CS 6240: Software Engineering Project Report (Fall 2016) Scaling Code Clone Detection on the Automata Processor

Elaheh Sadredini, Evaristo Koyama, Baishakhi Ray (Instructor)
Department of Computer Science
University of Virginia
Charlottesville, VA 22904
{es9bt, ek4ks, br8jr}@virginia.edu

ABSTRACT

There exists many tools for detecting code clones, but very few that work with large codebases. Many tools suffer from excessive memory usage and long execution time. SourcererCC is one such tool that avoids those problems. Performance of SourcererCC on large codebases have shown to be reasonably well with acceptable accuracy, but there is still room for improvement. For this project, we exploit the Automata Processor, in order to provide a scalable code clone detection. We find the AP can significantly improve performance using SourcererCC's algorithm without losing any precision and recall.

1. INTRODUCTION

Code cloning is a common practice done by many software developers. It is a quick way to resolve any small and immediate problems, but they usually come with their own issues. Maintenance can be difficult and costly as developers need to keep track and update all clones of a changed code fragment [1]. Bugs can propagate when a programmer copies a buggy section or forgets to edit the section to match the surrounding code. Code cloning can also increase the size of software, which can strain resources and degrade performance [2]. As software grows, the aforementioned issues can seriously deteriorate software quality and make maintenance prohibitively expensive.

To help manage growing codebases, a scalable code clone detection tool can be used. With such a tool, refactoring efforts can be improved and can minimize the consequences of code cloning. It can also improve bug detection. By keeping track of clones, a programmer can quickly correct any buggy clones and check whether the clone is adapted correctly. Programmers can also use efficient scalable code clone detector to run more accurate, but slower clone detector on a subset of the code. Techniques for scalable code clone detection can be used for other applications with large datasets, such as plagiarism detection and find function.

Most of the existing state-of-the-art solutions have difficulty with large databases, and fail due to scalability limits [3] [4]. Common limits include untenable execution time, insufficient system memory, limitations in internal data structures, or crashing or reporting an error due to their design not anticipating such a large input.

The main goal of the project is provide an efficient and scalable clone detection solution on the newly introduced hardware accelerator, the Automata Processor (AP). The Automata Processor is a novel hardware implementation of the non-deterministic finite automata, which makes it potential to be a candidate target hardware accelerator for the pattern matching tasks. The AP excels in implementing complex regular expressions where up to several thousands of them can be checked against the input in parallel, while memory bottleneck and memory footprint are the main two burdens in implementing regular expressions (regex) on the conventional central processing units (CPUs) [5].

2. RELATED WORK

Difficulties in detecting clones can vary greatly. There are dependent on the structures of the code fragment. These clone structures are categorized into 4 different types [2]. Type-1 clones are code fragments that only differ in formatting, such as annotations, whitespace, and comments. These clones are the easiest to detect and verify as code fragments are identical. Type-2 clones are code fragments that can also differ in naming. Type-3 clones are code fragments that can have additional, reordered, or deleted statements. These two types are the most common pattern, since programmers typically modify related code fragments [6]. Type-4 clones are code fragments that are semantically the same, but can be entirely different algorithm. A common example is the iterative and recursive versions for calculating the Fibonacci sequence. Clone detection efforts have primarily focused on the first three types.

Different techniques have been proposed for clone detection [7] [8]. Each technique have their own advantages and disadvantages in factors such as precision, recall, performance, clone size, and scalability. They are typically categorized into 5 types, string-based, token-based, tree-based, graph-based, metric-based, with some taking a hybridized approach. String-based techniques directly analyzes the text in the source code. It is usually the simplest and quickest technique, but can have difficulties detecting clone of types other than 1. Token-based techniques takes the code and generates tokens to analyze. This technique is usually slower than string-based, but is able to better detect type-2 clones. Tree-base techniques uses abstract syntax trees. They compare sub-trees to find clones. This technique can be slower than token-based, but usually provides better accuracy. Graph-based techniques uses a program dependency graph. This technique is the slowest, but it usually provides the best results with type 1, 2, and 3 clones. Metric-based techniques looks at patterns in code fragments. This technique is usually very quick, but accuracy can be relatively low. String, token, and metric-based are generally scalable. Tree-based techniques are dependent on the algorithm and optimizations, but are less scalable than the former. Graph-

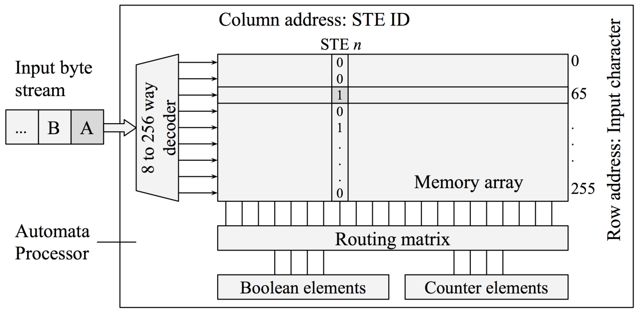


Figure 1: The architecture of one AP block. Each row corrosponds to one $8-bit$ input character, each column corresponds to one STE. If a memory cell is set to 1 then the input character for its row matches with the STE for its column.

based techniques are not scalable as comparison operations are too expensive.

3. BACKGROUND

3.1 The Micron's Automata Processor

Micron's Automata Processor (AP) is a non-von-Neumann hardware implementation of non-deterministic finite automata (NFAs) designed for massively parallel complex pattern searching.

Architecture: The AP chip has three types of functional elements - the state transition element (STE), the counters, and the Boolean elements [9]–see Fig. 1. An STE represents an NFA state, activated or deactivated, via a one-bit register; and its transition rules via a DRAM column—each possible input symbol maps to one row of the DRAM, and each column holds a 1 or a 0 in that position to indicate whether or not that state (column) will match on that input. The AP uses a homogeneous NFA representation [9] for a more natural match to the hardware operation. The STE is the central feature of the AP chip and is the element with the highest population density. Each STE can be configured to match any subset set of 8-bit symbols.

In terms of Flynn's taxonomy, the AP is therefore a very unusual multiple-instruction, single-data (MISD) architecture: each state (column) holds unique responses (instructions) to potential inputs, and they all respond in parallel to each input. Most other commercial architectures are von Neumann architectures, e.g. single CPU cores (SISD), multicore or multiprocessors (MIMD), and GPUs (SIMD).

Counters and Boolean elements are designed to improve the spatial efficiency of automaton implementations and to extend computational capabilities of AP chip beyond NFAs [10]. The counter element counts the occurrence of a pattern described by the NFA connected to it and activates other elements or reports when a given threshold is reached. One counter can count up to $2^{12}-1$, which may not be enough for pattern counting in some cases. In such a scenario, two counters can be daisy-chained to handle a threshold up to $2^{24}-1$. Counter elements are a scarce resource of the first-generation AP chip, and therefore become an important limiting factor for the capacity of the SPM automaton proposed in this work. If future generations of the AP support a higher ratio of counters, the AP's advantage on SPM will improve.

Micron's current generation AP-D480 boards use AP chips built on 50nm DRAM technology, running at an input symbol (8-bit) rate of 133 MHz. A D480 chip has 192 blocks. Each block has 256 STEs, 4 counters and 12 Boolean elements [9]. We assume a AP board with 32 AP chips, so that all AP chips process input data stream in parallel. Each AP D480 chip has a worst case power consumption of 4W [9].

Input and output: The AP takes input streams of 8-bit symbols. The double-buffer strategy for both input and output of the AP chip enables a implicit data transfer/processing overlap. Any STE can be configured to accept the first symbol in the stream (called start-of-data mode, small "1" in the left-upper corner of STE in the following automaton illustrations), to accept every symbol in the input stream (called all-input mode, small "∞" in the left-upper corner of STE in the following automaton illustrations) or to accept a symbol only upon activation. The all-input mode will consume one extra STE in the compiled version.

Any type of element on the AP chip can be configured as a reporting element; one reporting element generates a one-bit signal when the element matches the input symbol. One AP chip has up to 6144 reporting elements. If any reporting element reports on a particular cycle, the chip will generate an output vector which contains 1's in positions corresponding to the elements that report at that cycle and 0's for reporting elements that do not report. If too many output vectors are generated, the output buffer can fill up and stall the chip. Thus, minimizing output vectors, and hence the frequency at which reporting events can happen, is an important consideration for performance optimization. We address this by designing structures that wait until a special end-of-input symbol is seen to generate all of its reports in the same clock cycle.

Programming and configuration: Automata Network Markup Language (ANML), an XML-like language for describing automata networks, is the most basic way to program AP chip, somewhat analogous to assembly language. ANML describes the properties of each element, e.g. STE, counter and Boolean, and how they connect to each other. In addition to ANML, Micron's AP SDK provides a graphical user interface, called the AP Workbench, for quick designing and debugging of small automata and macros. The Micron's AP SDK also provides C, Java and Python binding interfaces to describe automata networks, create input streams, parse output and manage computational tasks on the AP board. A "macro" is a container of automata for encapsulating a given functionality, similar to a function or subroutine in common programming languages. Macros can be templatized (with the structure fixed but the matching rules for STEs and the counter thresholds to be filled in later).

Placing automata onto the AP fabric involves three stages: placement and routing compilation (PRC), routing configuration and STE symbol set configuration. In the PRC stage, the AP compiler deduces the best element layout generates a binary version of the automata network. Depending on the complexity and the scale of the automata design, PRC takes several seconds to tens minutes. Macros or templates can be precompiled and composed later. This shortens PRC time, because only a small macro needs to be processed for PRC, and then the board can be tiled with as many of these macros as fit.

Routing configuration/reconfiguration programs the connections, and needs about 5 milliseconds for a whole AP board. The symbol set configuration/reconfiguration writes the matching rules and initial active states for the STEs. A pre-compiled automata only needs the last two steps. If only STE rules/states change, only the last step is required. This feature of fast partial reconfiguration play a key role in a successful AP implementation of SPM: the fast symbol replacement helps to deal with the case that the total set of candidate patterns exceeds the AP board capacity as well as algorithms like GSP with multiple passes; the quick routing reconfiguration enables a fast switch from k to $k+1$ level sequence mining.

3.2 Levenshtein Distance

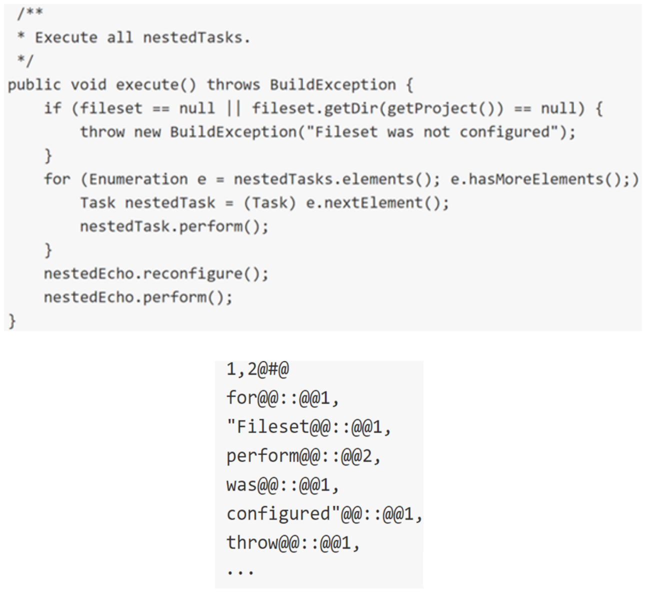
Not all clones are exactly identical. While type-1 clone have identical set of tokens, type-2 and type-3 clones can have a number of differences. To detect type-2 and type-3 clones, the AP needs to consider the Levenshtein Distance between a pair of code blocks. The Levenshtein Distance measures the difference between a pair of entities. It is equal to the number of single edits to transform one entity to the other. For this application, the entity is a code block. It consists of a number of tokens. A single edit is either a 1-token insertion, 1-token deletion, or 1-token substitution. If a code block can be transformed into another code block in less than a set number of edits, we can define the pair as clones.

4. SOURCERERCC

SourcererCC [11] is the code clone detector we are using for comparison. It is a scalable token-based approach that is able to detect type-1, type-2, and type-3 clones. The program uses a partial indexing scheme and optimizations based on properties of code blocks to provide scalable performance.

4.1 Tokenization

Before running the clone detector, the source code first needs to be tokenized. The Turing eXtender Language is used to produce a file containing the set of tokens. The tokenizer accepts the files and a set of parameters with granularity, token size, and code fragment size. Granularity can be set to statements, basic blocks, functions, or files. This parameter determines how tokens should be grouped. Finer size code blocks allows better detection of meaningful clones, but are also more expensive to run. Token size determines the minimum and maximum number of tokens each code block should have. If code blocks have too few tokens, the number of false positives can increase and worsen the quality of the clones detected. If code blocks have too many tokens, the clone detector may take a long time to complete. Code fragment size determines the size of source code each code block can represent. Similar to token size, it can affect the false positive, the quality of the clones, and performance. The tokenizer goes through the source files and identifies each code block. It assigns two numbers for every code block, a block id and a parent id. A block id is an integer that uniquely identifies the code block. A parent id is an integer that groups code blocks together. If two different code blocks have the same parent id, then the clone detector can ignore the comparison. The tokenizer separates the source code into a set of tokens, separated by whitespace.



```
/**
 * Execute all nestedTasks.
 */
public void execute() throws BuildException {
    if (fileset == null || fileset.getDir(getProject()) == null) {
        throw new BuildException("Fileset was not configured");
    }
    for (Enumeration e = nestedTasks.elements(); e.hasMoreElements();)
        Task nestedTask = (Task) e.nextElement();
        nestedTask.perform();
    }
    nestedEcho.reconfigure();
    nestedEcho.perform();
}
```

```
1,2@#@
for@@::@@1,
"Fileset@@::@@1,
perform@@::@@2,
was@@::@@1,
configured"@@::@@1,
throw@@::@@1,
...
```

Figure 2: Tokens generated by the function assuming function granularity.

The token value and the number of occurrences in the code block is recorded. A bookkeeping file is also generated that maps each code block in the token file to the appropriate lines in the source code.

4.2 Clone Detection

The clone detector takes the generated tokens file and threshold value. The threshold value defines how similar a pair of code blocks should be to be considered a clone pair. The tokens in each code block are sorted by number of occurrences starting with the least frequently used. SourcererCC then indexes the tokens. To improve performance, the program only indexes tokens that appear in a sub-block dependent on the number of tokens and threshold. This optimization works since there should be matching tokens near the beginning of each pair of code blocks for a clone to be detected. SourcererCC then uses the index to compare pairs of code block. The comparisons can end early if too many differences are detected early. The program returns a file containing pairs of clones represented by block ids.

4.3 Why SourcererCC?

There are other clone detectors available, but we find SourcererCC to be the most suitable tool to improve and analyze. The clone detection algorithm can be mapped to the AP with little modifications. This allows us to more accurately determine the performance boost of the AP. SourcererCC is also competitive with other available tools. From figure 4.3, SourcererCC does relatively well in detecting clones compared to other similar tools. SourcererCC also provides scales extremely well. From figure 4.3, SourcererCC is competitive with other tools when managing relatively small codebases. It is also the only tool that was able to scale to 250 MLOC. The other tools had issues with memory and time.

5. PROPOSED METHOD

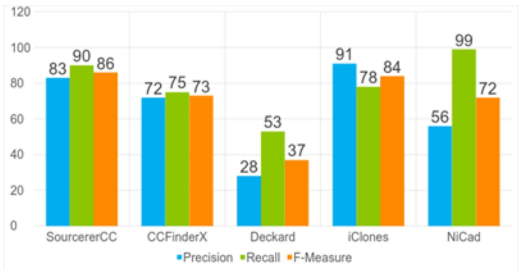
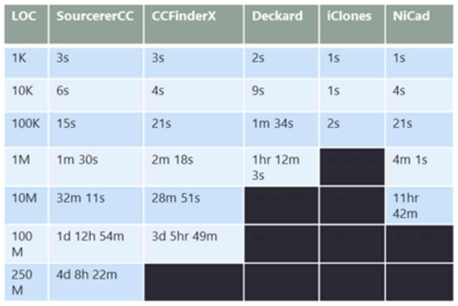


Figure 3: Accuracy of 5 different clone detection tools



LOC	SourcererCC	CCFinderX	Deckard	iClones	NiCad
1K	3s	3s	2s	1s	1s
10K	6s	4s	9s	1s	4s
100K	15s	21s	1m 34s	2s	21s
1M	1m 30s	2m 18s	1hr 12m 3s	■	4m 1s
10M	32m 11s	28m 51s	■	■	11hr 42m
100 M	1d 12h 54m	3d 5hr 49m	■	■	■
250 M	4d 8h 22m	■	■	■	■

Figure 4: Scalability of 5 different clone detection tools

In this section, we represent how the similarity between two code units can be calculated using the Levenshtein distance. Figure 5 represent the tokenized output of two code units, code unit A and code unit B. Each token is encoded to an either a 8-bit symbol (if the number of unique tokens is less than 256) or 16-bit (if the number of tokens is larger than 255 and less than 65536). For example in Code Unit A, *if* is encoded to *Y*, *PingTest* is encoded to *A*, *canRun* is encoded to *U*, and *return* is encoded to *H*. In code unit B, *MacOS* is encoded to *o*, and it should be repeated twice, because frequency of *MacOS* is two in this code unit. As a result, the code unit A is a sequence of letters *YAHU* and the code unit B is a sequence of letters *WAHOO*.

Now, the main task is to calculate the distance between these two strings. If the threshold is 40% (note that the threshold is calculated based on the bigger code unit), in order to consider these two code units as the clones, the number of common tokens should be more than 2 ($40\% \times 5 = 2$). Furthermore, Levenshtein distance will be 3 (which is the difference between the tokens in the larger code and number of common tokens).

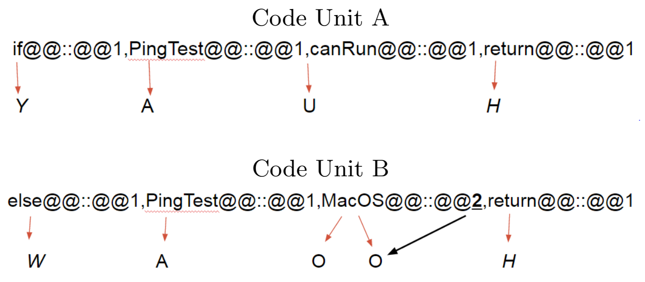

Code Unit A
if@@::@@1,PingTest@@::@@1,canRun@@::@@1,return@@::@@1
Y
A
U
H
Code Unit B
else@@::@@1,PingTest@@::@@1,MacOS@@::@@2,return@@::@@1
W
A
O
O
H

Figure 5: An example of the tokenized output of two code units

Threshold = T, #token in larger code unit = N
C (#tokens should be in common) = T $\times$ N
LD (Levenshtein Distance) = N - C

Figure 6: Some equations

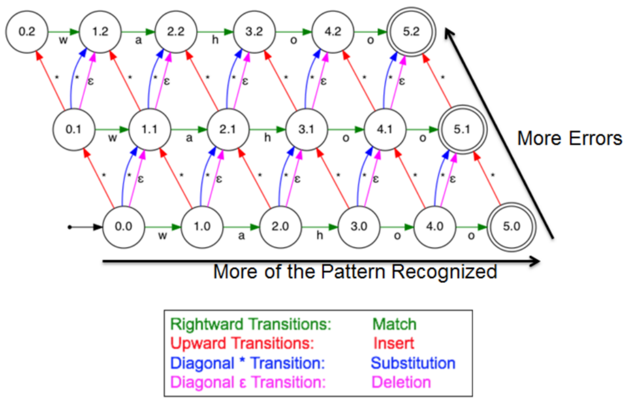

0.2
1.2
2.2
3.2
4.2
5.2
0.1
1.1
2.1
3.1
4.1
5.1
0.0
1.0
2.0
3.0
4.0
5.0
w
a
h
o
o
*
ε
More Errors
More of the Pattern Recognized
Rightward Transitions: Match
Upward Transitions: Insert
Diagonal * Transition: Substitution
Diagonal ε Transition: Deletion

Figure 7: Levenstein Automata

5.1 Levenstein Automata

The Automata Processor is mainly designed to support regular expression matching. The AP is capable of executing many automaton simultaneously. Our problem can be mapped to Levenstein distance and Levestein distance can be much cheaper to be performed on the AP than on the conventional CPUs. In a von-Neumann processor, the Levenstein structure should first be converted to a deterministic finite automata (DFA) and the power-set construction from non-deterministic finite automata (NFA) to DFA can be exponential. As a result, the AP could potentially handle larger automaton than von Neumann processors.

The actual Automata Structure is proposed [12]. Figure 7 shows the automata structure to match string *wahoo* with edit distance 2. All the 5.0, 5.1, and 5.2 are the reporting states. If state 5.0 reports, it means the string is matched against *wahoo* is itself *wahoo*. If the edit distance of the input string with the automata is one (two), state 5.1 (5.2) will report. In this structure, rightward transitions (green transitions) are activated, when there is a match, and upward transitions (red transitions) are activated, when an inset operation in needed. Similarly, blue and purple transitions get activated, when there is a substitution and deletion, respectively. After recoding the token in the code units, we order the symbols in order to faciliate the matching process.

5.2 Program Infrastructure

Fig. 8 shows the complete workflow of the AP-accelerated clone detection proposed in this project. The input database is the source code databases. In pre-processing stage, the code is tokenized and the tokens are recoded to the 8-bit or 16-bits symbol. Then, the corresponding AP input stream will be generated. Then, the appropriate pre-compiled template macro of automaton structure for Levenstein automata

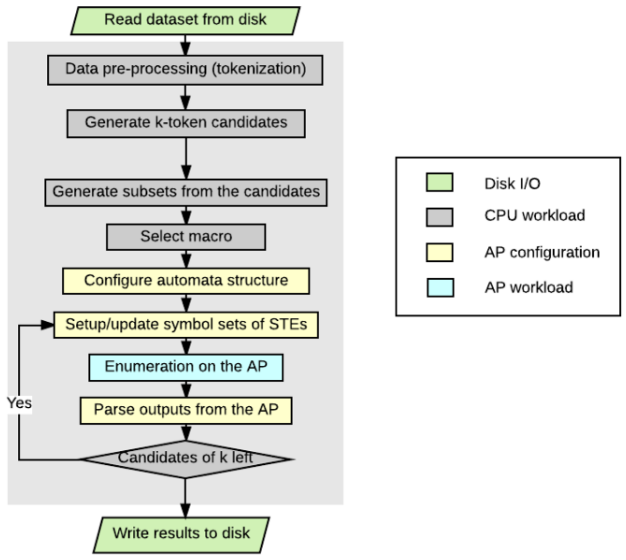

Read dataset from disk
Data pre-processing (tokenization)
Generate k-token candidates
Generate subsets from the candidates
Select macro
Configure automata structure
Setup/update symbol sets of STEs
Enumeration on the AP
Parse outputs from the AP
Candidates of k left
Yes
Write results to disk
Disk I/O
CPU workload
AP configuration
AP workload

Figure 8: Flowchart

is selected according to the number of tokens and the edit distance is configured on the AP board. The symbols are generated on the CPU and are filled into the selected automaton template macro. The input data formulated in pre-processing is then streamed into the AP board for calculating the similarity. If there are candidates left to be processed on the AP, the symbols will be changed with a new set of symbols. Fortunately, the latency of symbol replacement could be significantly reduced in future generations of the AP (because symbol replacement is simply a series of DRAM writes), which improves the overall performance greatly. Finally, the result (which are the code clone pairs) are written to the disk.

6. EXPERIMENTAL RESULTS

6.1 Testing Platform and Dataset

The experiments are tested using the following hardware:

- CPU: Intel CPU i7-5820K (6 cores, 3.30GHz), memory: 32GB, 2.133 GHz
- AP: D480 board, 133 MHz clock, 32 AP chips (simulation)

As our target we use IJaDataset 2.0 [13], a large inter-project Java repository containing 25,000 opensource projects (3 million source files, 250MLOC) mined from SourceForge and Google Code.

6.2 Configuration

In SourcererCC, Granularity of the code units is functions or blocks. The block granularity is only supported for Java. In this project, we are interested in smaller granularity, and our database is in Java, so we choose block as the code unit. The following parameters avaialble in the sourcererCC are defined here:

- minTokens: A code unit should have at least these many tokens to be considered for further processing. Setting the minTokens = 0 means no bottom limit

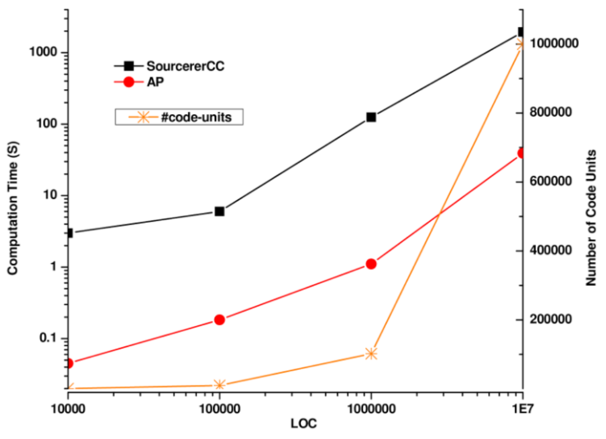


Figure 9: AP performance vs SourcererCC on CPU

- maxTokens: A code snipper should have at most these many tokens to be considered for further processing. Setting the maxTokens = 0 means no upper limit
- minLines: A code unit should have at least these many lines to be considered for further processing. Setting the minLines = 0 means no bottom limit
- maxLines: A code unit should have at most these many lines to be considered for further processing. Setting the maxLines = 0 means no upper limit

We only set the limit for the maximum number of tokens as 20. We also set the threshold to be 60% for our experiments. Therefore, the Levenstein atomata will be of size 20 and maximum edit distance of 8.

6.3 SourcererCC vs the AP solution

Figure 9 represents the performance of the AP solution versus the performance of SourcererCC in the CPU platform. The left vertical axis is in log space and shows the computational time. The left vertical axis represents the number of code units. The horizontal axis shows the number of lines of code. Clearly, the number of code units increase, when increasing *LOC*. The performance of the AP is higher than sourcererCC, which is up to 130X, and the AP advantages increases by increasing *LOC*.

The accuracy of the AP solution is the same as SourcererCC, because the functionality of clone detectors are similar. Smaller code units result in higher accuracy, and interestingly, the AP performs better, because the smaller automata structure decreases the routing complexity. However, the finer granularity results in more code unit, which makes it more complicated for SourcererCC.

7. CONCLUSION AND FUTURE WORK

The AP can improve the performance of scalable code clone detection without sacrificing precision and recall. We have shown that the algorithm of SourcererCC can be adapted and modified in order to be implemened on the AP. We see significant performance boost using the AP for codebases of varying lengths. Regarding future work, we would like to decrease the number of false positives introduced in sourcererCC, by adding sequential information to the algorithm.

This will help to increase the precision of the task. Also, the routing complexity for the AP increases in larger edit distance, which we will try to work around this problem. Furthermore, we would like to perform more experiments on larger datasets.

8. REFERENCES

[1] A. Monden, D. Nakae, T. Kamiya, S.-i. Sato, and K.-i. Matsumoto, “Software quality analysis by code clones in industrial legacy software,” in *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on.* IEEE, 2002, pp. 87–94.

[2] D. Rattan, R. Bhatia, and M. Singh, “Software clone detection: A systematic review,” *Information and Software Technology*, vol. 55, no. 7, pp. 1165 – 1199, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584913000323

[3] J. Svajlenko, I. Keivanloo, and C. K. Roy, “Scaling classical clone detection tools for ultra-large datasets: An exploratory study,” in *Proceedings of the 7th International Workshop on Software Clones.* IEEE Press, 2013, pp. 16–22.

[4] ——, “Big data clone detection using classical detectors: an exploratory study,” *Journal of Software: Evolution and Process*, vol. 27, no. 6, pp. 430–464, 2015.

[5] M. Becchi and P. Crowley, “A hybrid finite automaton for practical deep packet inspection,” in *Proceedings of the 2007 ACM CoNEXT conference.* ACM, 2007, p. 1.

[6] R. K. Saha, C. K. Roy, K. A. Schneider, and D. E. Perry, “Understanding the evolution of type-3 clones: an exploratory study,” in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on.* IEEE, 2013, pp. 139–148.

[7] C. K. Roy, J. R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[8] A. Sheneamer and J. Kalita, “A survey of software clone detection techniques.”

[9] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, “An efficient and scalable semiconductor architecture for parallel automata processing,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 12, pp. 3088–3098, 2014.

[10] K. Wang and K. Skadron, “Cellular automata on the micron automata processor,” U. Virginia Dept. of CS, Tech. Rep. CS-2015-03, April 2015.

[11] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, “Sourcerercc: scaling code clone detection to big-code,” in *Proceedings of the 38th International Conference on Software Engineering.* ACM, 2016, pp. 1157–1168.

[12] T. Tracy, “Nondeterministic finite automata in hardware-the case of the levenshtein automaton,” in *Proceedings of the workshop on Speech and Natural Language.* White paper.

[13] “mbient Software Evoluton Group. IJaDataset 2.0., howpublished = http://secold.org/projects/seclone, note = January 2013.”