YACCA: Code Clone Detection on Multi-core Processors

Simone Livieri, Katsuro Inoue
Graduate School of Information Science and Techonology
Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
{simone, inoue}@ist.osaka-u.ac.jp

Abstract

Code clone detection is a mature topic in software engineering research and, over the past decades, many methods and tools have been proposed. Detection algorithms have been continuously refined but tool's implementations often fail to exercise the full potential of contemporary hardware. In this paper, we propose to fill this gap, and present our new code clone detection tool.

1. Introduction

In the last decades, code clones, source code fragments resembling each other, have attracted much attention from the research community. As result, over the years, more and more refined methods and tools for detecting and analysing code clones have been proposed[1, 5, 6].

Recently, urged by the advancements in manufacturing technology that made available and common processors integrating multiple computational cores, it has been argued that software, not only hardware, should follow Moore's law and double its parallelism every two years to fully take advantage of recent progress in personal computing power[3].

At present, at least to the knowledge of the authors, there is no tool for code clone detection that takes full advantage of the new multi-core processors.

In [7], we have shown how code clone detection can be considered an *embarrassingly parallel* problem – a problem than can be trivially decomposed into several parallel and independent sub-problems – and presented a tool called `D-CCFinder`, implemented to distribute the code clone analysis of very large source code repositories over a cluster of several machines.

The tool we present in this position paper stems from the work done on `D-CCFinder` and wants to bridge the gap between the available technology and the current state-of-art in token-based code clone detection.

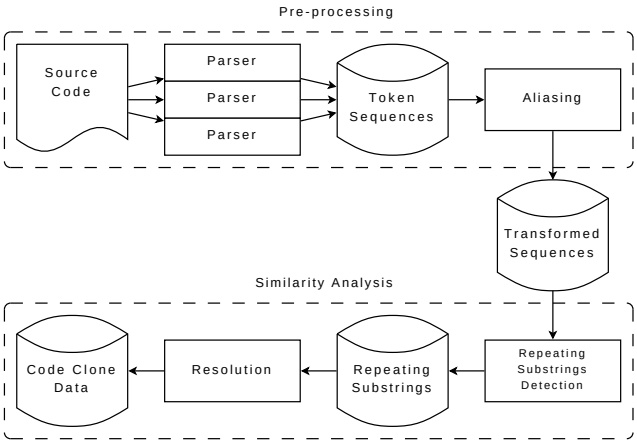

Figure 1. YACCA: System's data flow

2. The Tool

We propose YACCA, a large-scale code clone detection tool aimed at taking full advantage of modern multi-core processors. Its main characteristics are summarised in the following three points:

1. **Multi-core processor support.** YACCA leverages the recently introduced multi-core processors, evenly distributing the total workload between the cores.
2. **Configurable detection.** YACCA uses a parameterized detection algorithm that permit to detect code segments with various degrees of similarity.
3. **Language independence.** YACCA is able to process files written in different languages and, to a certain extent, to detect cross-language code clones.

3. The Method

In this section we briefly describe our tool. Figure 1 shows the tool's internal data-flow.

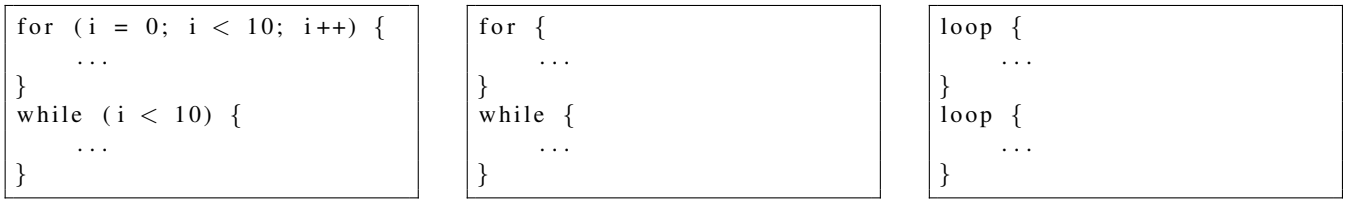

Figure 2. Different levels of *aliasing* on two loop statements.

3.1. Source Code Parsing

Source code files are parsed into lexical token sequences including data on tokens' original positions and their associated character sequences.

Because code clones are most useful when they are contained in syntactical units, the token sequences are split into smaller sequences corresponding to specific syntactical unit bodies (methods, functions, etc.). Since our tool is based on the detection of repeating substrings, having shorter sequences gives an upper limit to the maximum code clone's length and consequentially hastens the detection process.

3.2. Duplicated Code Detection

Detecting duplicated code involves three-steps.

1. **Aliasing.** The token sequences are transformed introducing some ambiguities (*aliasing*) that increase the sequence's similarity while keeping intact the overall code structure (e.g., by replacing the control block of a *for*-loop or a method name with a special token, see Figure 2). Not essential code – as simple assignment statements – can be removed during this step.
2. **Repeating Substrings Detection.** All the repeating substrings in the token sequences are computed using a parallel algorithm and divided into groups. The algorithm used is straightforward and finds repeating substrings by progressive refinements. More sophisticated and faster algorithms exist but they require more space for their executions. For example, Crochemore's algorithm [2] in its general implementation requires at least $80n$ bytes of storage for a string of length n. In order to process larger amount of source code, we chose a simpler algorithm requiring less space. Because each refinement is independent from each other, the algorithm has been easily adapted for parallel execution.
3. **Resolution.** The detected substring are grouped into primitive code clone sets[4]. Each clone set is further refined by analysing identifier's appearance order[6, 5].

4. Impact on Global Development

Software companies tend to outsource part of their development to other companies to reduce costs and shorten time-to-market. This practice carries two main inherent risks: first, different teams can create similar sets of routines because of lack of team communications; second, deadline pressure can lead to reuse source code released under copyright terms conflicting with those of the final product. Both of these problems can be tackled with the use of code clone detection: in the first case, code clone detection can be used to detect similarities between source code produced by different teams, and refactor them into shared libraries; in the second case, code-clone detection can be used to detect the abuse of third-party source code, as better described in [7].

Acknowledgements

This work has been conducted as a part of StagE Project, the Development of Next Generation IT Infrastructure, 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] 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.

[2] M. Crochemore. An optimal algorithm to compute all the repetitions in a word. *IPL*, 12(5):244–248, 1981.

[3] I. Fried. Intel: Software needs to heed moore's law. Website, May 2007.

[4] 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.

[5] 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.

[6] 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.

[7] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 106–115, 2007.