Text Mining in Program Code

Alexander Dreweke[1]
Computer Science Department 2
Friedrich-Alexander University Erlangen-Nuremberg
Martensstr. 3
91058 Erlangen
Germany
voice: +49 9131 852 7923
fax: +49 9131 852 8809
email: dreweke@cs.fau.de

Ingrid Fischer*
Nycomed Chair for Applied Computer Science
University of Konstanz
BOX M712
78457 Konstanz
Germany
voice: +49 7531 88 5016
fax: +49 7531 88 5132
email: Ingrid.Fischer@inf.uni-konstanz.de

Tobias Werth
Computer Science Department 2
Friedrich-Alexander University Erlangen-Nuremberg
Martensstr. 3
91058 Erlangen
Germany
voice: +49 9131 852 8865
fax: +49 9131 852 8809
email: werth@cs.fau.de

Marc Wörlein
Computer Science Department 2
Friedrich-Alexander University Erlangen-Nuremberg
Martensstr. 3
91058 Erlangen
Germany
voice: +49 9131 852 7623
fax: +49 9131 852 8809
email: woerlein@cs.fau.de

[1] Authors are in strict alphabetical order.

Text Mining in Program Code

ABSTRACT

Searching for frequent pieces in a database with some sort of text is a well-known problem. A special sort of text is program code as e.g. C++ or machine code for embedded systems. Filtering out duplicates in large software projects leads to more understandable programs and helps avoiding mistakes when reengineering the program. On embedded systems the size of the machine code is an important issue. To ensure small programs, duplicates must be avoided. Fast program execution can be ensured, when frequently used duplicates are encoded in hardware. The most successful approaches for finding code duplicates are based on graphs representing the data and control flow of the program and graph mining algorithms. Compared to applications of suffix tries on the code or fingerprinting, where some kind of special form of program parts is calculated, more duplicates are found.

INTRODUCTION

Computer programs are a special form of text. Words of a programming languages are combined to form correct sentences in this programming language. There exists a wide variety of programming languages, ranging from high-level object-oriented languages like Java or C++ to machine code, the language a processor can actually "understand". Programming languages are usually translated with the help of compilers from high- to low-level. To produce this kind of “text” - the computer programs - is the daily work of many programmers; billions of lines of code have been written. Mostly, this code is not well documented and not really understood by anybody after the original programmer stopped working. Typically, many programmers are working on one project and often old code from former versions or other projects is used.

A special problem in big amounts of program code are duplicated code fragments. These duplicated fragments can occur because of excessive use of "copy & paste", because something was simply re-programmed or also because of the compiler. When translating from the high-level to intermediate or low-level languages, new duplicates can be introduced, e.g. by using code templates for instructions and instruction sequences.

Finding these duplicates has been in the focus of interest for many years. Code duplicates are called clones and clone detection has produced many different algorithms. If program code is simply viewed as text, clone detection is nothing else than mining in this text with the goal of finding the duplicate or similar code. Merging application areas and algorithms from the data mining community on the one hand and clone detection leads to fruitful new insights and results.

Finding duplicated code in programs can have different goals. First these duplicates can be visualized as a hint for programmers that something has to be done about this specific piece of code. Second, the redundant code can be replaced automatically by subroutine calls, in-lined procedure calls, macros etc. that produce the same result. This leads to smaller code that is easier to understand or to maintain. Third, methods to detect and replace duplicated code can be integrated into compilers. Finally, finding duplicated code can lead to special hardware for the duplicates in the area of embedded systems.

In the case of program code, duplicates are not always "totally equivalent". It is not only the one-to-one duplicate from a piece of code that is interesting. Also near duplicates or even pieces of code, that are syntactically different, but semantically equivalent must be found. E.g. in two fragments only two independent pieces of code having no side effect onto each other can be exchanged. Variable names can be different or registers in machine code can vary.

The application of clone detection ranges from high-level languages to machine code for embedded systems. The latter is the main topic in this chapter. The clone detection algorithms especially for embedded systems are described in detail.

Clone Detection and Embedded Systems

In recent years, many small programmable devices appeared on the market. PDAs, mobile phones, and navigation systems are examples for these systems. Typical software on these gadgets deals with speech or image processing, encryption, calendars or address books. Therefore, the devices contain small processors, handling the different tasks.

But also in larger technical devices, such as cars, the number of small processors exploded in the last years. They offer e.g. efficient engine control, drivers assistance functions such as the “Electronic Stability Program” or “Line Departure Warning”, that warns the driver when he/she is leaving the right lane or more comfort functions such storing the different seat features for different persons. In modern high-end cars a network of up to sixty communicating processors are used.

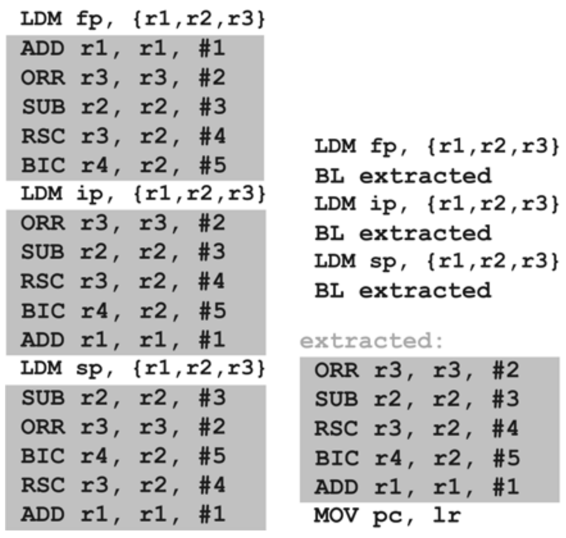
These processors, also called embedded systems, are much more specialized in nature than general computing systems (e.g. desktop computers or laptops). Typically, they are produced at low cost, have low energy consumption, are small, and meet rigid time constraints for their tasks. Normally, they are fabricated in high quantities. These requirements place constraints on the underlying architecture. An embedded system consists of a processor core, a Read Only Memory (ROM), Random Access Memory (RAM), and an Application Specific Integrated Circuit (ASIC). The cost of developing an integrated circuit is linked to the size of the system. The largest portion of the integrated circuit is often devoted to the RAM for storing the application. Nevertheless, the ROM is much smaller than storage offered on desktop computers. Developing techniques reducing code size are extremely important in terms of reducing the cost of producing such systems. Programmers writing for embedded systems must be aware of the memory restrictions and write small programs in the sense of code size. Less lines of code can make a huge difference on embedded systems. As the complexity of embedded systems programs grows, programming machine code directly gets more complicated. Today it is common to use high-level languages as C or C++ instead. However programming in these languages results in larger machine code compared to machine code written directly by the programmers. One reason is that compiler optimization classically focuses on execution speed and not so much on code size. Introducing code size optimizations on embedded systems often leads to speed trade-offs, but when execution speed is not critical, minimizing the code size is usually profitable.

Different automated techniques have been proposed over the years to achieve this goal of small code (Beszdes, Ferenc, Gyimthy, Dolenc, Karsisto, 2003; Debray, Evans, Muth & Sutter, 2000). These techniques can be separated into two groups. In code compression the code is somehow compressed and must be decompressed again at runtime. This technique results in very small code, but an additional step is necessary to decompress necessary parts. Decompression itself does not take too much time (depending on the compression algorithm) but there must be enough memory to store the decompressed code parts again. Another method is code compaction: the code is shrunk, but remains executable.

It is also typical for embedded systems, that they are inflexible. Once the mobile phone is fabricated, the functionality can not be changed. If a new communication standard is proposed, the old mobile phone is thrown away and a new phone is build. For that reason, more flexible embedded systems are researched. The more applications they can handle, the more they will be used. The problem is, that the flexibility of these general purpose processors is paid with performance trade-offs. To handle these performance trade-offs, sets of instructions that are frequently executed are implemented as fixed logic blocks. Such blocks perform better than the softer, reconfigurable parts of the device. If they are well chosen, the performance of the system is enhanced. The goal is to have the frequent parts in hard logic implementation, while simultaneously affording flexibility to other components.

Recapitulating, three problems in programming embedded systems are based on code clones: code compression, code compactification, and finding candidates for hard logic implementations. A possible example of code clones in machine code is given in the next section.

An Example



```
LDM fp, {r1,r2,r3}
ADD r1, r1, #1
ORR r3, r3, #2
SUB r2, r2, #3
RSC r3, r2, #4
BIC r4, r2, #5
LDM ip, {r1,r2,r3}
ORR r3, r3, #2
SUB r2, r2, #3
RSC r3, r2, #4
BIC r4, r2, #5
ADD r1, r1, #1
LDM sp, {r1,r2,r3}
SUB r2, r2, #3
ORR r3, r3, #2
BIC r4, r2, #5
RSC r3, r2, #4
ADD r1, r1, #1
```

```
LDM fp, {r1,r2,r3}
BL extracted
LDM ip, {r1,r2,r3}
BL extracted
LDM sp, {r1,r2,r3}
BL extracted

extracted:
ORR r3, r3, #2
SUB r2, r2, #3
RSC r3, r2, #4
BIC r4, r2, #5
ADD r1, r1, #1
MOV pc, lr
```

Figure 1 Example instruction sequence and corresponding minimum code sequence

Figure 1 on the left hand side shows a sequence of 18 ARM assembler instructions. The ARM architecture is a popular embedded architecture with a common 32-bit RISC instruction set. Three fragments of the code are highlighted. These three parts contain the same operations but in different orders. It is easy to see that most of these operations are independent of each other, so reordering the operations does not change the semantics of the code piece. E.g. the result of **`ADD r1, r1, #1`** is not of interest for any other operation of the gray shaded area as register **`r1`** containing the result of this operation is not used anywhere else. The instruction sequence on the left of Figure 1 can be maximal compacted to 12 instructions (see Figure 1 right hand side) by reordering the instructions according to their data dependencies. After doing this one can remove the instructions by a call (operation **`BL`**) to a single code instance labeled "extracted". At the end of this new code the program counter (**`pc`**) must be restored to the link register (**`lr`**) – containing the return address of a function - in order for the program to execute the next instruction after the call to the extracted code. We use this example to compare the various text mining approaches to explains their advantages and limitations.

In the remainder of this paper an overview on mining in program code for embedded systems is given. As a basis, we take several research areas from embedded system design and compiler construction all having the same goal to find repeated code fragments. In the next section, the different application areas, in which shrinking program code is of interest, are described. In the third section the main focus is set on the different methods to find frequent fragments in program code.

BACKGROUND

Procedural abstraction and cross jumping

In the seventies and eighties of the last century code size moved in the focus of embedded system research. Until then only speed was of interest, the size of the ROM changed this view. Procedural abstraction and cross jumping are two techniques, that can be applied at different stages of compilation ranging from source code at the beginning over the intermediate code of the compiler to machine code at the end. Both methods start with finding repeated code fragments. To find these fragments, several possibilities exist which are explained in the next section *"Main Focus"* of this chapter. The repeated code fragments are candidates to be extracted into a separate procedure as shown in Figure 1. This is the basic idea of procedural abstraction (Fraser, Myers, & Wendt, 1984; Dreweke, Wörlein, Fischer, Schell, Meinl & Philippsen, 2007). But before extraction the fragments found must be examined whether they are really suitable. The main problem is that frequent fragments might overlap in the code. In this case only one fragment can be extracted while the second fragment is ignored. It must be calculated which groups of frequent fragments has the most savings at the end. Procedural abstraction then turns repeated code fragments into separate procedures. The fragments are deleted in their original positions and procedure calls are inserted instead. Our running example in Figure 1 shows procedural abstraction after the compilation applied to machine code.

Cross jumping is also known as tail merging. It is applicable if the identified fragments have a branch instruction to the same destination in common on the last instruction. One fragment is left as it is and the other occurrences are replaced by branch instructions to the start of this fragment. This branch instruction at the end of the frequent fragment ensures that program execution continuous at the correct position even after the extraction of the frequent fragment.

Both strategies have their benefits and costs in code space and execution time. While the size of the original program shrinks, execution might take longer for procedural abstraction as procedure calls take their time. Therefore, it is not advisable to apply procedural abstraction to code that is often used during runtime (so called hot code). Procedural abstraction and cross jumping require no hardware support at all.

There are several approaches and enhancements that do not check for completely equivalent fragments. E.g. fragments for machine code can differ in the registers used. It might be possible to extract two code sequences that only differ in their registers into one procedure and map the registers onto each other. If the register mapping does not introduce more code than the extraction into a procedure saves, this approach is worth it (Cooper & McIntosh,1999).

Procedural abtraction is applied on different stages of compilation. The problem of having different registers in fragments can be avoided when procedural abstraction is applied on the intermediate representation of the compiler before the register mapping takes place. There are also several approaches where procedural abstraction is applied before or post link-time. If applied after library programs etc. are linked to the main code, more duplicates can be found. Another difference is what portions of the code are taken into account. Sometimes the whole program is taken into consideration. This is computationally more complicated. In other cases the program is split into smaller parts. E.g. for machine code, basic blocks are often considered quite often. A basic block is code that has one entry point one exit point and no jump instructions contained within it. Procedural abstraction can also be applied to source code, but due to the rich syntax of high level programming languages, it is more difficult to find real code duplicates. For source code, semantically equivalent but syntactically different code pieces are more interesting. Tools successfully applying procedural abstraction are aipop (Angew. Informatik GmbH, 2006) and Diablo (University of Gent, 2007).

Dictionary Compression

What procedural abstraction is in code compaction, dictionary compression (Brisk, Macbeth, Nahapetian & Sarrafzadeh, 2005; Liao, Devadas & Keutzer, 1999) is in code compression. Redundant sequences of instructions are identified in the program as for procedural abstraction. But these sequences are then extracted into a dictionary and replaced with a codeword acting as index into the directory. At runtime the program is decompressed again based on the keywords and the directory. When a key is reached, control is transferred to the dictionary and the corresponding sequence is executed. Afterwards control is transferred back to the original program.

Compiler construction for dictionary compression has focused on identifying redundant code sequences in a post-compilation pass, often at link time. Famous variants for dictionaries are the *Call Dictionary* (CALD) instruction, an early hardware-supported implementation of dictionary compression. The dictionary consists simply of a list of instructions. Sequences of instructions are extracted from the original program and replaced with a CALD instruction. The CALD instruction has the form **`CALD(Addr, N)`** where **`CALD`** is an operational code and **`Addr`** is the address of the beginning of the code sequence. **`Addr`** is given as the offset from the beginning of the directory. To execute a sequence of commands in the directory, control is transferred to the dictionary address **`Addr`** when a CALD instruction is reached. The next *N* instructions in the directory are executed. Finally control is returned to first instruction after the CALD instruction.

Echo instructions are similar to CALD instructions. Instead of storing the instruction sequences externally, one instance is left in inline the program. All other instances of the sequence refer to this special one. Using echo instructions the directory resides inside the program in opposition to CALD instructions where the dictionary is external to the program.

As for basic procedural abstraction the code sequences must be completely identical for the basic versions of CALD and echo instructions. But variations to allow fragments that contain syntactical differences are developed.

Regularity Extraction and Template Generation

As mentioned in the introduction, a lot of embedded systems are extremely inflexible as they can not be reused after program changes, especially ASICs can not be reused. Instead reconfigurable devices are developped, that can be easily re-programmed (Amit, Sudhakar, Phani,Naresh & Rajesh, 1998; Brisk, Kaplan, Kastner & Sarrafzadeh, 2002; Zaretsky, Mittal, Dick & Banerjee, 2006; Liao, Devadas & Keutzer, 1999) They have been proposed as the new design platform for specialized embedded processors. Therefore, new reconfigurable devices build upon programmable logic technology appeared. To ensure reasonable runtime, functions that have to be executed frequently are implemented as hard logic blocks. If these functions are well chosen, flexibility and runtime can be combined. In the area of embedded systems and hard logic implementations finding these frequent functions is called regularity extraction. These regularities are nothing else than frequent code fragments. It is common in this are to search for frequent fragments on a graph representing the control and data flow of the corresponding program in the intermediate representation of the compiler. Possible frequent fragments are called templates. The process of finding these templates is named template generation. Despite this name template generation is nothing else than mining for frequent fragments. Working on the templates, i.e. putting them into hard logic, is called template extraction. In the beginning, regularity extraction started with a given hand-tuned database of templates that were just combined to look for templates in the given code. There are also many algorithms looking for templates of a special form. As program code is represented as graph, templates are subgraphs. Restrictions e.g. limit to subgraphs with only one entry and/or one exit node. Other algorithms work on directed acyclic graphs or restrict the size of templates. Due to the high complexity of graph mining, many algorithms work with the help of heuristics. Applications of regularity extraction can also be found in reduction of data-path complexity and power reduction.

MAIN FOCUS OF THE CHAPTER

In this section an introduction into the standard techniques to decrease code size is given.

Searching on strings with Suffix trees

Fraser, Myers & Wendt (1984) were the first trying to find repeated fragments in machine code. They searched the machine code like a string of letters using suffix trees to identify repeated sequences within the sequence of assembly instructions. The fragments were then abstracted out into procedures. Applied to a range of Unix utilities on a Vax processor, this technique managed to reduce code size by about 7% on the average.

A suffix tree is a trie that is a common search structure to detect strings in text. The term trie comes from "retrieval". Due to this etymology it is pronounced "tree", although some encourage the use of "try" in order to distinguish it from the more general tree. A *trie* is a rooted tree whose edges are labelled with one character of the underlying character set. It stores all words of the text. Therefore, all words are inserted based on their character sequences. For each word exist a path from the root to one leaf so that the corresponding labels of the edge sequence express the stored word. Each node has for each existing character just one outgoing edge so that common word prefixes share the same part of the tree.

When searching an existing word, the algorithms has to walk through the trie beginning at the root and follow the edges according to the character sequence of the word. The word is stored, is there is a path from the root to a leaf and the edges are labelled with the characters of the word in the correct order. Otherwise the traversal stops at some leaf and the word remains uncomplete. At this node the missing suffix sequence of the word has to be added if the word should stored in the trie.

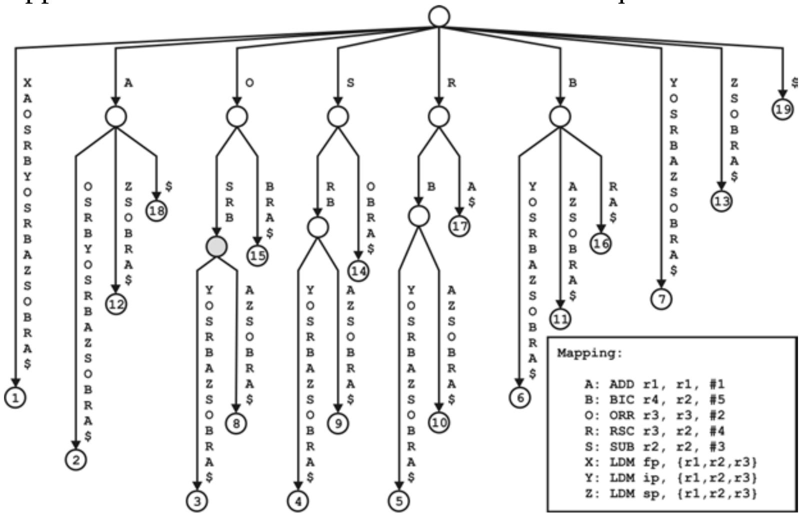
A *patricia trie* is a more compact description of a trie. Sequential parts of the trie where each node has just one incoming and at most one outgoing edge are collapsed into single edges labelled with the whole character sequence of that part.

A *suffix trie* of a string is a trie representing all the suffixes of the string. When a suffix trie for string *A* has been constructed, it can be used to determine whether another string *B* is a substring of string *A*. If there exist a path from the root to a node so that the edge sequence express the searched subsequence *B* then the *B* is a substring. With a special "string end" character (*$*) each suffix terminates by this character that never occurs inside a sequence. Therefore, there is one leaf for each suffix. In this node the starting position of the suffix is stored.

A *suffix tree* of a string is a patricia trie containing all the suffices of the string. These suffix tree can be constructed in linear time and requires linear space

(Ukkonnen, 1995). It is an ideal data type for detecting substrings. A suffix trie and a suffix tree express the same (with different space requirements) so further on just suffix trees are used.

To store program code in suffix trees the instruction sequence(s) must be mapped to a corresponding alphabet. Textual or semantically identical instructions have to be mapped to the same character to be detected as equal.



```
Mapping:

    A: ADD r1, r1, #1
    B: BIC r4, r2, #5
    O: ORR r3, r3, #2
    R: RSC r3, r2, #4
    S: SUB r2, r2, #3
    X: LDM fp, {r1,r2,r3}
    Y: LDM ip, {r1,r2,r3}
    Z: LDM sp, {r1,r2,r3}
```

Figure 2 Suffix tree of the running example

For the example program the first letters of the operational code (if it is non-ambiguous) is chosen. So the whole example is represented by the string "XAOSRBYOSRBAZSOBRA$". In Figure 2 the suffix tree for this string is presented.

Each path from the root to a node in the suffix tree represents a subsequence of the given program. To decide which patterns are interesting for extraction, the size (number of instructions) and the number of occurrences are relevant. The number of instructions is equal to the number of characters of the subsequence and can be counted during the insertions process.

The number of occurrences can also be read out of the suffix tree. Each path from one inner node to a leaf contains a possibility to extend the subsequence to a suffix of the whole code. Because of the "string end" character each leaf represent one unique suffix, so the number of reachable leaves represent the number of occurrences of the substring. Each suffix stored in the reachable leaves starts with the substring.

Not only the number of occurrences, even their location is stored in the suffix tree. Each leaf contains the start of the suffix and therefore points to one occurrence of this substring. No search in the code is necessary after the creation of the suffix tree. Subsequences may overlap in the code sequence. This can be detected by comparing the distance between the start points and the length of the substring.

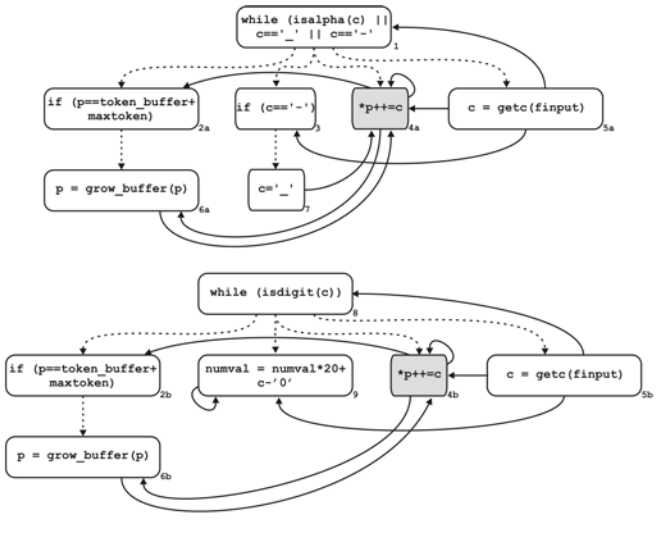
Out of this information the "interesting" candidates can be selected. In the example the only fragment that leads to smaller code is the sequence OSRB with its two occurrences marked in Figure 2. Extracting this sequence in a function results a code sequence with one instruction saved. All other subsequences are to short or too seldom. Compared to Figure 1 this leads to a smaller gain in code size. As suffix trees can only handle syntactically equivalent code pieces, the order variations of if the extracted fragment in Figure 1 can not be detected. Other methods are needed to get the maximal size gain for this example.

Slicing

Slicing-based detection of duplicated code works on graphs representing the control and data flow of a program, the so called control data flow graphs (CDFGs). Program slicing is commonly used when specializing an existing program at some point of interest of the source code, the slicing criterion. A program slice consists of all instructions and values that might affect this point of interest, also known as slicing criterion. These parts of the program can be extracted into a new program without changing the semantics for the slicing criterion (Komondoor & Horvitz, 2003).

Therefore, the CDFG is built up by inserting nodes for each instruction and edges of one type for each data dependency and edges of another type for each control flow dependency. Program slicing is done from the slicing criterion backwards on the edges, until no new node is marked as visited.

Kommondoor & Howritz (2003) search for isomorphic subgraphs in several CDFGs in order to extract the associated code by a new procedure and the occurrences by calls to this newly created procedure. The nodes of a CDFG also represent the statements of the program, whereas the edges reflect a control flow or data dependency between the instructions. The nodes and therefore the instructions are partitioned in equivalence classes by means of the instructions. Within an equivalence class, pairs of matching nodes are the start point for growing isomorphic subgraphs. Slicing backwards (respectively forwards) is done if and only if the matching predecessor (respectively successor) of the current investigated node also is a matching node.

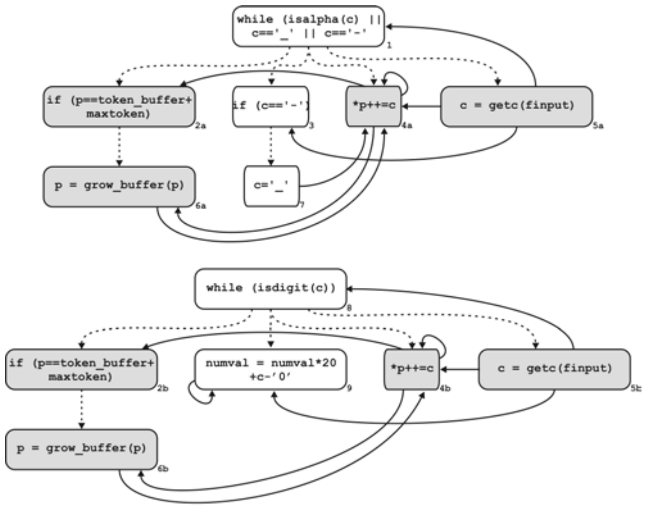


```
while (isalpha(c) ||
        c == '_' || c == '-') {
    if (p == token_buffer + maxtoken)
        p = grow_buffer(p);
    if (c == '-') c = '_';
    *p++ = c;
    c = getc(finput);
}
```

```
while (isdigit(c)) {
    if (p == token_buffer + maxtoken)
        p = grow_buffer(p);
    numval = numval*20 + c - '0';
    *p++ = c;
    c = getc(finput);
}
```

Figure 3 Control data flow graphs of *bison*, its source code, and extracted function

In Figure 3 an example from the source code of the GNU utility bison is illustrated. As a starting point the two matching nodes 4a and 4b from the different code sequences are selected because they are in the same equivalence class. From now on, further nodes are added to this initial code clone using backward (or forward) slicing. Slicing backwards, on the control dependency edges of the nodes 4, leads to the two while-nodes 1 and 8, which are not in the same equivalence class and, therefore, do not match. The self-edge to node 4 does not lead to new matches, of course. But slicing backward to node 5a/5b and 6a/6b adds to matching nodes to the isomorphic graphs. The node 7 also is reached through a data dependency edge but is not considered because there is no corresponding match in the other program dependence graph. Slicing backwards now from nodes 5a/b leads to the while-nodes 1/8 again. But from the edges of the nodes 6a/6b the matching nodes 2a/2b can be reached by backward slicing, so they also are added to the isomorphic graphs. No further nodes can be added using slicing in this example. The resulting graphs are shown in Figure 3.



```
void next_char(char **ptr_p, char *ptr_c){
    if (*ptr_p == token_buffer + maxtoken)
        *ptr_p = grow_buffer(*ptr_p);
    *(*ptr_p)++ = *ptr_c;
    *ptr_c = getc(finput);
}
```

Figure 4 Isomorphic subgraphs of program dependence graph in Figure 3

When using program slicing for specialization, typically solely backward slicing is used. However, using slicing for text mining also requires forward slicing for "good" code clones. This can be explained by conditional statements (like if), which have multiple successors that cannot be reached by backwards slicing from any other conditional branch.

The isomorphic subgraphs are expanded by utilizing the transitive closure. Therefore, if already found the duplicated code pairs (C1, C2) and (C2, C3), (C1, C3) also are duplicates and would be combined to group (C1, C2, C3).

Instead of other approaches that deal with source code, this one is able to find non-sequential or reordered instructions, which can be extracted, although, without changing the semantics of the program. Additionally, it is able to find similar not only identical program sequences, when defining equivalence classes without respecting the literals. The extraction must rename the variables or insert some temporary. The isomorphic subgraphs must not cross loops, since this may change the meaning of the program. Furthermore, the duplicated code is checked for extractability.

Using backwards slicing leads to isomorphic subgraphs, but does not lead to the whole set of possible subgraphs. This way, not only the computationally complexity of subgraph isomorphism is avoided but also the number of code clones is reduced leading to more meaningful code clones in average.

Fingerprinting

In clone detection, comparison of code fragments to find frequent fragments is expensive. If the number of code fragments is very large, many single comparisons have to be done. In order to reduce the amount of expensive code comparisons, often a hash code or fingerprint of the code fragments is used.

If two code blocks are identical, they have the same fingerprints, but not vice versa. If two code blocks have identical fingerprints, these blocks do not necessarily have to be identical. Nevertheless, different fingerprints for two code blocks always mean that these blocks are also different.

The calculation of the fingerprint depends on application demands, especially on the interpretation of "identical". Some applications require that code blocks are identical in the instructions and in the instruction sequence. Other requires just semantically, not structural identities, so a more flexible fingerprint is necessary. A very open fingerprint is to count each occurrence of each type of operational code regardless to their parameters or order. This scheme is resistant to code reordering, register mapping or variable renaming, but results in false positives. Code blocks with identical fingerprint have to be compared to decide, if they are really identical.

Less false positive can be reached by storing the code order in the fingerprint. For example, Debray, Evans, Muth & Sutter (2000) encode only the first sixteen instructions of a basic block in a 64-bit value, for the reason that most basic blocks are short. In this way, each of the sixteen instructions is encoded as a 4-bit value, representing the operational code of the instruction. There is a maximum of sixteen code strings with four bits, even since most systems have more operational codes. In that case all operational code are sorted by frequency, determine an encoding for the most frequent fifteen operational codes, and use the last encoding for all other operational codes. Additionally, the fingerprints are hashed in order to reduce the number of fingerprint comparisons. Fingerprints in different hash buckets are different and must not be compared. In Table 1 the hash code of the fingerprints as described s shown.

Table 1 Hashing the first 16 instructions of the code shown in Figure 1

1001	1000	0011	0100	1110	1100	1001	0011	0100	1110	1100	1000	1001	0100	0011	1100
LDM	ADD	ORR	SUB	RSC	BIC	LDM	ORR	SUB	RSC	BIC	ADD	LDM	SUB	ORR	BIC

Other approaches calculate the fingerprint by using the Rabin-Karp string matching algorithm. Therefore, each instruction is encoded to an integer, reflecting its operational code and operands. The fingerprint of a code block is computed out of these integers. Therefore each instruction number is multiplied with increasing powers of as special base *b,* which is bigger than the largest encoded integer. The terms for all instructions of the code block are summed up and the remainder of the division by an fixed integer *p* (mostly prime) is the final fingerprint *f* for this code sequence:

$$f_i = (c_i * b^{n\text{-}1} \; + \; c_{i+1} * b^{n\text{-}2} \; + \; ... \; + \; c_{i+n\text{-}1} * b^0) \bmod p$$

Those fingerprints are calculated for a specified length n in the whole code. Like a sliding window of the size n, the fingerprints for all subsequences of the size n can be calculated out of the first fingerprint. Therefore the highest summand representing the unnecessary first instruction has to be subtracted out of the finger print and the newly instruction has to be inserted. All other instructions parts just have to be multiplied by the basis *b*:

$$f_i = ((f_{i-1} * b) + c_{i+n-1} - c_{i-1} * (b^n \bmod p)) \bmod p$$

As you can see, the calculation of the next fingerprint is done with a constant number of mathematical instructions, independent of the length of the subsequences. The term $b^n \bmod p$ is a constant than can be precomputed.

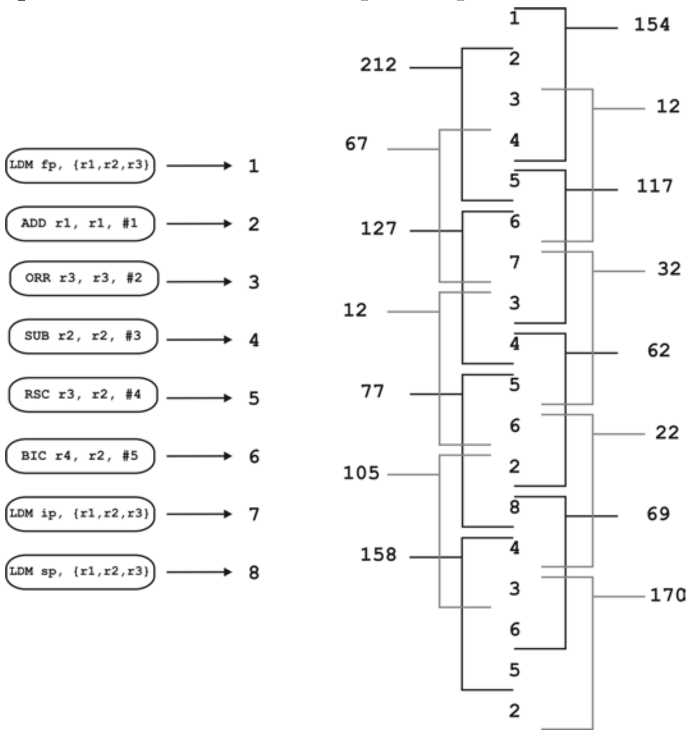


Figure 5 Rabin Karp method

As can be seen in Figure 5, this approach is able to find the matching instructions with hash value twelve. Of course, these matches must be compared instructions by instruction afterwards, since a matching hash value is not a guarantee matching instruction sequences.

Graph Mining

The main advantage of using graph mining for clone detection relies on the possibility of reordering instructions in program code. The fragment extracted in Figure 1 contains order variations that have to be detected.

The general idea of graph mining is to detect all frequent subgraphs in a database of general graphs (Meinl & Fischer, 2005; Washio & Motoda, 2003). A fragment is frequent, if it can be embedded in at least a given number of database graphs. This number is called *support.* The support can also be given as a fraction of the database size (normally in per cent) and is than called *frequency*. Support and frequency express the same attribute.

To determine a fragment as frequent, you have to detect all database graphs the fragment is a subgraph of. A brute-force attempt for one fragment is to iterate over all database graphs and explicitly check and count subgraph isomorphism. But the subgraph isomorphism problem is known as NP-complete so it is in general hard to decide if one graph is a subgraph of another. Because graph mining implicitly solves the subgraph isomorphism problem all general graph mining attempts are also NP-complete.

A better approach is to store and reuse implicitly the results of previous subgraph tests. If *B* is a subgraph of *C* and *A* a subgraph of *B*, *A* is also a subgraph of *C*. If *A* is no subgraph of *C*, the supergraph *B* of *A* also cannot be a subgraph of *C*. These dependencies can be used to minimize the number of subgraph tests.

To find all frequent subgraphs, the already found ones are stepwise extended to new bigger ones. This way, out of an initial set of subgraphs and a correct set of extension rules, all subgraphs are traversed. Common initial subgraphs are all frequent graphs with just one node or with just one edge. Normally, the graphs are extended by adding a new edge and simultaneously a new node, if it is required. Elsewhere a new edge between existing nodes is inserted that leads to cyclic fragments. So it is assured that the old graph is a subgraph of the new one without any explicit subgraph isomorphism test.

The whole search space for an algorithm can be arranged as a lattice which expresses the extensions of each fragment to the bigger ones. For general graphs, a fragment normally can result out of different extensions of different parent fragments.

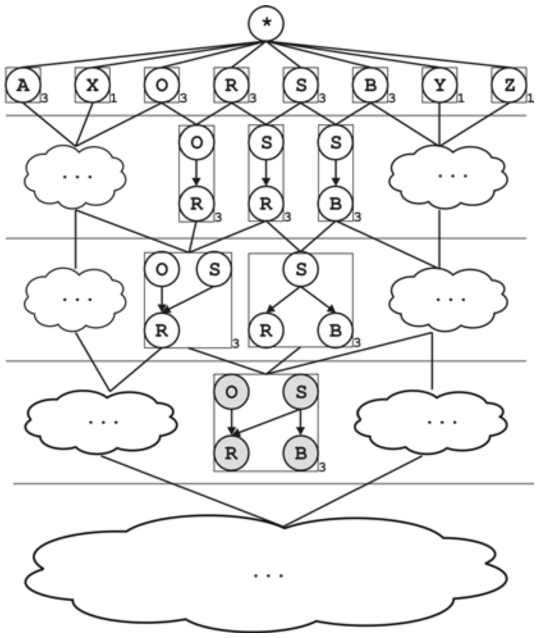


Figure 6 Extract of the whole search lattice for the example in Figure 1.

Because of the huge number of subgraphs, partly shown in Figure 6, it is necessary to traverse this search lattice as efficient as possible. That means that all frequent fragments and less as possible infrequent fragments have to be checked for frequency.

Thanks to the construction of the search lattice the children and grandchildren of a fragment have a frequency equal or less the fragment, as shown with the small indices in Figure 6. As said above an extended fragment may only be subgraph of the database graphs, the parent fragment also occurs in, because the extended fragment is a super graph to its parent. Therefore, it might just exist in the same or less number of database graphs, and so cannot have a higher support.

The bigger the fragment, the smaller its support and thanks to this antimonotony, an extension of infrequent fragments in the lattice is not necessary. Just infrequent ones emerge out of these fragments. This is similar to the frequency antimonotonicity principle in frequent item set mining and removes most infrequent fragments.

Additionally to this frequency pruning, each fragment can have different children **and** different parents in the search lattice and therefore different paths lead to the same fragment. Graph miners are interested in all fragments, but each fragment just once, so during the traversal of the search lattice those multiple reachable fragments have to be detected. If a fragment is reached a second time, all its children were still expanded before, so an additional expansion is not necessary and will create just more duplicates.

To detect those duplicates different methods are used in different mining algorithms. On the one hand, a newly found fragment can be compared with all previously found frequent fragments. These tests are done just for those fragments that are frequent, because for the required graph isomorphism test no polynomial algorithm is known and the complexity of graph isomorphism is not yet proven. It seems to be a complexity between polynomial and NP-complete. With the help of several graph properties (like node/edge count, node/edge label distribution …) two fragments can also be tested for isomorphism, so complete graph isomorphism test can be reduced to a bearable level.

A problem for that approach is that with an increasing number of found frequent fragments, the required tests become more and more often and so the search slows down. Another approach without this benefit is to build a unique representation for each fragment and just compare these representations (Yan & Han, 2002). Different codes are used, i.e. a row-wise concatenation of the adjacency matrix of the graph. Such a code is normally not unique for one graph as there are different possible adjacency matrices for one graph. Therefore one special representation, normally the lexicographical smallest or biggest one, is defined as the canonical form for a grown fragment. The complexity of the graph isomorphism test is hidden in the test for being canonical, but this test is independent of the number of found graphs.

An efficient method for canonical forms is to create an ordered list of comparable tuples representing the edges in the fragment. This edge list can easily be generated during the expansion process out of the order the edges are inserted to the graph. As above one list (i.e. lexicographic smallest) is used as the canonical representation. This sequence represents one path through the search lattice. If the algorithm generates an edge sequence which is not the canonical representation for the corresponding fragment, it reaches this fragment over an alternative way than the preferred one, so the search can be pruned at this point. Depending on the canonical form the expansion rules can be adjusted to generate fewer duplicates. Not all possible extensions, but just those with the possibility to generate a new canonical representation need to be followed.

In addition to those structural pruning rules, specialized pruning rules for different application areas are possible. I.e. if just trees or graphs with maximal ten nodes are relevant, than the search can be pruned if a fragment becomes cyclic or still has too much nodes. For a comparison of the most popular graph mining algorithms see Wörlein, Meinl, Fischer & Philippsen (2005).

The main adaption to use graph mining algorithms for clone detection is to build the graph database out of program code. The reorderable instructions, the graph mining approach rely on, is not represented in the pure code sequence. Therefore, common known structures from compiler techniques are available. The so called *data flow graphs* (DFG) or *control/data flow graphs* (CDFG) express concurrent executable instructions, that have a fix order in the code sequence. A set of DFGs build out of the basic blocks of a ARM binary as in the beginning of this chapter is the basis for the following approaches. For each basic block (code with one entry and one exit point without jump instruction inside) a separate graph is generated.

A standard graph mining run on these graphs results in all fragments occurring in a special number of basic blocks. For clone detection it is more important to know how often a fragment occurs in the database. The number of basic blocks it occurs in is irrelevant. If a fragment appears twice in a basic blocks, both appearances should be counted. An occurrence of a fragment in a basic block is called an embedding of this fragment. The number of extractable embeddings is equal to the number of independent, non-overlapping embeddings of a fragment. Two embeddings of the same fragment overlap if they use some or all nodes in common. In this case, after extracting one of the embeddings the second occurrence will be destroyed and cannot be extracted additionally. The best set of non-overlapping embeddings is calculated out of a collision graph containing all overlapping embeddings. Each node of the collision graph represents one embedding of the fragment and there is an edge between two nodes if the corresponding embeddings overlap. An independent set of such a graph is a subset of nodes, so that between two nodes of the subset there is no edge in the original graph. A maximal independent set of this collision graph represents the maximal number of embeddings that can be extracted together. To detect such subsets, general algorithms are available (Kumlander, 2004).

The number of extractable embeddings for a fragment is also antimonotone as the original graph count, so frequency pruning is still applicable. For clone detection fragments that just occur twice but are quit big, are also interesting in contrast to small ones with higher frequency. So frequency pruning is possible but the resulting search space is still quite big. A more extensive description is given in Dreweke, Wörlein, Fischer, Schell, Meinl & Philippsen (2007).

To keep the search still bearable some special properties of clone detection can be used. The created DFGs are directed acyclic graphs (DAG), so the fragments found are also DAGs. Not every fragment is extractable. The resulting graph (without the extracted fragment) still has to be acyclic, so embeddings/fragments leading to cyclic structures are irrelevant. For some fragments it is provable that each child and grand child also leads to cyclic structures, so the corresponding search branch can be pruned.

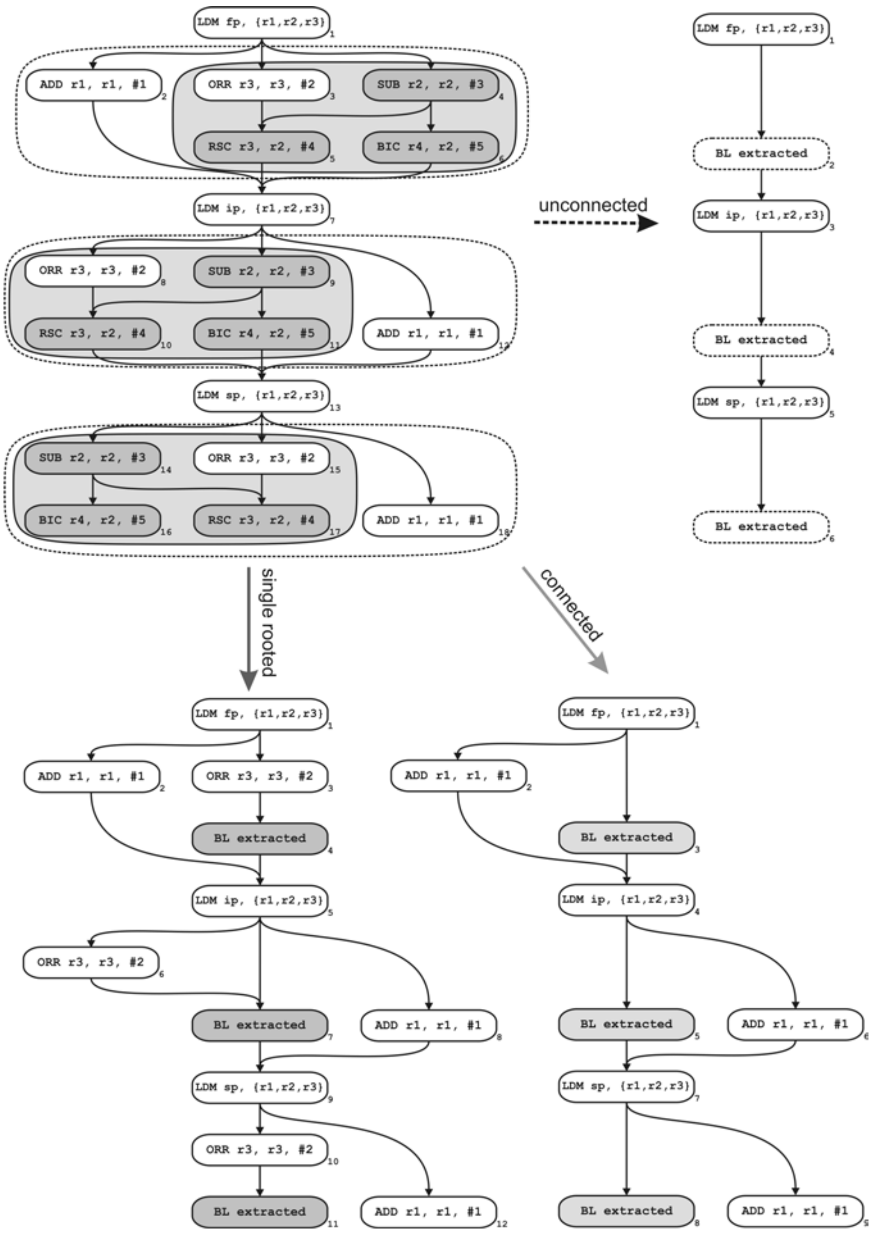


Figure 7 Detected single-rooted, connected, and unconnected code clones and their extraction

To simplify the required search, most graph based approaches reduce their search space to single-rooted fragments. The graph for a single-rooted fragment has just one entry point. A search for single-rooted fragments on the DFG for the example of Figure 1 results in the most frequent fragment consisting of the shaded nodes in Figure 7 and the resulting code (without the extracted procedure) in the bottom left.

A search for general connected graph fragments detects additionally that each fragment can be extended by the white ORR-node to the shaded fragment in Figure 7 so a connected search results in the code on the bottom right.

Connected graph mining does not detect the optimum presented in Figure 1. By extending the search to unconnected fragments the optimal fragment for a single extraction is found, but unconnected mining increases the number of subgraphs and the size of the search lattice dramatically. Figure 7 shows the corresponding optimal pattern as the dashed fragment in and the resulting code (without the extracted procedure) in the first line.

FUTURE TRENDS/CONLUSION

Text mining in program code is commonly known as clone detection, but is used in a wide variety of application areas. In this chapter we mainly focused on software engineering and embedded systems. Much research was done twice because of the different names for clone detection in the different areas. There are also many overlappings with research in graph mining. This chapter described the similarities and differences between the manifold approaches with varying goals and showed that graph mining algorithms can be applied very successfully in clone detection.

Large software systems often produce "legacy" code, evolving over several versions. Studies showed that up to thirty percent of the code in large software systems, such as the JDK, are code clones. The more code clones are within a software, the more complex it is to maintain the code. Therefore the demand for tools supporting the programmer while writing and maintaining the code is growing more and more. This demand is satisfied by searching similarities mostly by using graph mining related approaches. Code clones need to be similar, not necessarily identical in order to be interesting. Therefore future research work will probably use other approaches that do have a smaller computational complexity. One of these concepts could e.g. be the representation of program text as a vector and clustering code clones with respect to their Euclidean distance. This should reduce the complexity significantly. Since software systems are not always written in one single language, tools that support more than one language or can easily be extended are preferable.

In the area of embedded systems, the programs need to be as small as possible in order to reduce manufacturing costs for the large number of pieces. The code clones are extracted by various techniques. Duplicate code means semantically identical code, which must not be contiguous, but can be reordered. Additionally, semantics must not change if variables have been renamed. Instructions, furthermore, not even have to be connected in some dependence graph (reflecting data or control flow dependencies), if they stay extractable afterwards. Studies showed that up to five percent of program code for embedded systems can be reduced by extraction techniques.

Both application areas have to deal with large, computational difficult problems, handling many lines of code and, therefore, an enormous search space. Besides, not every code clone or duplicate is interesting for each application. "Interesting" in the area of embedded systems mostly means extractable, preferably saving as much instructions as possible. In the area of software engineering, "interesting" code clones may be "copy & paste" clones that are similar to a certain degree. The needs for the "interesting" duplicates and for an reduced complexity can be satisfied simultaneously. Not the whole search space has to be searched, but some branches of the search tree can be pruned without reducing the set of interesting clones too much. This is possible by development and application of appropriate heuristics.

Another common way to reduce the complexity of text mining is to restrict the maximal size of the code clones -- typically the limitation is set to twenty instructions. In software engineering applications, this restriction is also reasonable from another point of view, because programmers cannot review larger code blocks easily. In embedded systems, the code clone size restriction and the associated decrease of runtime has to be carefully opposed the instruction savings.

New areas of applications of text mining in program code or clone detection are available. For example, for teaching or software patents the original author of the code is relevant. Therefore, automatically checks to detect plagiarism are useful. Two programs or files can be compared, if one evolved into the other or if they are written completely independent by comparing their sequential code and semantic structures.

REFERENCES

Amit, C., Sudhakar, K., Phani, S., Naresh, S., & Rajesh, G. (1998). A general approach for regularity extraction in datapath circuits, *IEEE/ACM international conference on Computer-aided design* (pp. 332-339). San Jose, CA, USA: ACM Press.

Angew. Informatik GmbH, S. (2006). aipop. 2007, from http://www.AbsInt.com/aipop

Beszdes, R., Ferenc, R., Gyimthy, T., Dolenc, A., & Karsisto, K. (2003). Survey of code-size reduction methods. *ACM Comput. Surv., 35*(3), 223-267.

Brisk, P., Kaplan, A., Kastner, A., & Sarrafzadeh, M. (2002). Instruction generation and regularity extraction for reconfigurable processors, *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (pp. 262-269). New York, NY, USA: ACM Press.

Brisk, P., Macbeth, J., Nahapetian, A., & Sarrafzadeh, M. (2005). A dictionary construction technique for code compression systems with echo instructions, *ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems* (pp. 105-114). Chicago, IL, USA: ACM Press.

Cooper, K., & McIntosh, N. (1999). Enhanced code compression for embedded RISC processors, *ACM SIGPLAN 1999 conference on Programming language design and implementation* (pp. 139-149). Atlanta, GA, USA: ACM Press.

Debray, S., Evans, W., Muth, S., & Sutter, B. (2000). Compiler techniques for code compaction. *ACM Transactions on Programming Language Sysems., 22*(2), 378-415.

Dreweke, A., Wörlein, M., Fischer, I., Schell, D., Meinl, T., & Philippsen, M. (2007). Graph-Based Procedural Abstraction, *Fifth International Symposium on Code Generation and Optimization* (pp. 259-270). San Jose, CA, USA: IEEE Computer Society.

Fraser, C., Myers, E., & Wendt, A. (1984). Analyzing and compressing assembly code, *SIGPLAN symposium on Compiler construction* (pp. 117-121). Montreal, Canada: ACM Press.

Further, S. (1996). *ARM System Architecture*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

Kastner, R., Kaplan, A., Memik, S. O., & Bozorgzadeh, E. (2002). Instruction generation for hybrid reconfigurable systems. *ACM Transactions on Automated Electronic Systems, 7*(4), 605-627.

Komondoor, R., & Horwitz, S. (2003). Effective, Automatic Procedure Extraction, *IEEE International Workshop on Program Comprehension* (pp. 33-43): IEEE Computer Society.

Kumlander, D. (2004). A new exact Algorithm for the Maximum-Weight Clique Problem based on a Heuristic Vertex- Coloring and a Backtrack Search, *5th International Conference on Modelling, Computation and Optimization in Information Systems andManagement Sciences: MCO 2004* (pp. 202-208). Metz, France: Hermes Science Publishing Ltd.

Liao, S., Devadas, S., & Keutzer, K. (1999). A text-compression-based method for code size minimization in embedded systems. *ACM Transactions on Automated Electronic Sysems, 4*(1), 12-38.

Meinl, T., & Fischer, I. (2005). Subgraph Mining. In J. Wang (Ed.), *Encyclopedia of Data Warehousing and Mining* (pp. 1059-1063). Hershey, PA, USA: Idea Group.

Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica, 3*, 249-260.

University of Gent, B. Diablo. 2007, from http://www.elis.ugent.be/diablo

Washio, T., & Motoda, H. (2003). State of the art of graph-based data mining. *SIGKDD Exploration Newsetter., 5*(1), 59-68.

Wörlein, M., Meinl, T., Fischer, I., & Philippsen, M. (2005). A quantitative comparison of the subgraph miners MoFa, gSpan, FFSM, and Gaston. In A. Jorge, L. Torgo, P. Brazdil, C. R. & J. Gama (Eds.), *Knowledge Discovery in Database: PKDD 2005* (Vol. Lecture Notes in Computer Science, pp. 392- 403). Porto, Portugal: Springer.

Yan, X., & Han, H. (2002). gSpan: Graph-Based Substructure Pattern Mining, *IEEE International Conference on Data Mining (ICDM'02)* (pp. 721-724): IEEE Computer Society.

Zaretsky, D., Mittal, G., Dick, R., & Banerjee, R. (2006). Dynamic Template Generation for Resource Sharing in Control and Data Flow Graphs, *19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design* (pp. 465-468): IEEE Computer Society.

KEY TERMS AND THEIR DEFINITIONS

Code Compaction: The code size reduction of binaries in order to save manufacturing costs or of source code in order to increase maintainability

Clone Detection: Methods to identify code clones i.e. a sequence or set of instructions that is similar or even identical to another one in different kinds of programming code.

CDFG: Control Data Flow Graph representing the control flow and the data dependencies in a program.

Embedded System: Unlike general purpose systems, an Embedded System is used and built for special purpose with special requirements (e.g. real-time operation) and is produced with a large number of units. Therefore, tiny cost savings per piece pay off often.

Graph Mining: Methods to identify frequent subgraphs in a given graph database.

Fingerprinting: Code fragments are associated with numerical (hash) codes to speed up the detection of code clones. If two code blocks are identical, they have the same fingerprints. If two code blocks have identical fingerprints, these blocks do not necessarily have to be identical.

Procedural Abstraction: The extraction code clones into functions and replacing their occurrences by calls to these new functions.

Regularity Extraction: The realization of code clones in hard logic on an embedded system.

Slicing: Method to identify code clones based on a CDFG of the program. Based on instructions or their operational code isomorphic subgraphs are grown; similar to graph mining.

Suffix Tree: A suffix tree is a data structure used for an efficient detection of duplicated strings in a text.