Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder: D-CCFinder

Simone Livieri[†] Yoshiki Higo[†] Makoto Matushita[†] Katsuro Inoue[†]

[†]Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
E-mail: {simone, y-higo, matusita, inoue}@ist.osaka-u.ac.jp

Abstract

The increasing performance-price ratio of computer hardware makes possible to explore a distributed approach at code clone analysis. This paper presents D-CCFinder, a distributed approach at large-scale code clone analysis. D-CCFinder has been implemented with 80 PC workstations in our student laboratory, and a vast collection of open source software with about 400 million lines in total has been analyzed with it in about 2 days. The result has been visualized as a scatter plot, which showed the presence of frequently used code as easy recognizable patterns. Also, D-CCFinder has been used to analyze a single software system against the whole collection in order to explore the presence of code imported from open source software.

1. Introduction

Code clone analysis is an interesting and emerging topic in software engineering research[13, 19]. Through code clone analysis, we can identify not only simple code shares, but also various characteristics of software systems such as code evolution[14, 22].

We are interested in applying code clone analysis to a large set of source code consisting of many software systems in order to see the relationships between them. In these days, many open source systems are being developed, and parts of those systems are commonly used by other systems[5]. We think that this kind of interaction between software systems can be identified and viewed by code clone analysis.

In order to achieve this goal, we need to resolve the scalability issue of current code clone detection systems. Many approaches and tools for code clone detection[1, 2, 7, 12, 15, 16, 17, 18] have been proposed and, among them, one tool offering good scalability and ease of use is CCFinder[12], a token-based code clone analysis tool that can analyze, in the ideal case, up to 5.2 million of lines of C code in about 18 minutes on a PC-based workstation (Intel Xeon 2.8GHz CPU with 2 GB memory).

In this paper, we have chosen, as the analysis target, the collection of open source software used for FreeBSD (hereinafter called *"the FreeBSD target"*), which consists of about 400 million lines in C and is about 10.8 G bytes in total. This is theoretically 80 times larger than the input size limit of CCFinder. In order to be analyzed with CCFinder, the target must be partitioned into small pieces, and each piece has to be analyzed with CCFinder. This would ideally require about 3200 single runs of CCFinder and it would take about 40 days on a single PC-based workstation.

We will reduce the required time by introducing a distributed computing system named Distributed CCFinder (D-CCFinder for short). In our experiments D-CCFinder performed the same task in about 2 days when ran on 80 PC workstations in our department's student laboratory. In this paper, we discuss the computational model and implementation of D-CCFinder.

D-CCFinder has been applied to the FreeBSD target, and we have obtained a global view of the code clones among many open source systems. Also, D-CCFinder is used to investigate the possible presence of imported code from the FressBSD target in one of our internal projects. We believe that this approach can be very useful in aiding the detection of illegal or unintentional use of copyright-protected source code.

Contributions of this paper are as follows:

- we will propose an approach to scale a code clone analysis tool using a distributed environment; the overall computational model and implementation of D-CCFinder will be discussed;
- we will show a global view of the code clones for a collection of many open source systems, which, to the knowledge of the authors of this paper, has not yet been

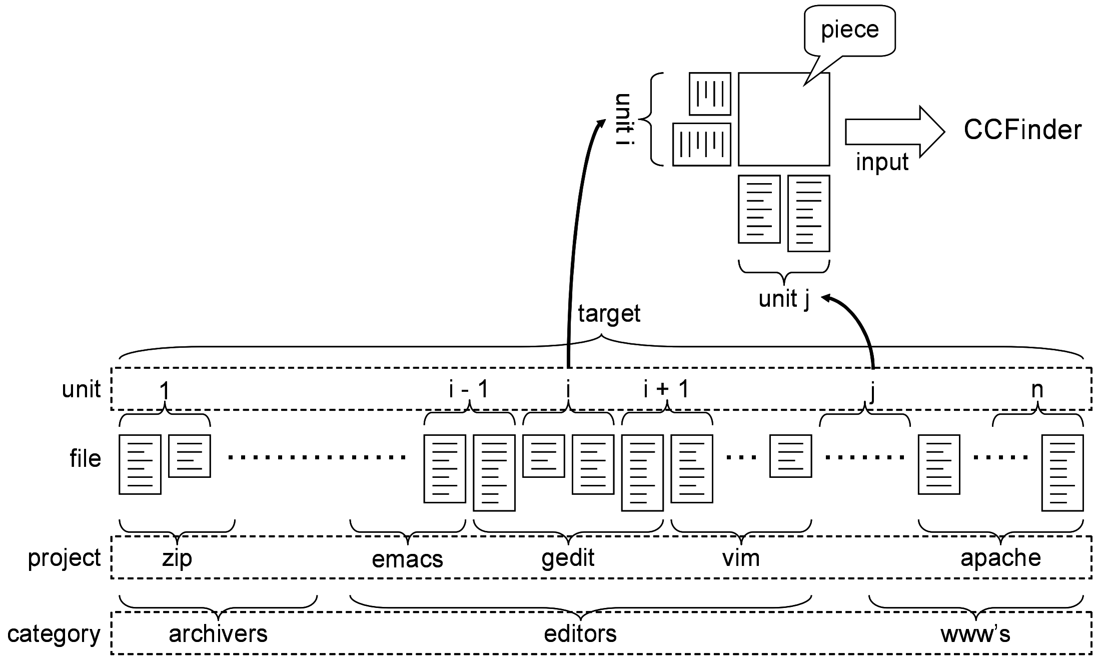

Figure 1. Relation between project, category, target and unit

presented before;

- application of D-CCFinder to software copyright violation detection will be explored; this will be performed by analyzing the code clones between a single project and a collection of many open source projects.

In Section 2, the computational model of D-CCFinder is defined, and the implementation of D-CCFinder is presented in Section 3. In Section 4, we show the experiments performed using D-CCFinder, and in Section 5, some discussions and related works are presented. Finally, we conclude our paper with a few remarks in Section 6.

2. D-CCFinder Model

In this section, we will describe D-CCFinder, which is an analysis system for the target input.

2.1. Target input

A collection of source files which are needed to build a software system is called *project*, and each project has its unique project name: ZIP, emacs, apache, and windowmaker are examples of project names (see Figure 1). To simplify our discussion, we will limit our analysis to projects written only in C language.

A collection of projects sharing a specific feature is called *category*. For example emacs, vim, and gedit are projects in the editors category.

The input of D-CCFinder is called *target* which is composed of multiple unique categories. We assume that there are no duplicated categories or projects in the target.

Another dimension by which the target can be partitioned is the *unit*. A *unit* is a collection of source files of a single or multiple projects. The total size of those files must be less than or equal to the *unit size*, decided by the user in advance.

A computational element specified by two units is called *piece*, and it will be the input of a distributed process. The upper part of Figure 1 illustrates the relation between these terms.

2.2. Prospective Output

The objective of D-CCFinder is to find clones among projects in the target. With clone we intend a pair of code fragments which are exactly the same except comments, white spaces, or carriage returns, or which differ only on user-defined identifiers. The former clones are called *type 1* and the latter ones are called *type 2* in [3]. We expect both type 1 and 2 clones existing in the target to be present in the output of D-CCFinder.

Because of the large size of the target, a considerable

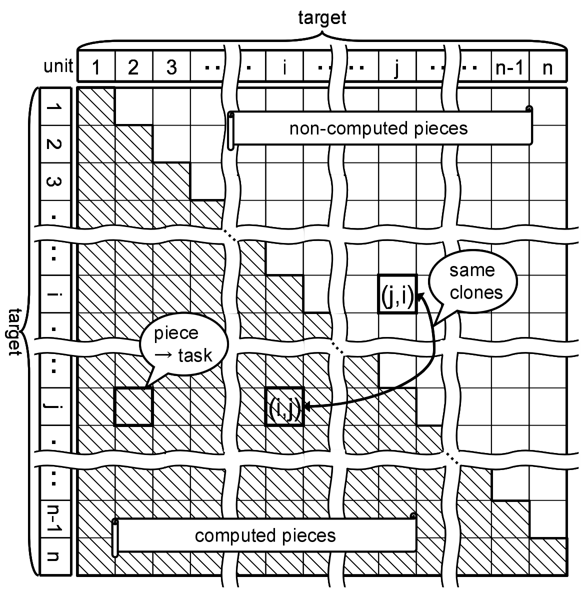

Figure 2. Computational Model of D-CCFinder

number of clones is expected to be detected[1], thus we need some abstraction of the output in order to grasp the overall characteristics.

The output format of D-CCFinder is a list of code clone sets, where each set is composed of a number of unique code fragments. This information should be abstracted into project or category level information, for example by computing the code clone coverage between each pair of projects or categories. Also, we might simply want to see the existence of any clones in each project or category. We will do this kind of work in the post processing of the output of D-CCFinder.

2.3. Computational Model of D-CCFinder

By default, CCFinder finds clones among all the files indicated in an input list. Since all the files need to be loaded into memory space, the execution of CCFinder is aborted if the target is large and there is not enough memory space.

For resolving this issue, we partition the target into not-intersecting pieces whose size is acceptable to CCFinder. By giving an option, CCFinder can take two lists of input files, and produce clone lists only between the listed files. The execution of CCFinder over a piece of the target is called *task*. The outputs of each task are virtually merged so that we can obtain all the clones in the target.

The computation of each task is independent in the sense that intermediate and final result of each task does not affect the computation of other tasks; therefore the problem of segmenting the code-clone analysis becomes an *embarrassingly parallel* problem[21].

[1]It is reported that about 5-20% of total lines are clones in [20].

As shown in Figure 2, assume that the target size is nu where n is the number of the partition, and u is the size of one unit. Each piece is indexed by (i, j) where $1 \leq i, j \leq n$. Because the code clones of piece (i, j) are the same of piece (j, i) we can reduce the number of tasks to be performed almost to half. The total and exact number N of tasks is given by:

$$N = \frac{n(n+1)}{2}$$

Through D-CCFinder, these tasks are distributed over a multiple CPU environment where an ordinary CCFinder is executed on each CPU. Since no interaction is needed, the scheduling of the tasks becomes very simple: we will assign any task to any idle CPU with appropriate input, and collect the output.

3. Implementation of D-CCFinder

D-CCFinder is a distributed system for code clone analysis based on CCFinder. Architecturally it is a master-slave application with single CCFinder jobs executed on slave nodes. D-CCFinder has been implemented in the Java language using Java RMI for message passing between the nodes, while the source repository resided on a network file-system, to which all the nodes had access.

D-CCFinder is integrated by a small set of utilities for the pre-processing of the input source code, the post-processing of the output data and image generation. A diagram of the whole system can be seen in Figure 3.

Indexer. The target is examined and informations about file size and number of lines of code, project name and project's category are stored in index files. Unit boundaries are also computed by the indexer. In the following experiments, we used 15 MBytes as the unit size, a value far smaller than the ideal case, because of the limitations of the hardware in the student laboratory. We got a total of 734 units and 269,745 tasks for the FreeBSD target.

Master node. The master node creates the input files for CCFinder and assigns tasks to the slave nodes. If a slave node fails, the task is assigned to a different node. If the task fails because of some CCFinder's internal error the file list of the task is kept for further investigation.

Slave node. We had 80 slave nodes executing CCFinder on each piece of the target. The files are first copied to the local node storage, and then analyzed. The text output of CCFinder is processed in order to remove redundancy and uninteresting clones consisting in simple repeated patterns:

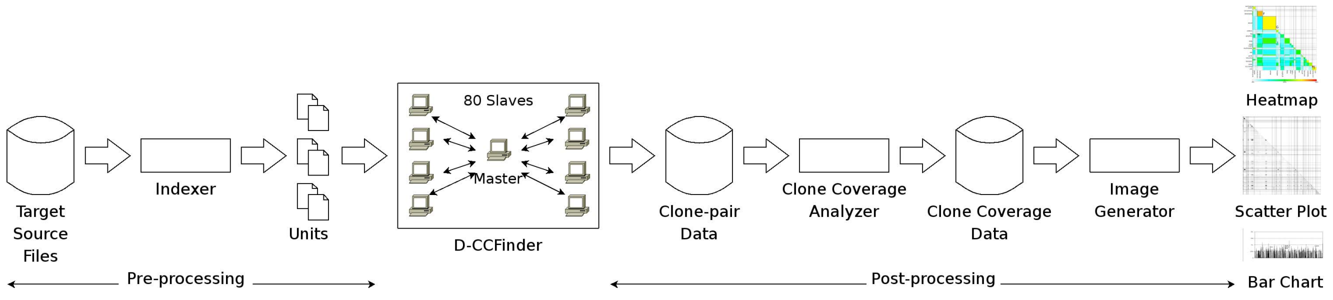

Figure 3. Process Overview for Code Clone Analysis using D-CCFinder

Table 1. Characteristics of the Master and Slave Nodes

Processor	Pentium IV 3GHz
Memory	1 GBytes
Network Link	Gigabit Ethernet connected to 100Mbit/s network hubs
OS	FreeBSD 5.3-STABLE
Local storage	40 $\sim$ 50 GBytes

the Repeated Token Ratio (RNR) metrics has been used to perform the filtering[9] (in our experiments we set the threshold value for RNR to 0.5). The final result is moved to the shared network file-system.

CCFinder's output consists of code fragment pairs. Each code fragment contains the index of the file it belongs to, and this index is unique with respect to the files processed. In order to have indexes consistent across the whole target code fragments re-indexed.

The characteristic of the the master and slave nodes are shown in Table 1.

Clone Coverage analyzer: The output of D-CCFinder is processed and data about the number of shared lines of code is computed for each pair of project, file and category. This data are used to generate three different types of graph: scatter plots, heatmaps and bar charts.

Image generator. For the purpose of visualization the user can choose between the generation of scatter plot or heatmap, and bar chart. Heatmaps of any portion of the target can be created at file level, while a global view is generated at project and category level. The generation of a heatmap at file level allows the user to specify a scale factor, reverting to the generation of a 2-color black-and-white scatter plot if the scale value is greater than 1.

4. Experiments

We performed two experiments with D-CCFinder. In the first experiment, we analyzed the whole FreeBSD target. In the second experiment, we used D-CCFinder to find clones between our research system, and the FreeBSD target.

We define *code clone coverage* ($Coverage_{M_0M_1}$ and $Coverage_{M_0}$) to measure the percentage of clones between two files, projects or categories as follows:

$$Coverage_{M_0M_1} = \frac{LOC(CC(M_0,M_1))}{LOC(M_0)+LOC(M_1)} \times 100$$

$$Coverage_{M_0} = \frac{LOC(CC(M_0,M_1)\cap M_0)}{LOC(M_0)} \times 100$$

with:
M_0, M_1: a pair of files, projects or categories;
$CC(M_0, M_1)$: segments of the code clones between M_0 and M_1;
$LOC(x)$: the total number of lines of code in x.
$Coverage_{M_0M_1}$: the ratio of code clone fragments against the total of M_0 and M_1;
$Coverage_{M_0}$: the ratio of code clone fragments between M_0 and M_1 contained in M_0 against M_0.

4.1. The FreeBSD Target

There are many collections of open source projects, such as SourceForge (http://sourceforge.net/) or Gnu (http://www.gnu.org/). Among them we have chosen the "Packages and Ports Collection" of FreeBSD as the target, since it is well maintained and it is already partitioned into categories.

The size characteristic of the FreeBSD target and its category names are shown in Table 2 and Table 3 respectively.

The FreeBSD target sometimes includes several versions of the same project, for example versions 1.3, 2.0, 2.1, 2.2 of the Apache web server. This is because the older systems are sometimes needed by the users or other systems for

Table 2. Characteristics of the FreeBSD target

Number of categories	45
Number of projects	6,658
Number of .c files	754,552
Total lines of code	403,625,067
Total size	10.8 GBytes

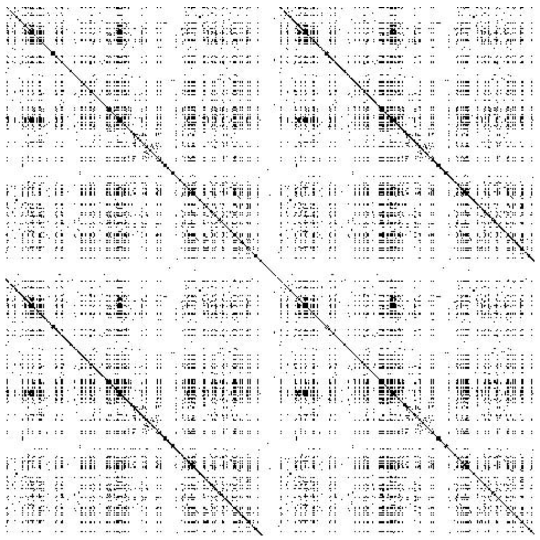

Figure 5. Diagonal Pattern from Area E of Figure 4

Table 3. Categories in the FreeBSD target

Index	Name	Index	Name
1	accessibility	24	math
2	arabic	25	mbone
3	archivers	26	misc
4	astro	27	multimedia
5	audio	28	net-im
6	benchmarks	29	net-mgmt
7	biology	30	net-p2p
8	cad	31	net
9	comms	32	news
10	converters	33	palm
11	databases	34	polish
12	deskutils	35	print
13	devel	36	science
14	dns	37	security
15	editors	38	shells
16	emulators	39	sysutils
17	finance	40	textproc
18	ftp	41	www
19	graphics	42	x11-clocks
20	irc	43	x11-fm
21	java	44	x11-fonts
22	lang	45	x11
23	mail		

backward compatibility. In such cases, many code clones across the versions are expected to be found.

4.2. Inter-Project Analysis for FreeBSD Target

We ran D-CCFinder and all associated tools, with a detectable minimum token length of 50 tokens. Since we chose a unit size of 15MBytes the number of tasks to be executed was 269,745. Figure 4 shows the resulting scatter plot. This figure has been generated using a scale factor of 200, hence each dot corresponds to a unit of 200x200 files. To accelerate the image generation process a dot is painted black if at least one pair of files shares one or more code portion, i.e., if D-CCFinder detected at least one code clone pair between those two files. This visualization method produces a fairly exaggerated view of the code clones present in the target. However, since the average $Coverage_{M_0 M_1}$ for the target is about 4%, we would get an almost-white scatter plot without using such a method.

The most peculiar feature of the diagram in Figure 4 is the presence of distinguishable artifacts: A, B, C and D are some examples.

A closer examination of the marked areas put in evidence a common repeated pattern; one example of this pattern is Area E, and it is closely shown in Figure 5, and manual investigation of the source code exposed the presence of sets of files that appear unchanged in multiple projects. This is due to the particular structure of the FreeBSD's port collection: different but related projects (for example plug-ins for the multimedia framework gstreamer) contains identical copies of a common set of source files. Being more specific:

Area A. Stripes contained in this area show how the source trees of php4 and php5 are used in various projects and in different categories.

Ares B. Enclosed in this area are four categories containing what the FreeBSD port maintainers have classified as x11-related software. Multiple copies of the core source tree of the X Window System are found in these categories.

Area C. Imake is a make-like build tool that's part of the X Window System. It has been classified as a member of the devel category and it's clearly visible that this project's source code contains copies of the source code of Area B.

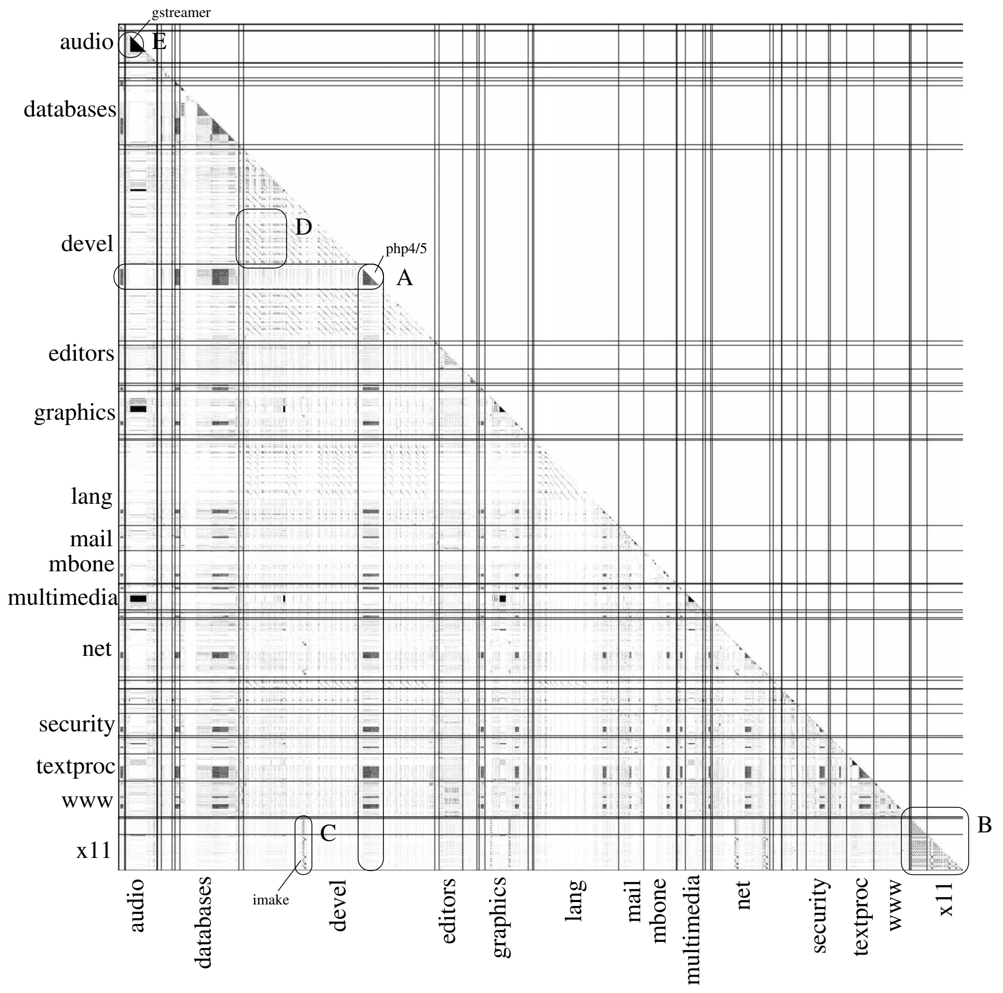

Figure 4. Scatter Plot of Inter-Project Code Clone Coverage for the FreeBSD Target

Area D. Most of the devel category exhibits a marked diagonal pattern. It is mostly due to the presence of multiple copies of the source code of the binutils software for different architectures.

Area E. Category audio contains the source code of the multimedia engine gstreamer's plug-ins, and the main tree of gstreamer is duplicated inside each project.

Most of the evident artifacts of Figure 4 are of the type previously mentioned: a diagonal pattern showing a 100% $Coverage_{M_0 M_1}$. It is not easy to detect a uniquely existing code share between only two projects by mere observation because of the approximation of our visualization method.

A second run of D-CCFinder with a minimum detectable token length of 200 tokens yielded a diagram presenting almost the same main artifacts.

Figure 6 shows a heatmap of the $Coverage_{M_0 M_1}$ between the categories of the FreeBSD target. As expected most of the highest $Coverage_{M_0 M_1}$ values lie on the diagonal, though values greater than 25% are not uncommon between different categories. Some interesting parts of Figure 6 have been marked.

Area F. The databases category has a $Coverage_{M_0 M_1}$ of 41%. This value can be ascribed to two factors: the presence of different versions of the same software system, and the presence of database drivers for languages as ruby and php. In the former case a substantial number of code clones is expected, in the latter, manual inspection, revealed the presence of multiple copies of the same source tree.

Area G. The $Coverage_{M_0 M_1}$ value for the devel category is 38%. The presence of different versions of the suite of GNU binary utilities and compilers for different architectures is the main reason of this high value.

Area H. Categories ftp and converters show a code clone

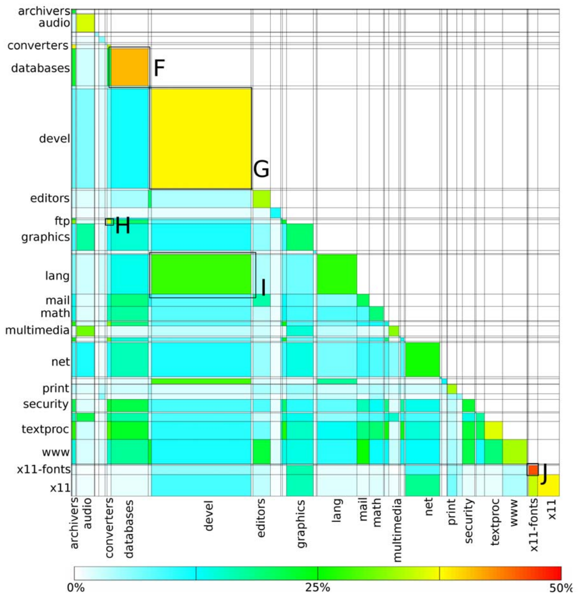

Figure 6. Color heatmap for the code clone coverage of the FreeBSD target (category view)

coverage of 37%. Manual inspection revealed that the presence of multiple copies of the php4 and php5 source tree can be accounted of the high value.

Area I. The $Coverage_{M_0 M_1}$ value for the categories lang and devel is 28%. This value can be ascribed to the devel category containing multiple versions of the suite of GNU compilers; and this code being mirrored inside category lang.

Area J. Category x11-fonts presents the highest $Coverage_{M_0 M_1}$ value: 46%. This is due to the relatively small size of the category, and the presence of seven copies of the source tree of the core of the X Window System

4.3. Single Project Analysis of SPARS-J and the FreeBSD Target

Recently, in parallel with the increased availability of open source programs for a wide variety of purposes, the probability of including, intentionally or not, part of their source code into new systems also increased; consequently, the risk of violating the terms of open sources licenses is emerging.

We used D-CCFinder to detect the presence of open source code fragments imported into the SPARS-J system which is a Java component analysis and search system developed at our laboratory[11]. SPARS-J is mostly written in C and has about 47,000 lines of code. We have examined how SPARS-J shares its code with the FreeBSD target. In this case, a single piece executed as a task contained the 47,000 lines of code of SPARS-J plus 15 MBytes of the FreeBSD target, and we had 734 tasks in total.

Table 4. Time elapsed

Indexer	22 minutes
D-CCFinder	51 hours
Scatter plot	
Clone Coverage Analyzer	23 hours
Image Generator	4 hours
Heat map	
Clone Coverage Analyzer	70 hours
Image Generator	2 minutes

Figure 7 shows the result of the analysis as a bar chart; each bar indicates the $Coverage_{M_0}$ value between SPARS-J and a single project in the FreeBSD target, i.e. the percentage of source code that SPARS-J shares with that project.

Upon the inspection of some of the projects with which SPARS-J shows the highest ratio of shared code we found that most of the code clones were from a single file (getopt.c) containing code for parsing command line options.

Further examination revealed a large number of projects in the FreeBSD target making use of this file, thus, in order to have a more readable and less dense diagram, these projects have been removed from the bar chart. Figure 8 is the resulting diagram. It is possible to see how SPARS-J shares very little code with the projects in the FreeBSD target (except for the code in getopt.c), but there are a few noteworthy cases, in which the $Coverage_{M_0}$ is greater tha 0.5%, that is, SPARS-J is sharing more than 200 lines of code.

A complete analysis of those projects revealed that SPARS-J uses code from two projects in the FreeBSD target for the handling of CGI requests, and that copies of the previously mentioned library for the parsing of command line options exist with different file names and in a specialized forms.

5. Discussion

The case studies presented show how D-CCFinder can be used to detect similarities in a large source repository, and between the whole repository and a single external project.

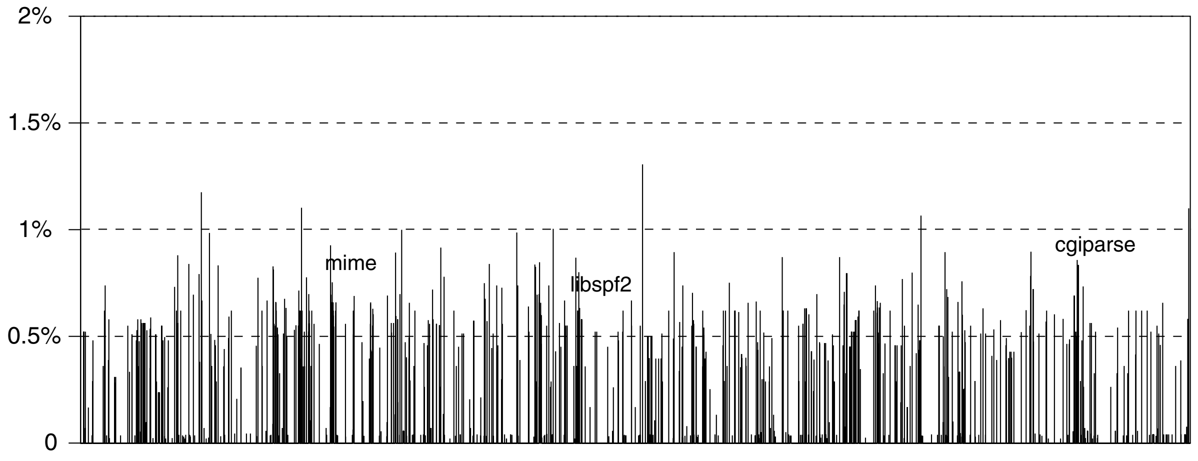

Figure 7. Code Clone Coverage for Project SPARS-J (before filtering)

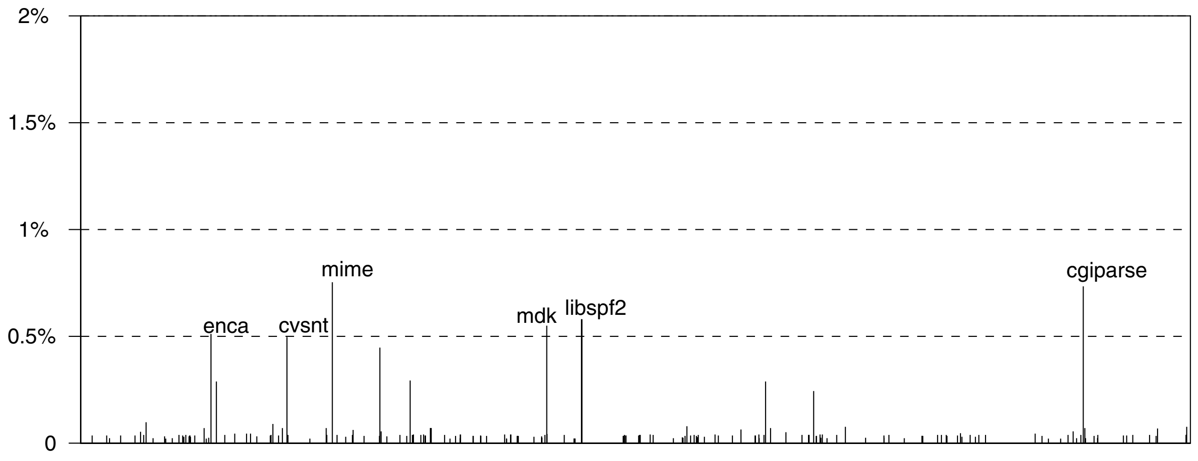

Figure 8. Code Clone Coverage for Project SPARS-J (after filtering)

Experience with the FreeBSD target. We performed a code clone analysis on the source code composing the FreeBSD ports collection.

As listed in Table 4, it took about 51 hours to execute the task with a cluster of 80 computers, a minimum token length of 50 tokens, and a unit size of 15 MBytes. Clone Coverage Analyzer and Image Generator were on a single 2.8 GHz Xeon PC-based workstation with 4 GB of memory.

In the ideal case, the acceleration gain should be 80 and the expected running time should be about 12 hours, but network congestion, master-slave synchronization, and output processing performed at each slave node increased the total time. Nonetheless, our approach was 20 times faster (i.e., the actual acceleration ratio is 20) than it would using a single workstation capable of analyzing the whole target. The introduction of faster network equipments (for instance, Gigabit hubs) in the student laboratory, would probably decrease the overhead and consequently increasing the acceleration gain.

The necessity to generate a scatter-plot of reasonable size but still with visual clues about the distribution of the code clones for a very large data set had us trading off accuracy for speed and space: the method used enhanced the presence of artifacts, making the organization of the source repository immediately visible, but at the same time it made impossible to detect finer details and relationships between small projects.

We need to explore other visualization methods for other analysis objectives. In the current implementation, Clone Coverage Analyzer requires a fairly long computation time. This could be reduced using the same distributed computation model that D-CCFinder employs. We will need to design an algorithm for this purpose.

SPARS-J and the FreeBSD target. Detecting the code clones between SPARS-J, and the whole FreeBSD target took about 40 minutes with a cluster of 80 computers. The output have been visualized as a bar chart. After filtering the obtained data we were able to effortlessly individuate a small set of files containing duplicate code. We believe that this use of D-CCFinder can be applied to the detection of illegal or unintentional use of copyrighted source code in an arbitrary software project.

Limitations. One limitation of the current implementation of D-CCFinder is the inability to efficiently filter unnecessary or uninteresting clones, such as simply repeated files, in order to reduce the data to be examined. While it would be possible to realize this filtering running an additional processing stage after the execution of CCFinder on each slave node, we believe that performing it during the clone detection phase would be more efficient. In the near future we are going to experiment other ways to perform code clone detection, particularly fingerprint based analysis. We will also parallelize both the post-processing phase and the image generation task.

Another limitation is implicit in the method used to generate the scatter-plot: the employed algorithm degrades the results to those obtainable with a mere file comparison. We are addressing this issue researching new ways for visualizing the results.

6. Related work

The concept of very-large size code clone analysis relates to the notion of Mega Software Engineering[10]. Mega Software Engineering refers to a collection of software engineering technologies that, supporting the large scale analysis of data from many projects, enables process improvement at organization level, rather than at single project level. With the decreasing cost of computer hardware, and its concurrent increment of performance, the notion of Mega Software Engineering becomes very important, and it is expected to see more frequent applications of its techniques, methodologies and practices in the field of software engineering research.

Given the total size of the source code to be analyzed, the only viable option was the parallelization of the task. The availability of the computers of our department's student laboratory lead us to organize a network connected cluster in order to solve the problem. Because the input and the output of the tasks are generally large and the computational model is trivial, we opted for a straightforward master-slave implementation using a shared network file-system, rather than embracing more sophisticated approaches at parallel computing, such as grids[8].

Various approaches to code clone analysis have been proposed: Baxter et al. illustrate a method for detecting code clones through the use of abstract syntax tree[2]; Ducasse et al. propose string matching[7]; Kamiya et al. suggest the use of prefix tree[12]; Krinke uses program dependence graphs[16].

Our familiarity with CCFinder[12] and its immediate availability, made us prefer this tool over others with similar characteristics of performance and scalability. We don't think that CCFinder is the only tool usable for this kind of analysis, on the contrary we believe that a finger-print based approach[17] can be very efficient if we want to perform some fine level filtering of the source code in order to remove unwanted code clones.

Previous works on the analysis of open source software have been done[6, 20] and various characteristics of open source software have been presented; however, those techniques lack the scalability our approach has, and they aren't able to show the inter-project analysis we proposed in this paper.

Blackduck Software's protexIP/development[4] Software Compliance Management system uses file and code snippet analysis to identify the use of code from third-party and open-source projects and detect licence conflicts.

D-CCFinder is being also used by the authors to perform a study of the evolution of the Linux kernel.

7. Conclusion

We have proposed a novel approach to distributed large scale code clone analysis. This approach has been implemented in our prototype D-CCFinder, and it has been applied to a vast collection of open source programs. We obtained a global overview of code clones among these programs, and the use of the same source code in multiple projects was visible as easy recognizable artifacts. We also analyzed a single project against the same collection, and it was possible to effortlessly individuate the use of code from software in the collection.

One of the objectives of this paper was the exploration of the application of a distributed approach at software engineering in the context of Mega Software Engineering. We believe that D-CCFinder illustrates a fairly cheap and practical method for large scale code clone analysis employing commodity hardware, freely available software, an already existing network infrastructure, and a trivial but efficient implementation.

D-CCFinder is our first experiment at large scale code clone analysis, hence it is incomplete and shows many weak points. In the near future we will explore the use of fingerprinting and data mining algorithm for the purpose of code clone analysis. We believe that this alternative approach could offer an improvement of performance.

Acknowledgements

The authors would like to thanks Toshihiro Kamiya in the National Institute of Advanced Industrial Science and Technology for his comments on the use CCFinder, and Shigeto Tajima and Fumihiro Koizumi of the Osaka University Information Science and Technology Department for the help and support provided with the implementation of D-CCFinder at the student laboratory. This work has been conducted as a part of EASE Project, Comprehensive Development of e-Society Foundation Software Program, and Grant-in-Aid for Exploratory Research(186500006), both supported by Ministry of Education, Culture, Sports, Science and Technology of Japan. Also it has been performed under Grant-in-Aid for Scientific Research (A)(17200001) supported by Japan Society for the Promotion of Science.

References

[1] B. S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 24:49–57, 1992.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of International Conference on Software Maintenance '98*, pages 368–377, Bethesda, Maryland, March 1998.

[3] S. Bellon and R. Koschke. A comparison of automatic techniques for the detection of duplicated code. Technical report, Institute for Software Technology, University of Stuttgart, 2003. available at `http://www.bauhaus-stuttgart.de/clones`.

[4] Blackduck Software. ProtexIP/development: Software compliance management system. `http://www.blackducksoftware.com /products/_protexip.html`.

[5] A. W. Brown and G. Booch. Reusing open-source software and practices: The impact of open-source on commercial vendors. In *Proc. of the 7th International Conference on Software Reuse*, volume 2319 of *Lecture Notes in Computer Science*, pages 123–136, Austin, Texas, April 2002. Springer.

[6] G. Casazza, G. Antoniol, U. Villano, E. Merlo, and M. D. Penta. Identifying clones in the linux kernel. In *Proc. of the First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 92–100, Florence, Italy, 2001. IEEE Computer Society Press.

[7] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. of the International Conference on Software Maintenance '99*, pages 109–118, Oxford, England, August 1999.

[8] I. Foster. What is the grid? a three point checklist, July 2002. available at `http://www-fp.mcs.anl.gov/ ~foster/Articles/WhatIsTheGrid.pdf`.

[9] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Method and implementation for investigating code clones in a software system. Submitted to Information and Software Technology.

[10] K. Inoue, P. Garg, H. Iida, K. Matsumoto, and K. Torii. Mega software engineering. In *Proc. of the 6th International PROFES (Product Focused Software Process Improvement) Conference*, pages 399–413, Oulu, Finland, 2005.

[11] K. Inoue, R. Yokomori, T. Yamamoto, M. Matusita, and S. Kusumoto. Ranking significance of software components based on relations. *IEEE Transaction on Software Engineering*, 31(3):213–225, April 2005.

[12] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002.

[13] C. Kapser and M. Godfrey. Improved tool support for the investigation of duplication in software. In *Proc. of the 21st International Conference on Software Maintenance*, pages 25–30, Budapest, Hungary, September 2005.

[14] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *Proc. of the 10th European software engineering conference*, pages 187–196, Lisbon, Portugal, 2005.

[15] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proc. of the 8th International Symposium on Static Analysis*, pages 40–56, Paris, France, July 2001.

[16] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. of the 8th Working Conference on Reverse Engineering*, pages 301–309, Suttgart, Germany, October 2001.

[17] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transaction on Software Engineering*, 32(3):176–192, March 2006.

[18] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proc. of the International Conference on Software Maintenance '96*, pages 244–253, Monterey, California, November 1996.

[19] D. C. Rajapakse and S. Jarzabek. An investigation of cloning in web applications. In *Proc. of the 5th International Conference on Web Engineering (ICWE 2005)*, Lecture Notes in Computer Science, pages 252–262, Sydney, Australia, 2005. Springer.

[20] S. Uchida, A. Monden, N. Ohsugi, T. Kamiya, K. Matsumoto, and H. Kudo. Software analysis by code clones in open source software. *The Journal of Computer Information Systems*, XLV(3):1–11, April 2005.

[21] B. Wilkinson, M. Allen, and C. M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 2004.

[22] T. Yamamoto, M. Matsushita, T. Kamiya, and K. Inoue. Measuring similarity of large software systems based on source code correspondence. In *Proc. of the 6th International PROFES (Product Focused Software Process Improvement) Conference*, pages 530–544, Oulu, Finland, 2005.