Clone detection by exploiting assembler

Ian J. Davis and Michael W. Godfrey
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
{ijdavis, migod}@uwaterloo.ca

ABSTRACT

In this position paper, we describe work-in-progress in detecting source code clones by means of analyzing and comparing the assembler that is produced when the source code is compiled.

Categories and Subject Descriptors

D.2.7 [Maintenance and Enhancement]

Restructuring, reverse engineering and reengineering.

General Terms

Algorithms, Measurement, Documentation, Reliability.

Keywords

Clone Detection, Software, Java, C, C++, Assembler.

1. INTRODUCTION

Source code clone detection aims to discover similar code fragments within different parts of a given code base, either so that such source code might be consolidated (making the resulting code both tighter and cleaner) [2] or so that the strong similarities between code in diverse parts of a source code might at least be recognized and documented [3].[1] There are a variety of approaches that have been used to analyze source code for clones, including string-, token-, and AST-based techniques.

In this work, we are taking a very different approach. We are developing tools that compile C, C++, and Java to assembler [6, 7], and then perform clone detection on the resulting stream of assembler instructions contained within functions. (Previous work by Baker and Manber [1] and Norman [4] have explored clone analysis of Java bytecode.)

IWSC2010 May 8, 2010, Cape Town, South Africa

While performing clone detection on assembler might seem a surprising strategy, we feel that that if offers several advantages that can complement source code-based techniques. First, while assembler-based clone detection tools are obviously tied to a particular assembly language, such as 32-bit GNU assembler or the JVM, they are also independent of the programming language; consequently, few if any programming language specific tools need be developed, which simplifies the creation of tools. Second, clone detection may also be performed on object code for which source code is lacking. Third, such tools obviously cope well with code that already has assembler embedded in it, which is traditionally hard for source code analyzers to deal with. Fourth, we note that compilation often serves to normalize syntactic variants of source into a compact canonical representation; in principle, this makes it easier to spot semantically equivalent (or merely similar) constructs that may ``look different'' when expressed as source code. Finally, analyzing assembler for clones provides a new perspective on this clone analysis, and may well yield surprising insights.

There are also disadvantages to this kind of approach. First, obviously any source code to be analyzed must be compilable; string- and token-based source code approaches do not make this requirement. Second, some compilers (notably not Java) generate assembler languages that are platform specific, while different compilers for the same machine might produce very different assembler. Similarly, different compilation settings (such as optimization level) from the same compiler may produce widely varying results.

2. PERFORMING CLONE DETECTION

After mimicking the build process to generate the relevant assembler, the sequence of assembler instructions associated with each function is read. Symbolic debug information is used to add variable names to the assembler, and to associate assembler instructions with the source from which they are derived.

Our detection algorithm aims to discover a maximal pairing of distinct matched assembler instructions from two distinct assembler subsequences (contained within a function) which occur in the same sequence, subject to some limit being imposed on the interleaved assembler instructions that do not match. In addition matched branches must address assembler instructions that do not occur on opposite sides of any matched pairing.

For each possible pairing of matched instructions (not yet identified as matched within any clone) a greedy algorithm is used to find subsequent matches. This associates a positive weight with matches and a negative weight with mismatches and terminates when the collective weight becomes negative. Then a hill-climbing algorithm is used to improve the number of matches within this clone. This logic iterates while improvement results.

When presented with longer sequences of instructions that match, our algorithm becomes correspondingly more forgiving of later instructions that do not. Modification of the parameterized weights permits exact matching of sub-sequences to be enforced, or more tolerance of mismatches.

3. REPORTING OF CLONES

Detected clones pairs are reported if the sequence of assembler instructions forming a clone has some user-specified minimum length, or spans an entire function having some user-specified minimum length.

Our reporting technique is to build HTML pages, which permit the various clones discovered to be easily navigated and explored. These pages describe each discovered clone pairing. They shows the source file name and line number from which the assembler is derived, as well as the contents of that line, and uses DHTML to permit optional hiding of either the source or assembler instructions involved in this pairing. A separate page highlights the longest clone sequences discovered. To permit visualization of the distribution of clones, these may also be viewed graphically [8].

4. SUMMARY

We are exploring a novel technique for performing clone detection on assembly language generated from source code; this complements source-based techniques, such as the token-based approach used by CLICS [5] and others. This approach permits one to perform clone detection on precisely those source files actively used in a build process, and to analyze source on which all the normal preprocessing involving macro and template expansions and header file inclusion have been performed. However, since our software examines only assembler instructions, it does not detect similarities in data contents within tables, macro definitions, or in sections of code that declare variables. Also, we have not yet enhanced our software so that it is sensitive to name changes that sometimes occur as consequence of code cloning.

Our initial investigations, which we do not detail here due to lack of space, suggest that our clone detection software is effective, forgiving of mismatched assembler code within clones, reasonably fast, and produces comprehensive results that can be viewed through a web browser, as well as a graphical visualization tool. While it must be customized for each assembler that it is to read, this is a one time cost. Our ongoing work includes a detailed comparison with the results of source-based approaches, grounded in case studies of large open source software systems.

Source for our tools are publicly available [6, 7].

5. ACKNOWLEDGMENTS

This research is supported by research grants from CA Canada Inc, OCE, and NSERC.

6. REFERENCES

[1] Baker, B. S. and Manber, U. Deducing similarities in Java source from bytecodes. *Proc. USENIX Annual Technical Conference. New Orleans, Louisiana, June 15-19, 1998.*

[2] De Sutter, B., De Bus B., De Bosschere, K. Link Time Binary Rewriting Techniques for Program Compaction. *ACM Transactions on Programming Languages and Systems,* 27(5) September 2005

[3] Kapser, C. J and Godfrey, M. W. Supporting the Analysis of Clones in Software Systems: A Case Study. *Journal of Software Maintenance and Evolution: Research and Practice.* 18(2), March 2006.

[4] Norman, M. Clone Detection applied to Java Bytecode. http://www.wcs.carleton.ca/~deugo/comp5900/papers/MNcomp5900Paper.pdf 10 December 2008

[5] SWAG: Software Architecture Group. CLICS clone detection tool. http://www.swag.uwaterloo.ca/clics

[6] SWAG: Software Architecture Group. JCD Java Clone Detector. http://www.swag.uwaterloo.ca/jcd

[7] SWAG: Software Architecture Group. ACD C, C++, Assembler Clone Detector. http://www.swag.uwaterloo.ca/acd

[8] Syntskyy, N., Holt, R. C., Davis, I. J. Browsing Software Architecture with LSEdit. *Proc. of Intl. Workshop on Program Comprehension.* 15 May 2005.