports collection repository where all other ports are held. The ports collection provides standard facilities for downloading, building, installing and uninstalling the corresponding applications. There are more than 10,000 ports, added roughly in a linear progression since 1995.

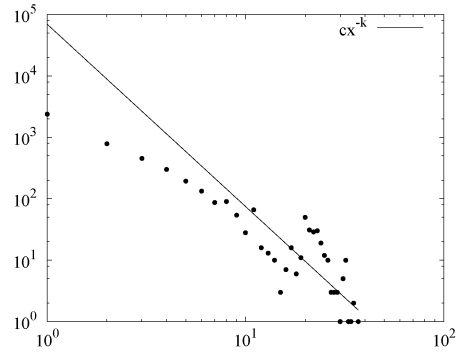
A port may have various types of dependencies on other ports. It may depend on a shared library, or on an executable program or data file during runtime; or it may require some executable programs or data files in order to be built (there are some additional dependency classes that do not concern us here). Each port lists these dependencies in a Makefile [Feldman 1979]. By processing each port’s Makefile we can construct the dependency networks for those dependencies of interest to us. Our data reflects the ports collection on October 25, 2004 (Figure 6).

3.6 T_EX and METAFONT

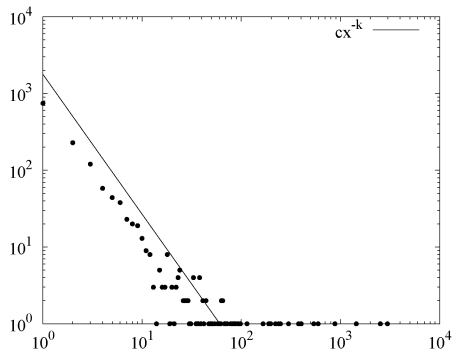
T_EX is a language for computer typesetting developed during 1977–1988 by Donald Knuth together with METAFONT, a language for designing fonts [Knuth 1984a; Knuth 1986b]. Both were developed using an approach called *literate programming* [Knuth 1984b]. In literate programming, the developer uses the



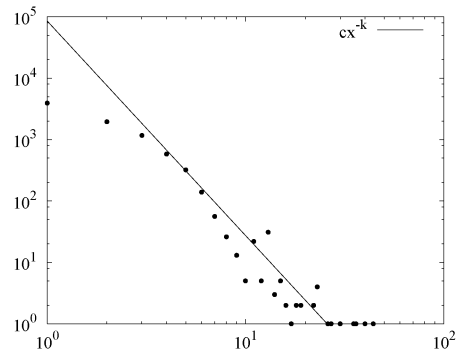
(a) FreeBSD libraries in ($k = 1.75$)



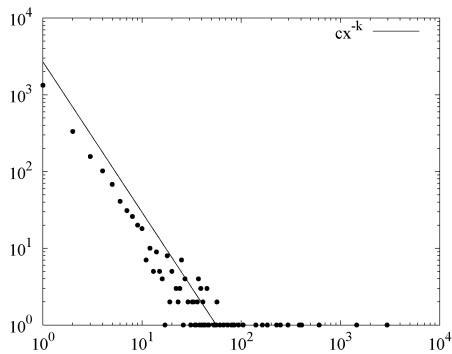
(b) FreeBSD libraries out ($k = 2.97$)



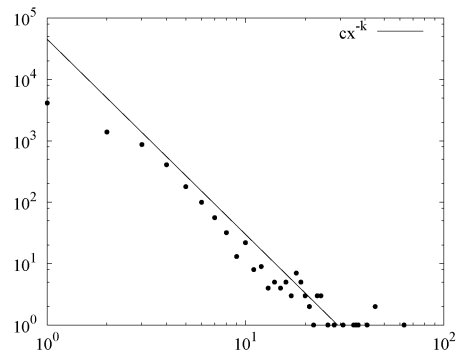
(c) FreeBSD build in ($k = 1.82$)



(d) FreeBSD build out ($k = 3.50$)

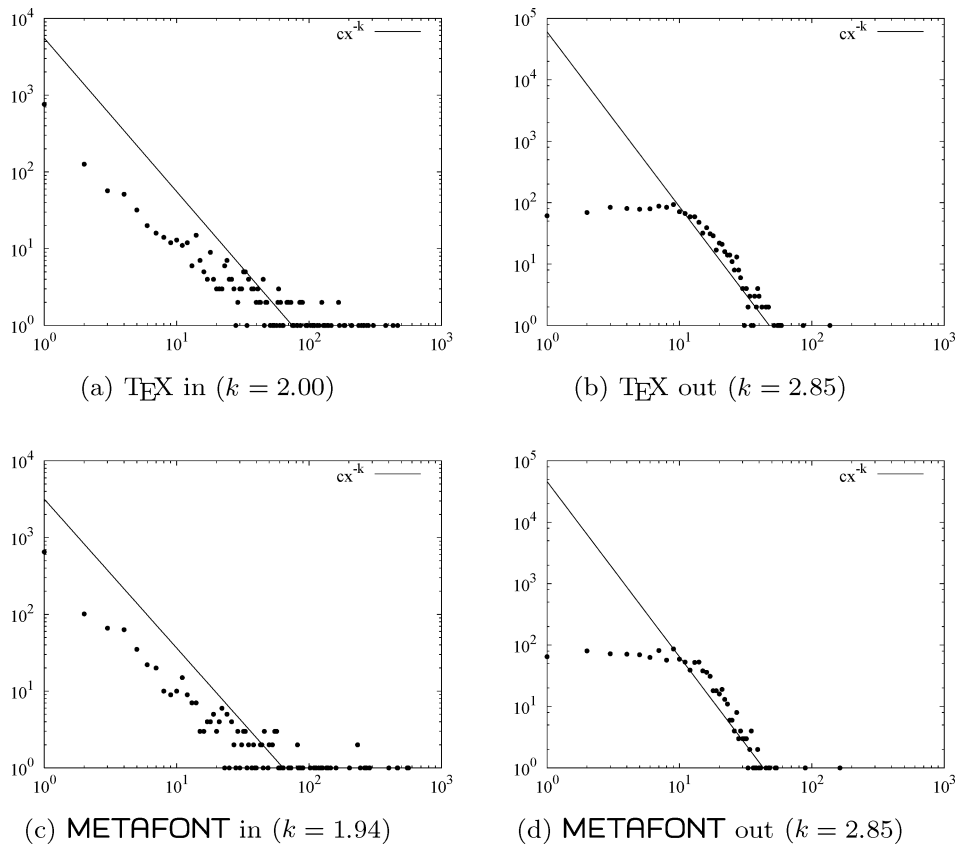


(e) FreeBSD run in ($k = 1.96$)



(f) FreeBSD run out ($k = 3.18$)

Fig. 6. FreeBSD ports dependencies.

Fig. 7. \TeX and METAFONT dependencies.

WEB system: a combination of a document formatting language, such as \TeX , and a programming language. A WEB user writes a program that contains both documentation and code; these are then separated by the system, producing elegant program documentation for human consumption, and program code for computer consumption.

\TeX and METAFONT were implemented using a structured programming approach (the underlying programming language was eventually Pascal). They comprise modules that are small program fragments. A module may include another module or use a function or a global variable defined somewhere else. The full WEB code of both \TeX and METAFONT has been made available by their author [Knuth 1986a, 1986c]. It is possible to work through their WEBS and build a dependency graph, which turns out to display scale-free characteristics (Figure 7).

3.7 Ruby

Ruby is a pure object-oriented scripting language. Ruby code can be put into Ruby library files that other code may require at runtime. Shared libraries can also be required by code that loads it as a Ruby extension. The Ruby 1.8.2

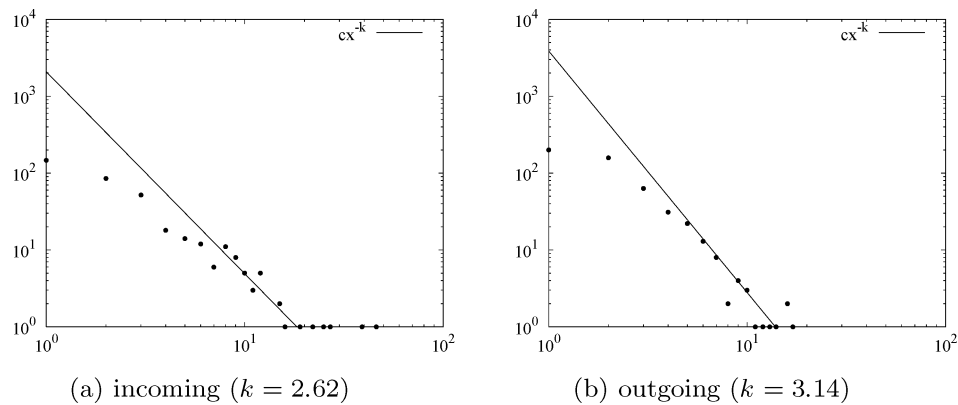


Fig. 8. Ruby dependencies.

distribution contains several hundred files requiring libraries, leading to a dependency graph whose link distribution is scale-free (Figure 8).

3.8 Other Evidence

A study of five UML class diagrams of Java projects [Valverde and Solé 2003] and 23 C/C++ projects found that class dependencies form scale-free networks (an earlier study examined only Java projects [Valverde et al. 2002]). The class diagrams described the architecture of the Java SDK version 1.2 (two diagrams), two video games, and a distributed application, and were either delivered by the program developers or, for the Java SDK, they were part of a software development tool. In contrast to the present study, where we use the full dependency graph, measurements concerned the largest connected components of the graphs. All graphs exhibited power law distributions. The proposed explanation was that software engineers may aim at optimal communication between modules, while avoiding the number of modules with a large number of dependencies; it is known that such strategies can lead to scale-free structures [Venkatasubramanian et al. 2004]. This design goal may relate to the fact that the cost of software graphs, when measured in terms of *cumulative component dependency* [Lakos 1996, p. 187], lies midway between optimal and worst-case structures—software designers do not appear to do too badly, nor exceedingly well; their performance is interesting since most likely it is the result of experience or common sense to them, and not of conscious efforts to build scale-free structures.

A similar study of C/C++ programs found power laws, both when graphs were constructed on the basis of collaboration between C++ classes, and when graphs were constructed on the basis of collaboration (calls) between C functions [Myers 2003]. Class graphs were generated indirectly, using the Doxygen documentation tool; the call graphs were obtained from the CodeViz package. *Refactoring*, the restructuring of existing code in order to optimize its design [Fowler 1999], is offered as a plausible explanation for the emergence of scale-free networks in software. A simple stochastic process, in which functions or classes

are split according to some probability when they are too long, reused instead of copied, and erased when used minimally, results in scale-free distributions. This assumes an evolutionary view of software construction, whereby software is continuously adapted to new requirements or bug fixing, and continually adapted to specific design goals; agile programming paradigms adopts such a view [Martin 2003].

More fine-grained analyses of program structure reveal similar results [Wheeldon and Counsell 2003]. A study using a Java source code analyzer found that twelve different metrics follow power laws. The metrics concerned the distribution of class references, methods, constructors, fields, and interfaces in classes, and the distribution of method parameters and return types. A more detailed follow-up study examined seventeen metrics at that abstraction level and identified fat tails, some of which would be best described by a power law, while some of them would be best described by other distributions [Baxter et al. 2006], something to which we return in Section 5.

The situation appears to be the same in the Smalltalk environment [Marchesi et al. 2004]. Smalltalk is a dynamically typed language, and relationships among classes and methods can be found using reflection (with some allowances made for the absence of static type information). A Smalltalk development system is also a Smalltalk program; the class dependencies of two Smalltalk systems were found to follow a power law. The implementation of methods with a specific name by classes was also found to follow a power law.

It is not just the static structure of software that is scale-free. When software is executed, an assortment of programming structures is allocated in the execution environment; these structures point to one another, so that a dependency graph during program execution emerges. An early study in 1977 examined the list structures in existence at the end of Lisp program execution. When the number of occurrences of each atom was plotted against its frequency rank in logarithmic scale, the data composed a line with slope approximately -1 , as postulated by Zipf's law [Clark and Green 1977]. More recently, a study of object graphs in snapshots taken during the execution of Java, C++, Smalltalk, and Self programs found that in all cases the object references followed a power law [Potanin et al. 2005]. The findings parallel those of the studies of the static structure of software: there is no typical object size, or scale; there is a significant number of popular objects that are heavily referenced, and there is also a significant number of objects that heavily reference others.

3.9 Remarks

Comparing Table I with Table II we see that the results are remarkably similar; this is even more remarkable when we consider that we examine far more variegated data.

When programs are viewed at the class level, our research corroborates the previous findings on class dependencies. Moreover, our examination of Java code is the only one that deals with compiled bytecode. In this way our results report on the nature of actual class dependencies, beyond those arrived at by dint of documentation diagrams or the distributed source code,

Table II. Review of Other Evidence

Dataset	size	k_{in}	k_{out}
Java, C/C++ [Valverde and Solé 2003]	27–5,285	1.94–2.54	2.41–3.39
C/C++ [Myers 2003]	187–5,420	1.9–2.5	2.4–3.3
Java [Wheeldon and Counsell 2003]	NA		0.71–3.66
Smalltalk [Marchesi et al. 2004]	1,797–3,022	2.07–2.39	2.3–2.73
Object graphs [Potanin et al. 2005]	15,064–1,259,668	2.5	3

which may not include all classes (as happens in the Java SDK); in addition, we are able to include in our analysis a very large closed source product—in fact, the size of the product dwarfs any other product examined in previous research.

Our analysis of \TeX and METAFONT shows that power laws are not confined to object-oriented languages, but also emerge in programs written using structured programming, as advocated in the early 1970s by Dahl, Dijkstra and Hoare, among others [Weiner 1978]. Programming language modules need not be classes in order for their dependencies to exhibit clustering; it seems that it is the concept of the module that matters, and not its specific language implementation.

We advance these findings on the microscopic level with a macroscopic view of software not undertaken before, where software systems are examined at the library or package level. Component-based software development is increasing in importance [Larsen (guest editor) 2000; Szyperski et al. 2002]. We found that on different operating systems, in different implementation languages, large scale components, like their fine-grained counterparts, also obey power laws.

Finally, the scope of the examined data, for each dataset, is more extensive than what was undertaken before. Instead of examining single projects, each macroscopic dataset contains material from a multitude of projects. While a Java or a C/C++ application may contain thousands of classes, these are mostly developed in the context of a single project, even if development is done by different people in different times. The CPAN, however, contains thousands of packages written for independent projects—and the same goes for the Unix libraries, the Windows dynamic link libraries and executables, and the FreeBSD ports.

4. IMPLICATIONS AND POINTS FOR FURTHER RESEARCH

The long, fat tails observed in our data impact on several aspects of software engineering. Software development is driven by various pragmatic considerations; in one way or another, a software engineer must allocate intellectual effort, time, money, and other resources efficiently. Unless a project is small, it is unavoidable that these resources will be allocated unequally among the project parts.

In practice, resources are indeed allocated unequally; not all people put in the same amount of work in the same project. We obtained an empirical indication of this fact in open source software development, by creating rank-size plots for the commits made by the committers of the Eclipse framework and the FreeBSD operating system. When in logarithmic scale, both plots had

a \lrcorner shape; a slightly inclined part followed by a sharply inclined part. The slightly inclined part corresponds to the bulk of the committers who make relatively few commits. Apart from them, there exist a few other committers who make many more commits. In other words, it appears that a core group of people contribute most to the project, aided by a sizeable number of people who contribute less or occasionally. Other studies on Linux show a similar “image of several *hundreds* of central members who do most of the coding, and several *thousands* of comparatively peripheral participants who contribute in a more indirect and sporadic fashion” [Weber 2004, p. 71]. This is not something unique in software, as “anyone who has worked or played on a team knows the apocryphal 80–20” rule [Weber 2004, p. 70] (recall also the discussion in Section 1).

The other distributions we have encountered have a similar structure of inequality. Some of the modules are more popular, and in this sense more important than others. Engineers can therefore allocate resources more efficiently by concentrating on these modules.

To get a measure of this, suppose we have a piece of software whose modules follow a power law. If we select at random a percent of the modules, we expect, if our sample is truly random, that we will cover approximately a percent of module usage. If, however, we rank the modules by their connectivity to arrive at a rank-size distribution, and then select the top a percent of the modules, we cover b percent of module usage, where, following (4), $b = a^\theta$. Going from a percent to a^θ percent is a marked improvement. For example, suppose we have a power law distribution with $k = 2.09$ (such as the J2SE SDK) and we choose $a = 0.05$. For the complementary cumulative distribution we have $k' = 1.09$. Using equation (4) we get $\theta \approx 0.0826$, $b \approx 0.78$. That is, we expect that 78 percent of module usage is covered by the selected 5 percent of the modules. The distributions we have seen in software prompt some points where further research might prove fruitful.

4.1 Reuse

Our work shows that reuse, in the projects we examined, is going on at a grand scale, and that it exhibits some special characteristics. With insufficient cataloguing of reusable components, the identification of suitable reusable components can be a bottleneck. It seems, though, that developers are adept at identifying and reusing promising candidates. The power law structures we have identified may play an important factor here: often reused components—those lying at the fat tail of the fan-in distribution—may enter the developers’ everyday vocabulary and become established parts of a software architecture’s design. The fact that the GTK *GLib* library is used by 728 ports in the FreeBSD repository, and the *libxml* by another 212, means that many developers find it more natural to reuse a C data structure library or an XML parser, than to build their own from scratch.

It seems that the burgeoning of open source software smooths the study and reuse of code fragments, or, *code scavenging*. Although code scavenging has not been measured in any way in the current work, the open publication of

large, production quality programs on the web provides ample opportunities of direct code reuse that are hard to neglect, for open source and close source projects alike [Spinellis and Szyperski 2004]. This was illustrated by the effort undertaken to estimate the extent of the damage caused by a bug found in the 1.1.3 version of the zlib library. According to the US-CERT vulnerability note VU #368819, 49 systems were affected (where a systems range from applications to full operating systems).

Pessimists note that software reuse may not live up to very high expectations due to the special characteristics of software, as opposed to other kinds of artifacts; in particular, that only a few reusable components can be reused profitably [Glass 1998]. Our data permits some validation of this observation. Although all our projects show reuse in action, the actual number of modules that are reused shows a marked inequality. Recall that one of the most conspicuous characteristics of power laws is that they exhibit a *winner takes all* pattern. In keeping with that, some modules, those with a high number of incoming links, tend to be overwhelmingly reused; at the same time, some modules, those with a high number of outgoing links, tend to use many other modules. The existence of huge class or module libraries may provide the potential for reuse, but developers tend to reuse only a small percentage of such libraries. This could have implications on the financial aspect of building reusable components, in that the added cost for making a component reusable will pay off very handsomely in some cases, but less so in the majority. As derived in equation (4), the top a percent of the modules will account for around a^θ of all module reuse.

We have no indication that the most heavily reused components are those of the best quality. Although this might be the case, it is not necessary. Existing software plays the role of a *shared infrastructure* on which new development builds. This shared infrastructure may be suboptimal, but the cost of changing it is too high to justify such efforts, which may lead to the *rich getting richer* model that is characteristic of power laws.

If we examine in more detail the emergence of power law distributions from reuse, given our observation in Section 3.8 that a process combining reuse and refactoring can arrive at scale-free structures, a parallel can be drawn with the emergence of power laws via optimization, where refactoring is seen as a redundancy removal mechanism. Further research is required to substantiate the claim, however. For instance, in information theory, power laws have been found in languages and the distribution of words in texts. This, however, does not reflect a deep law in natural language or communication, since even randomly generated strings exhibit power-like laws [Li 1992]. Rather, making very weak assumptions, such as a probability measure that favors small words (or, in our case, functions), and choosing a particular representation (i.e., rank as the independent variable) are enough to arrive at similar distribution patterns.

4.2 Quality Assurance

During the past 30, years empirical studies of programming errors have found that defects cluster in a manner that conforms to Pareto rules or power law

distributions [Endres 1975; Möller 1993; Ohlsson and Alberg 1996; Fenton and Ohlsson 2000; Ebert 2001; Chou et al. 2001; Ostrand and Weyuker 2002; Shull et al. 2002]. It seems that, on average, “about 80 percent of the defects come from 20 percent of the modules [. . .] Studies from different environments over many years have shown, with amazing consistency, that between 60 and 90 percent of the defects arise from 20 percent of the modules, with a median of about 80 percent” [Boehm and Basili 2001].

We performed a check ourselves by examining the errors in a well known program. The errors of \TeX have been minutely logged by its developer over the years [Knuth 1989]. In that context, an error denotes any modification, be it an enhancement (seen as an error of omission in the original specification), or a blunder, or an efficiency optimization, and so on. This is an unconventional project, in that the program users have been many, while the maintainer has been the same person throughout. Moreover, the data set is small, so the results are only indicative. Still, we confirm the existing findings, by finding that the errors of \TeX follow a power law, as shown in Table I.

We must note here, however, that program errors do not translate automatically to program failures; an error may remain undetected throughout the whole lifetime of a system, or it may appear so rarely that it makes no economic sense to fix it. According to a study on the costs of fixes, probably less than 10 percent of errors are worth fixing [Adams 1984].

We propose taking into account the power laws present in software in order to focus development efforts and save resources. Even though, as software developers, we may not be able to locate troublespots in a system, we have a measure of the impact of our efforts. Selecting modules at random, we may expect that approximately a percent of the dependencies will not lead to bugs propagated from bugs in the selected modules. Using equation (4) again, we find that by focusing on the top a percent of the modules, we may avoid the propagation of errors to up to a^θ other dependent modules. In general, scale-free networks are largely immune from random failures, but very sensitive to failures in the hubs [Albert et al. 2000]; this is also important in the security aspect of quality assurance.

The success and failure of beta-testing can be illuminated if we consider the scale-free distribution of bugs; beta-testers will quickly discover the small number of defects that make up a large proportion of those that can be found; at the same time, there will always be other effects, with a much lower probability of being found during testing, that will continue to torment unlucky users during production. However, despite the best of efforts, a system may still fail. Recovery-Oriented Computing accepts this as a fact of life and demands that systems appropriate for rapid recovery should be identified at various levels of abstraction [Candea et al. 2004]. This suggests that hub modules could be suitable candidates.

4.3 Optimization

The earliest references to power laws in software research came in 1963; they concern the Pareto principle and rank-size distributions in record

references [Heising 1963], and the Zipf-Mandelbrot law in the composition of dictionaries for efficient message transmission [Schwartz 1963]. In both cases, the unequal popularity of the examined elements provided opportunities for optimization. Similarly, the unequal popularity of modules means that some of them are much more likely to be looked up and used than others. Consequently, in a mechanism that organizes and provides access to such modules, we would expect efficiency gains if they were organized in a way that takes into account their popularity.

This has been a subject of research in search algorithms. For example, if we use a binary search tree for looking up the modules, and these modules follow Zipf's law, we can benefit if we insert the keys in decreasing order of importance [Knuth 1998, p. 435]. Of course, we may not know each module's popularity in advance. To deal with this, researchers have devised self-organizing data structures that adapt to the search frequencies [Albers and Westbrook 1998].

Such optimization may not matter a lot when modules are kept in a server and developers browse them at their leisure; but it may matter quite a bit when a system's speed of execution depends on efficient storage and retrieval of the required modules.

For instance, on the Java Virtual Machine [Lindholm and Yellin 1999], when a specific class's code is required, such as for creating a new instance, the runtime system looks up the class code in a memory area called the *method area*. The implementation does not specify how the method area should be organized, but it is clear that it should be efficient. We examined two Java Virtual Machine specifications, the open source Kaffe and the Java 2 Platform Standard Edition 5.0 implementation provided by Sun. Kaffe's documentation states that the method area is implemented using a hash table. We examined the source code of the Sun SDK to uncover the method area access method, which turned out to be, again, a hash table. None of the implementations takes into account the classes' unequal popularity.

Self-organizing structures are not suitable for all situations; but they behave very well when requests cluster around certain objects, as happens in compiler symbol tables and business transactions [Allen and Munro 1978]. For instance, a *move-to-front* heuristic, where a referenced symbol is moved to the front of a list, could reduce substantially the run time of interpreted programs that store symbols in a linear symbol table [Bentley and McGeoch 1985].

In some cases, dynamic loading of modules takes account of their popularity. Hence, references to Unix libraries in many implementations are resolved using a cache mechanism maintained by the *ldconfig* program (which might also benefit by taking into account library popularity). It would be interesting to see whether, in similar situations, using a self-organizing structure would result in efficiency gains.

The unequal popularity of modules is also an important guiding factor in the design and use of multi-level storage architectures. The power law structures we identified both describe and prescribe a *locality of reference* phenomenon on access patterns of a system's code [Denning 2005]. By shaping a processor's code cache or an operating system's disk buffer cache according to expected access patterns, we may be able to eliminate waste of memory resources or

increase the storage system's performance. The frequency of machine instructions has already been used in optimizing CPU architectures: early designs used instruction frequencies to optimize the length of variable length opcodes, while later RISC designs took a more radical approach and optimized the processor's instruction set around the most frequently used instructions.

4.4 Language and Library Design

One of the early discussions of power laws, more than 50 years ago, was in the domain of human languages; specifically, it was established that the frequency of words seems to follow Zipf's law, or some related formula. This hints that we should examine whether the same applies to computer programs.

As a prototype probe, we examined the use of Unix system calls and C library functions in Unix libraries. The Unix system calls provide the programmer with access points to operating system services. The system calls differ in various Unix implementations; we examined a Fedora Linux Core 2 and a FreeBSD 5.21 system. We plotted the number of calls for each specific call, and ordered them by rank, obtaining a rank-size distribution. We did the same for the functions defined in the standard C library. The popularity of the 242 identified system calls in the Linux installation we checked, is almost a straight line in logarithmic scale; the same is true for the 295 system calls identified in the FreeBSD system. 135 C standard library functions trace similar paths for both systems; the results can be seen in Table I.

That both the system calls and the C library functions seem to follow a Zipfian distribution, allows some suggestions. First, in line with what we have already said, extra care should be paid in the most popular, and critical, system calls or functions. The effect of bugs or bottlenecks there would ripple through and reach many other parts of a system. Second, this kind of research could be extended to other aspects of a programming system, other computer languages, and other operating environments. In a feature-rich language, we would expect that not all features are used with the same intensity: some are reached rarely, only by the more experienced programmers.

At a lower level, it is interesting to research further whether similar patterns are found in language keywords and computer instruction sets. We examined the CPU instruction frequencies for various operating systems and machine architectures. The results are shown in Figure 9. In all cases, there are some instructions that are much more frequently used than others, again pointing to a Zipfian distribution (note that even though most of the data lies in the tails, making the fit doubtful, the point about the inequality among the instructions holds). That does not mean that the same instructions are always the most popular. In FreeBSD 4.11, 15% of all instructions were *push*, while in Linux 2.4 only about 2% were, which raises the question whether Linux is less modular than FreeBSD.

4.5 Patterns in Our Results

In Table I one may spot that $k_{in} < k_{out}$, $r_{in}^2 > r_{out}^2$, and $r_{in}^2 \gg r_{out}^2$ for the Linux and FreeBSD libraries. The findings of others in Table II show a similar pattern

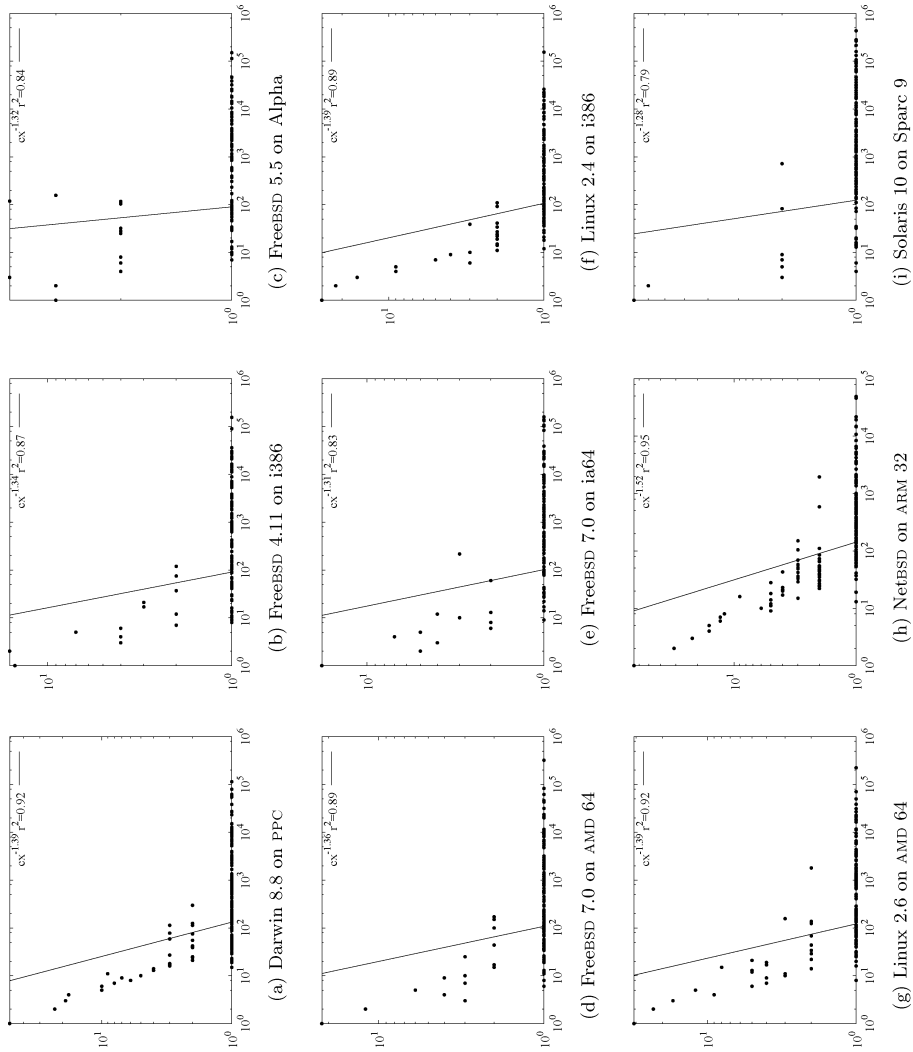


Fig. 9. CPU instruction frequencies; the number of times an instruction appears is in the x axis, the number of such instructions is in the y axis.

for the exponents. A smaller exponent denotes a less unequal distribution. The probability that a module has a certain number of incoming links decreases more slowly than that of outgoing links. In terms of dependencies, it seems that it is more likely to have modules that are very popular and used by many others, than to have modules that use a large number of other modules.

One explanation for this fact could be that the marginal cost of incoming links is considerably lower than that of outgoing links. Once a module is reused, the cost of adding other reuse clients is minimal, because most support costs (such as bug fixes and new releases) are simply distributed among a larger number of clients. On the other hand, there is no mechanism for establishing economies of scale for using other modules: there is always a considerable cost associated with adding new dependencies to a system. One other reason for this difference could be that incoming links can benefit from network effects [Economides 1996]: widely used modules are more likely to be robust and better supported. Again, no such mechanism appears to aid outgoing links.

The difference in the r^2 measurements shows greater dispersion for the outgoing links, especially in the Linux and FreeBSD libraries. It seems that outgoing links do not fit the power law distribution as ideally as the incoming links; in general, questions of statistical inference must be approached with care, as we see next.

5. CONCLUSIONS

The apparent ubiquity of power laws inspires intriguing images of common underlying structures, patterns, and laws governing all sorts of complex systems. It is perhaps wise to temper enthusiasm with caution, as was indeed advocated when this ubiquity was first intimated: “No one supposes that there is any connection between horse-kicks suffered by soldiers in the German army and blood cells on a microscope slide other than that the same [. . .] scheme provides a satisfactory abstract model of both phenomena. It is in the same direction that we shall look for an explanation of the observed close similarities along the [various] distributions” [Simon 1955].

In fact, the situation is not without precedent. More than 60 years ago, the statistics literature was teeming with evidence that a distribution fitted all kinds of phenomena. “Lengthy tables, complete with chi-square tests, supported this thesis for human populations, for bacterial colonies, development of railroads, etc. Both height *and* weight of plants and animals were found to obey the [same] law even though it is theoretically clear that these two variables cannot be subject to the same distribution. Laboratory experiments on bacteria showed that not even systematic disturbances can produce other results. Population theory relied on [. . .] extrapolations (even though they were demonstrably unreliable)” [Feller 1971, pp. 52–53].

The above does not refer to a power law distribution, but to a different one, the logistic distribution; this was discredited when it was found that, in fact, other distributions could fit the same data as well or better. “Theories of this nature are short-lived because they open no new ways, and new confirmations of the same old thing soon grow boring. But the naive reasoning as such has not been

superseded by common sense, and so it may be useful to have an explicit demonstration of how misleading a mere goodness of fit can be” [Feller 1971, p. 53].

In fact, power laws do not enjoy a monopoly in having a good fit over a range of physical, economical, or social phenomena. There exist other distributions that also have a good fit with respect to some of them [Laherrère and Sornette 1998; Shiode and Batty 2000; Mitzenmacher 2004; Baxter et al. 2006]. It seems that what is surprising is that power law distributions are easy to generate, and by a variety of mechanisms [Fox Keller 2005]. This might be alarming, given what we have just quoted.

All these distributions, however, have in common long, fat tails where a small fraction of the population takes over a large portion of the measured resource. It is based on this that we have proposed a set of implications for software. Whether fat tails result from a lognormal distribution, or a stretched exponential distribution, or yet another, their existence is important to software engineering.

Fat tails appear in the relationships between software modules, be they functions, classes, or libraries, open source or closed source, recently developed or vintage code. The difference in scale and implementation between the modules we have considered leads us to believe that it is not the specific size or technology of those modules that somehow goad a system into displaying these characteristics.

We refrain from offering a theory on how fat tails emerge in software; a comprehensive theory would show which, if any, single distribution molds software, and why other promising distributions are incompatible. If there seems to be a common thread linking all our data, it is that it is structured around relatively independent modules that offer their services to other modules, whose services they might, in their turn, use; whether such structure tends, in general, to lead to power laws, in software or elsewhere, is yet to be seen; the pronouncement of general laws must be approached with care. But even though we might not know the underlying reason, the presence of fat tails illuminates several facets of software engineering research, prescribes current practice, and suggests important new research venues.

APPENDIX

The dependencies in the various networks considered in this work were derived by a suite of custom-written programs.

The Java dependencies can in principle be derived by decompiling the byte-codes of the Java files; we were able to work at a relatively higher level by using the Byte Code Engineering Library (BCEL), developed by the Apache Software Foundation.

We studied the Perl CPAN packages with a bot, written in Perl, that connects to CPAN, reads the complete list of CPAN packages, downloads and parses each one of them, and establishes package dependencies by detecting *use* and *require* directives.

To build the Unix shared objects dependency network we wrote a Perl script that uses the *ldd* and *readelf* utilities to find the dependencies between

libraries; *ldd* gives the transitive closure of a library's dependencies and *readelf* can be used to sift the direct dependencies only.

The FreeBSD ports collection was studied with a Bourne shell script; for each port, the script calls *make* on its Makefile with appropriate arguments to obtain the values of Makefile variables containing the required dependencies; the output was fed to a Perl script that produced dependency lists for each port.

DLL dependencies can be found with the DUMPBIN command line tool, provided with the Windows SDK. DUMPBIN displays information about COFF binary files. We used a Perl script to drive and process the results of DUMPBIN for all files that import DLLs.

We studied T_EX and METAFONT by using the original WEB sources for the two programs as provided by their author in the Comprehensive T_EX Archive Network (CTAN). We used WEAVE to derive the corresponding documentation in T_EX format; from the indices we extracted the dependencies using Perl scripts. We assumed that an index item that was defined only once would be a global variable or function that we could use. To analyze the errors of T_EX, we used the error log as posted by its author on CTAN. A Perl script extracted the modules affected by each error.

The Eclipse and FreeBSD committers were ranked by relying on the information recorded in the cvs system, used by both projects. We extracted all commit information using the cvs *annotate* command, and then used Perl scripts to rank the results.

To check the distributions of the Unix system calls and the C standard library functions, we used a couple of Perl scripts that collected the system calls defined in each implementation. Another Perl script examined the libraries in the systems and counted the uses of each system call or standard library function by parsing the output of the *readelf* program. Ruby dependencies were derived using a Ruby script that catalogued and examined all Ruby files in the Ruby 1.8 distribution. The CPU instruction frequencies were derived from the operating system kernels using a series of Unix pipes and further processed with Perl and Python scripts.

Power laws are easily cast into linear terms by taking the logarithm of the data so that we can use simple least squares fitting. To avoid bias due to outliers, we worked with the complementary cumulative definition (2), in which outliers are subsumed, and then converted the results of the fit to the initial distribution. We wrote a suite of Python programs that try to fit the data using the plain or the complementary cumulative distribution. The complementary distribution was not used for the system calls and the C standard library functions that were gathered in rank-size distributions, nor for the city sizes figure. The correlation coefficient r^2 is given with the results.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their insightful comments that improved this work considerably. The discussion on *shared infrastructures*, among other things, was directly suggested by one of them.

REFERENCES

- ADAMIC, L. A. 2000. Zipf, power-laws, and Pareto—a ranking tutorial. <http://www.hpl.hp.com/research/idl/papers/ranking/ranking.html>.
- ADAMIC, L. A. AND HUBERMAN, B. A. 2002. Zipf's law and the internet. *Glottometrics* 3, 143–150.
- ADAMS, E. N. 1984. Optimizing preventive service of software products. *IBM J. Resear. Devel.* 28, 1, 2–14.
- ALBERS, S. AND WESTBROOK, J. 1998. Self-organizing data structures. In *Online Algorithms: The State of the Art*, A. Fiat and G. J. Woeginger, Eds. Lecture Notes in Computer Science, vol. 1442. Springer-Verlag, Berlin, 31–51.
- ALBERT, R., JEONG, H., AND BARABÁSI, A.-L. 1999. Diameter of the World-Wide Web. *Nature* 401, 130.
- ALBERT, R., JEONG, H., AND BARABÁSI, A.-L. 2000. Error and attack tolerance of complex networks. *Nature* 406, 378–382.
- ALLEN, B. AND MUNRO, I. 1978. Self-organizing binary search trees. *J. ACM* 25, 4, 526–535.
- BARABÁSI, A.-L. 2002. *Linked: The New Science of Networks*. Perseus Publishing, Cambridge, MA.
- BARABÁSI, A.-L. AND ALBERT, R. 1999. Emergence of scaling in random networks. *Science* 286, 509–512.
- BARABÁSI, A.-L., ALBERT, R., AND JEONG, H. 1999. Mean-field theory for scale-free random networks. *Physical A* 272, 173–187.
- BARABÁSI, A.-L. AND BONABEAU, E. 2003. Scale-free networks. *Scientific Amer.* 288, 5, 50–59.
- BAXTER, G., FREAN, M., NOBLE, J., RICKERBY, M., SMITH, H., VISSER, M., MELTON, H., AND TEMPERO, E. 2006. Understanding the shape of java software. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*. ACM Press, New York, NY, 397–412.
- BENTLEY, J. L. AND MCGEOCH, C. C. 1985. Amortized analyses of self-organizing sequential search heuristics. *Comm. ACM* 28, 4, 404–411.
- BOEHM, B. AND BASILI, V. R. 2001. Software defect reduction top 10 list. *IEEE Softw.* 34, 1, 135–2001.
- BOEHM, B. W. 1987. Industrial software metrics top 10 list. *IEEE Softw.* 4, 9, 84–85.
- CANDEA, G., BROWN, A. B., FOX, A., AND PATTERSON, D. 2004. Recovery-oriented computing: building multitier dependability. *IEEE Comput.* 37, 11, 60–67.
- CHOU, A., YANG, J., CHELE, B., HALLEM, S., AND ENGLER, D. 2001. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating System Principles*. ACM Press, New York, NY, 73–88.
- CLARK, D. W. AND GREEN, C. C. 1977. An empirical study of list structure in Lisp. *Comm. ACM* 20, 2, 78–87.
- DENNING, P. J. 2005. The locality principle. *Comm. ACM* 48, 7, 19–24.
- DOROGOVITSEV, S. N. AND MENDES, J. F. F. 2003. *Evolution of Networks: From Biological Nets to the Internet and WWW*. Oxford University Press, Oxford, U.K.
- EBERT, C. 2001. Metrics for indentifying critical components in software projects. In *Handbook of Software Engineering and Knowledge Engineering*, S. K. Chang, Ed. Vol. 1, Fundamentals. World Scientific, London, U.K.
- ECONOMIDES, N. 1996. The economics of networks. *Int. J. Indust. Org.* 16, 4, 673–699.
- ENDRES, A. 1975. An analysis of errors and their causes in system programs. *ACM SIGPLAN Notices* 10, 6, 327–336.
- FALOUTSOS, M., FALOUTSOS, P., AND FALOUTSOS, C. 1999. On power-law relationships of the internet topology. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'99)*. ACM Press, New York, NY, 251–262.
- FELDMAN, S. I. 1979. Make—a program for maintaining computer programs. *Softw. Prac. Exper.* 9, 4, 255–265.
- FELLER, W. 1971. *An Introduction to Probability Theory and Its Applications* 2nd ed. Vol. 2. John Wiley & Sons, New York, NY.

- FENTON, N. E. AND OHLSSON, N. 2000. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Softw. Eng.* 26, 8, 797–814.
- FOWLER, M. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, MA.
- FOX KELLER, E. 2005. Revisiting “scale-free” networks. *BioEssays* 27, 10, 1060–1068.
- GLASS, R. L. 1998. Reuse: What’s wrong with this picture? *IEEE Softw.* 15, 2, 57–59.
- HEISING, W. P. 1963. Note on random addressing techniques. *IBM Syst. J.* 2, 2, 112–116.
- HENRY, S. AND KAFURA, D. 1981. Software structure metrics based on information flow. *IEEE Trans. Softw. Eng.* 7, 5, 510–518.
- HUBERMAN, B. A. AND ADAMIC, L. A. 1999. Growth dynamics of the World-Wide Web. *Nature* 401, 131.
- KNUTH, D. E. 1984a. *The TeXbook*. Computers & Typesetting, vol. A. Addison Wesley Publishing Company, Reading, MA.
- KNUTH, D. E. 1984b. Literate programming. *Comput. J.* 27, 97–111.
- KNUTH, D. E. 1986a. *TeX: The Program*. Computers & Typesetting, vol. B. Addison Wesley Publishing Company, Reading, MA.
- KNUTH, D. E. 1986b. *The METAFONT Book*. Computers & Typesetting, vol. C. Addison Wesley Publishing Company, Reading, MA.
- KNUTH, D. E. 1986c. *METAFONT The Program*. Computers & Typesetting, vol. D. Addison Wesley Publishing Company, Reading, MA.
- KNUTH, D. E. 1989. The errors of TeX. *Softw. Prac. Exper.* 19, 7, 607–685.
- KNUTH, D. E. 1998. *Sorting and Searching*, 2nd ed. The Art of Computer Programming, vol. 3. Addison-Wesley, Reading, MA.
- LAHERRÈRE, J. AND SORNETTE, D. 1998. Stretched exponential distributions in nature and economy: “fat tails with characteristic scales.” *Europ. Phys. J. B* 2, 525–539.
- LAKOS, J. 1996. *Large Scale C++ Software Development*. Addison-Wesley, Boston, MA.
- LARSEN (GUEST EDITOR), G. 2000. Component-based enterprise frameworks. *Comm. ACM* 43, 10, 24–66.
- LI, W. 1992. Random texts exhibit zipf’s-law-like word frequency distribution. *IEEE Trans. Inform. Theory* 38, 6, 1841–1845.
- LINDHOLM, T. AND YELLIN, F. 1999. *The Java Virtual Machine Specification*, 2nd ed. Addison-Wesley, Reading, MA.
- MANDELBROT, B. 1953. An informational theory of the statistical structure of language. In *Proceedings of the 2nd London Symposium on Communication Theory*, W. Jackson, Ed. Butterworth, London, 486–504.
- MANDELBROT, B. M. 1951a. Adaptation du message à la ligne de transmission: I. Quanta d’information. *Comptes Rendus des séances de l’Académie des Sciences* 232, 1636–1740.
- MANDELBROT, B. M. 1951b. Adaptation du message à la ligne de transmission: II. Interprétation physiques. *Comptes Rendus des séances de l’Académie des Sciences* 232, 2003–2005.
- MANDELBROT, B. M. 1983. *The Fractal Geometry of Nature*. W. H. Freeman and Company, New York, NY.
- MARCHESI, M., PINNA, S., SERRA, N., AND TUVERI, S. 2004. Power laws in Smalltalk. In *Proceedings of the 12th European Smalltalk User Group Joint Event*. Köthen, Germany.
- MARTIN, R. C. 2003. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, Upper Saddle River, NJ.
- MITZENMACHER, M. 2004. A brief history of generative models for power law and lognormal distributions. *Internet Mathematics* 1, 2, 226–251.
- MÖLLER, K.-H. 1993. An empirical investigation of software fault distribution. In *Proceedings of the 1st International Metrics Symposium*. IEEE Computer Society Press, Los Alamitos, CA, 82–90.
- MYERS, C. R. 2003. Software systems as complex networks: structure, function, and evolvability of software collaboration graphs. *Phys. Rev. E* 68, 046116.
- NEWMAN, M. E. J. 2005. Power laws, pareto distributions and zipf’s law. *Contem. Phys.* 46, 5, 232–351.

- OHLSSON, N. AND ALBERG, H. 1996. Predicting fault-prone software modules in telephone switches. *IEEE Trans. Softw. Eng.* 22, 12, 886–894.
- OSTRAND, T. J. AND WEYUKER, E. J. 2002. The distribution of faults in a large industrial software system. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM Press, New York, NY, 55–64.
- PARETO, V. 1897. *Cours d'Économie Politique*. Rouge, Lausanne.
- POTANIN, A., NOBLE, J., FREAN, M., AND BIDDLE, R. 2005. Scale-free geometry in object-oriented programs. *Comm. ACM* 48, 5, 99–103.
- SCHWARTZ, E. E. 1963. A dictionary for minimum redundancy encoding. *J. ACM* 10, 4, 413–439.
- SHIODE, N. AND BATTY, M. 2000. Power law distributions in real and virtual worlds. In *Proceedings of the 10th Annual Internet Society Conference (INET'00)*. Yokohama.
- SHULL, F., BASILI, V., BOEHM, B., BROWN, A. W., COSTA, P., LINDVALL, M., PORT, D., IOANA, R., TESORIERO, R., AND ZELKOWITZ, M. 2002. What we have learned about fighting defects. In *Proceedings of the 8th IEEE Symposium on Software Metrics (METRICS'02)*. IEEE Computer Society, Los Alamitos, CA.
- SIMON, H. A. 1955. On a class of skew distribution functions. *Biometrika* 42, 3/4, 425–440.
- SPINELLIS, D. AND SZYPERSKI, C. 2004. How is open source affecting software development? *IEEE Softw.* 21, 1, 28–33.
- SZYPERSKI, C., GRUNTZ, D., AND MURER, S. 2002. *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley, London.
- TIS COMMITTEE. 1995. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification. Version 1.2.
- VALVERDE, S., CANCHO, R. F., AND SOLÉ, R. V. 2002. Scale-free networks from optimal design. *Europhysics Lett.* 60, 4, 512–517.
- VALVERDE, S. AND SOLÉ, R. V. 2003. Hierarchical small worlds in software architecture. Working Paper 03-07-044, Santa Fe Institute, Santa Fe, NM.
- VENKATASUBRAMANIAN, V., KATARE, S., PATKAR, P. R., AND MU, F.-P. 2004. Spontaneous emergence of complex optimal networks through evolutionary adaptation. *Comput. Chem. Engin.* 28, 9, 1789–1798.
- WEBER, S. 2004. *The Success of Open Source*. Harvard University Press, Cambridge, MA.
- WEINER, L. H. 1978. The roots of structured programming. *ACM SIGCSE Bull.* 10, 1, 243–253.
- WHEELDON, R. AND COUNSELL, S. 2003. Power law distributions in class relationships. In *Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*. IEEE Computer Society Press, Los Alamitos, CA, 45–54.
- YULE, G. U. 1925. A mathematical theory of evolution, based on the conclusions of Dr. J. C. Willis, F.R.S. *Philoso. Transa. Royal Soc. London: Series B* 213, 21–87.
- ZIPF, G. K. 1935. *The Psycho-Biology of Language: An Introduction to Dynamic Philology*. Houghton Mifflin, Boston, MA.
- ZIPF, G. K. 1949. *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology*. Addison-Wesley, Reading, MA.

Received August 2005; revised January 2007; accepted September 2007