Graph-Based Procedural Abstraction

A. Dreweke, M. Wörlein, I. Fischer, D. Schell, Th. Meinl, M. Philippsen
University of Erlangen-Nuremberg, Computer Science Department 2, Martensstr. 3, 91058 Erlangen, Germany, {dreweke, woerlein, schell, philippsen}@cs.fau.de
ALTANA Chair for Applied Computer Science, University of Konstanz, BOX M712, 78457 Konstanz, Germany, {Ingrid.Fischer, Thorsten.Meinl}@inf.uni-konstanz.de

Abstract

Procedural abstraction (PA) extracts duplicate code segments into a newly created method and hence reduces code size. For embedded micro computers the amount of memory is still limited so code reduction is an important issue. This paper presents a novel approach to PA, that is especially targeted towards embedded systems. Earlier approaches of PA are blind with respect to code reordering, i.e., two code segments with the same semantic effect but with different instruction orders were not detected as candidates for PA. Instead of instruction sequences, in our approach the data flow graphs of basic blocks are considered. Compared to known PA techniques more than twice the number of instructions can be saved on a set of binaries, by detecting frequently appearing graph fragments with a graph mining tool based on the well known gSpan algorithm. The detection and extraction of graph fragments is not as straight forward as extracting sequential code fragments. NP-complete graph operations and special rules to decide which parts can be abstracted are needed. However, this effort pays off as smaller sizes significantly reduce costs on mass-produced embedded systems.

1 Introduction

As in the early days of computing, code-size optimization is important, especially for embedded systems [7]. Cost and energy consumption depend on the size of the built-in memory. More functionality fits into a given memory size. Whereas earlier approaches to automatic code-size reduction focused on fast algorithms that kept the compile-time short, in the area of mass-produced embedded systems, a larger time budget can be spent on optimizations since the cost per piece is relevant in mass production. Manual code squeezing (although done in practice) is costly, time-consuming, and error-prone and can only be afforded once towards the end of the development process. Hence, batch optimizations that can be done much earlier and more often improve the current practice. Even a night or weekend of optimization is worthwhile since the resulting savings when building cars or cell phones are enormous. Moreover, off-line optimizations of machine code can even be applied to code that is only available in compiled form.

Of course there are compiler flags that avoid code bloat and there are smart linkers [38, 41] that reduce code size. But still, there are space-wasting code duplications that are mainly caused by the compiler's code generation templates, by cut-and-paste programming, by use of standard libraries, and by translation technology for templates/generics (although the latter are not yet in daily use in the embedded world) .

The most important technique to deal with code repetitions is *Procedural Abstraction* (PA). The technical questions are first to find repeated pieces of code, second to construct a unifying new code segment that can be executed instead, and third, to redirect program execution whenever one of the original pieces is reached (and back). For each PA an evaluation is needed to determine whether it pays off at all (both with respect to size and runtime as it is not wise to factor out hot code [16]). Finally there is a global optimization problem to be solved: a code piece moved to a new procedure can overlap in various ways with other frequent code pieces. Greedy strategies as used in our current approach reach acceptable results as we show in Section 4.

This paper only deals with the first technical question, the detection of repeated code fragments and makes use of known results for the other tasks. In contrast to the traditional way that considers the whole program as a sequence of instructions [22], our idea is to transform it into its data flow graphs. Since in these graphs different and more general candidates for PA can be found, graph-based PA increases the number of instructions that can be outlined.

Given the data flow graphs of all basic blocks, it is a complex problem to search for common graph fragments. It is NP-complete to detect that a single graph fragment is part of another (subgraph isomorphism). Things get even more complicated by the fact that the fragment itself must first be found. This so-called *frequent subgraph mining* problem has been well studied in the last few years, and many algorithms exist (see [32, 43] for an overview). However, the problem requires an exponential effort in the general case.

The rest of the paper is organized as follows. Section 2 describes Procedural Abstraction in general and our special approach. After a brief introduction to graph mining, Section 3 presents a miner for data flow graphs. After some benchmark results in Section 4, we describe future work and conclude.

2 Procedural Abstraction in General

Code-size reduction as surveyed in [8] comprises code compression and code compaction. The former requires a compression and decompression step. The decompression is done at run-time and can be implemented in hardware. Storage is needed to hold the decompressed code. Code compaction, the main focus of this paper, is an orthogonal approach producing shorter code that can be executed immediately.

The main focus of code compaction based on procedural abstraction is to detect repeated code fragments that can be extracted afterwards. Early (and fast) compaction approaches considered the whole code as a sequence of instructions and used suffix tries to detect common subsequences [23]. It has been shown in [18] that better results can be achieved when a program's analysis the control flow is taken into account. The authors consider basic blocks and single-entry-single-exit regions of code instead of a flat sequence of instructions. For each such unit, they compute fingerprints. Whereas two basic blocks can have the same fingerprint only if they differ just with respect to register names, two basic blocks with different fingerprints can never be outlined into a single procedure. A quick comparison of fingerprints speeds up duplicate detection. Currently there are two major shortcomings of the detection of code duplication in Procedural Abstraction. First, even approaches that are based on a whole program analysis and on basic blocks and control flow graphs, consider instructions *within* a basic block only in their given order (except for some special cases) [18]. Second, tools like aiPop [3] and Diablo [19] that implement PA require a specific compiler tool chain and cannot work on arbitrary binaries. Our Procedural Abstraction is a pure post link-time optimization that does not impose requirements on the compiler tool chain used. All the information needed to perform the procedural abstraction is extracted from the binary itself.

2.1 Our approach to Procedural Abstraction

Modern compilers already perform classical optimizations that are effective in reducing code size like dead code elimination or common-subexpression elimination. Nevertheless,there are many opportunities to reduce program size post link-time. For this reason our portable, modular, and retargetable framework for post link-time code optimization has to deal with classical binary translation and rewriting problems as described in [40]. The framework is structed in the following units:

1. The statically linked program is transformed into an internal instruction representation by decompiling the binary into a sequence of assembler instructions.
2. The sequence of assembler instructions is then divided into functions.
3. A architecture specific analysis and code preprocessing phase then identifies and marks all *jump* and *call* targets. For each target a *label* instruction is inserted into the instruction sequence. By introducing *labels*, *jumps* become independent of the original addresses and the actual sequence of assembler instructions. The reason for doing so is that hence PA removes parts from the code. Tn the new code, labels have changed addresses, i.e. jumps with explicit addesses will be buggy.
4. In addition to direct *jump* and *call* instructions, there are often *jump* instructions that target addresses relative to the *instruction counter*. If the exact target of these instructions can be determined, *label* instructions are also added at the targets. Hence, we can now completely abstract from the use of specified addresses.
5. By means of the new labels introduced in phases (3) and (4), the code can be split into basic blocks. Depending on the architecture there will be some blocks, that neither follow a conditional jump nor have a *label*. In general, these blocks can contain interwoven data, that is used on fixed-width architectures to address the main memory. If the optimizer can prove that one address of such a block is used as data, the whole block can be considered to be interwoven data and can hence be excluded from subsequent PA processing. Other blocks might be reached via function pointers (register indirect *jump* targets). Although it cannot be decided by points-to analysis in general, in code for embedded systems, the targets of function pointers can typically be determined [5].[1]

[1]Our current prototype only comprises a basic points-to analysis. However, there are ways to avoid illegal PAs of blocks of unknown type.

```
ldr r3, [r1]!
sub r2, r2, r3
add r4, r2, #4
ldr r3, [r1]!
sub r2, r2, r3
ldr r3, [r1]!
add r4, r2, #4
```

Figure 1. Running example: Basic Block of ARM Code.

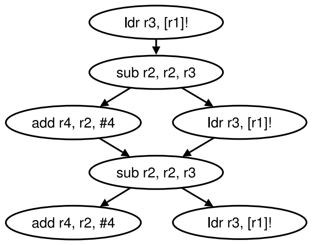


Figure 2. Running example: Data Flow Graph for the ARM code in Fig. 1,2

6. The resulting data flow graphs (DFG) are input of the subsequent mining process. All instructions of a basic block are analyzed to determine the dependencies between them.

7. On this set of DFGs the miner searches for frequent code fragments. The one with the highest code size reduction benefit (determined by its size and its number of appearances) is marked.

8. As various found code fragments can overlap, only one of the marked code fragments is extracted per mining step. Two different extraction methods are used: if a fragments ends with an unconditional return statement or a branch instruction, a cross jump or tail merge is used [16,17]. Otherwise a subroutine call is necessary. After extraction, phase (6) is repeated as long as code fragments are found that reduce the overall number of instructions in the program.

2.2 A Running Example

We explain the basic ideas of our approach to PA with a (synthetic) piece of ARM code given in Fig. 1 and its corresponding DFG shown in Fig. 2. Our example steps through the values of an array (register `r1` points to the current element in the array) and performs some calculations.

Suffix trie based approaches to PA [22] detect the code fragment `ldr r3,[r1]!, sub r2,r2,r3`

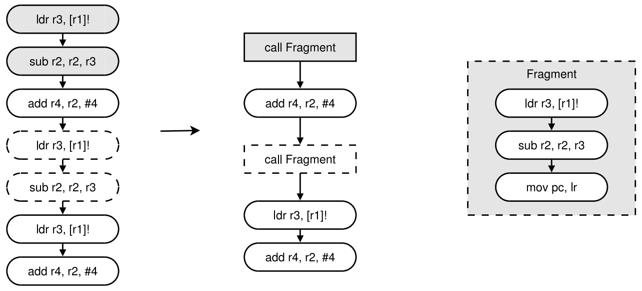


Figure 3. Suffix trie-based PA for the running example (Fig. 2).

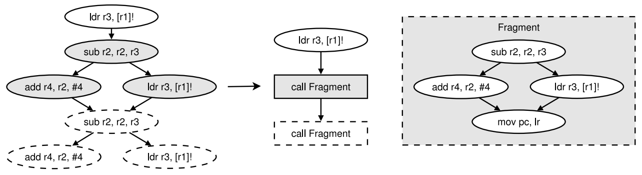


Figure 4. One option of graph-based PA for the running example (Fig. 2).

twice. Assume that two appearances are frequent and the size of two instructions per appearance is big enough for outlining the fragment into a procedure. Fig. 3 shows the resulting $5 + 3 = 8$ instructions.

Graph-based PA is more successful, even for this tiny example. Fig. 4 and Fig. 5 show the extraction of two different fragments of size three that both appear twice and that both result in $3 + 4 = 7$ instructions. The varying order of instructions prevents suffix tries from detecting these fragments.

The graph miner discussed below discovers all possible fragments for PA. Based on a cost/benefit heuristic (over the size and the frequency of each fragment) one of these candidates is extracted.

3 Mining of directed Graphs

3.1 Introduction

For several years now, graph mining is a hot topic in the data mining community. The main idea is to find the most frequent graph fragments in a graph database. For our approach to PA, the given graphs are the data flow graphs of the program's basic blocks. The frequent graph fragments

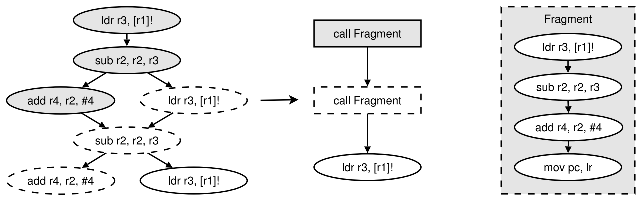


Figure 5. Another option of graph-based PA for the running example (Fig. 2).

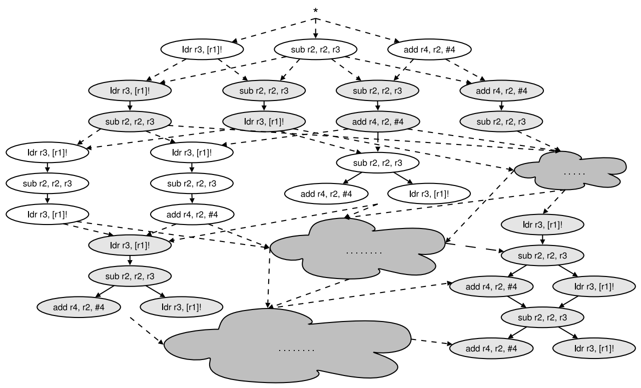


Figure 6. Parts of the search lattice for the running example (Fig. 2).

of the DFGs may be code pieces that can be extracted. Data flow graphs of basic blocks are directed acyclic graphs.

A common way of searching for frequent graph fragments is to arrange all possible subgraphs in a lattice. Fig. 6 shows a part of this lattice for the subgraphs of the DFG shown in Fig. 2. The empty subgraph (*) at the top is followed by the subgraphs with just one node (i.e. one instruction). Each step downwards adds a new edge (and node, if necessary) to a subgraph. In our example lattice, the second level contains graph fragment with two nodes and one edge. On the third level there are three nodes. Only an edge without an additional node must be inserted to close the circle (if seen as undirected graph) in the middle of the running example. At the end, all graphs of the given graph database are given completely. In our example lattice the final DFG is given in the right bottom corner. In Fig. 6 the . . . indicate missing parts because the full lattice is too big to be printed (even for this small example). Real search lattices are even more complex because the underlying databases consist of many different and often much bigger graphs.

There are many different ways how this lattice can be searched for frequent graph fragments. Breadth-first and depth-first are the most common options, but also priority-based versions are known. For each fragment in the lattice, the number of embeddings has to be calculated during the search. If this number is higher than the given frequency, a fragment is considered frequent. In the running example two fragments have frequency 2 (Fig. 4, 5).

As can be seen in the lattice, there are many different paths to reach the same subgraph. So a traversal may find (and extend) the same subgraph multiple times a resource consuming problem if a big graph database is searched. To avoid such duplications a graph miner has to detect duplicates. Two different approaches can be found in the literature. Either graph isomorphism tests are used or a canonical form is generated from the graphs. Based on this canonical form a search strategy is defined that avoids most of the duplicates.

Compared to the vast majority of graph miners as described in [32, 43] there is an important difference to the graph miner needed for PA. We are interested in every occurrence of a frequent graph fragment (i.e. code fragment) especially if this frequent code fragment appears several times in one graph (i.e. basic block). In both Fig. 4 and Fig. 5 two fragments appear twice. As has been shown above, both occurrences (also called *embeddings*) are important for PA and can be outlined. However, classical graph miners do not count the two occurrences but they only count the single graph in which the fragments appear. Thus, classic graph miners will find fewer frequent fragments than embedding-based miners. But if embeddings are counted, the different occurrences may not overlap, as only one of two overlapping occurrences can be outlined into a procedure.

Pruning is another important technique to reduce the search space. In the literature several pruning strategies can be found. Most important is frequency pruning: a child in the lattice of an infrequent parent can not be frequent. This does hold either if frequency is fragment-based or embedding-based. In the latter case, the embeddings must be edge-disjoint [31]. In PA overlapping embeddings are not interesting, as only one of them can be extracted, so this condition holds.

In the remainder of this chapter, the algorithm used is explained along these general lines. First, the related work is sketched.

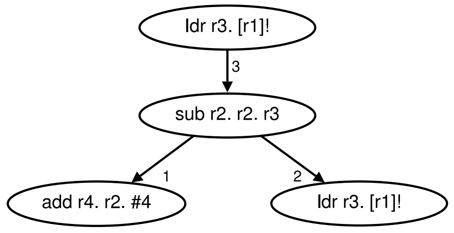
3.2 Related Work

In graph mining a lot of different algorithms have been published in recent years. Most algorithms originate from bio- or chemoinformatics where molecules or proteins are mined [9, 27, 28, 31, 36]. In this application area undirected graphs are mined based on the number of graphs a fragment occurs in.

Graph mining based on the number of occurrences is done only in five different miners. In [15, 35] possible candidates for frequent fragments are searched greedily based on different heuristics. This does not ensure that all possible candidates and all possible frequent subgraphs are found. Ghazizadeh and Chawathe [25] developed an algorithm that works well only when the input graphs contain a relatively small number of frequent subgraphs with high frequency, and is not effective if there are a large numbers of frequent subgraphs with low frequency (as in typical program code). Vanetik et al. [42] presented an algorithm for undirected graphs only. Their algorithm is also not based on completely disjoint graph fragments but on edge disjoint graph fragments. Occurrences are edge disjoint, if they do not share any edges, node sharing is allowed. Also the approach presented in [31] is based on edge disjoint graph fragments. Using edge disjoint graph fragments is not useful in the context of PA since each node represents a disassembled instruction and as such can only be extracted once (and not as part of several frequent code fragments).

In the area of template generation for field-programmable gate arrays (FPGAs) similar problems arise. High-level synthesis compilers often reproduce reoccurring patterns. To reduce the number of functional units, these patterns are extracted to efficiently use hardware. This so called *template generation* is similar to the mining of frequent patterns in graph mining. In [34] five heuristics are presented to detect these patterns in a DFG. Brisk et al. [10] use a DFG-based approach to identify frequent patterns for a hard-logic implementation on embedded systems. The approach is similar to [35]. Ten years earlier [39] had similar ideas. The heuristics do not ensure that all relevant patterns are detected. In [48] the DFG is searched step-wise and a string representation similar to a canonical form is used to substitute the subgraph isomorphism test. Only fragments with one root node can be found, so also this algorithm may miss frequent fragments. A heuristic and a look ahead is used to grow fragments. It is a popular idea to mine for trees or single rooted DAGs as in this case the complexity of the algorithm is lower [14]. Other approaches in this area use libraries with predefined templates [12] to simplify the main problem our paper wants to solve. Of course, they can not guarantee that the predefined templates occur often in a special piece of code.

Clone detection as described in [6] works on source code and tries to identify duplicated identical code ("clones") or nearly identical code ("near miss clones") by analyzing the abstract syntax trees (ASTs) of a program. The identification of duplicated code must be done by a programmer by hand or by rather limited tools. Tools like "Dup" [4] are limited to find textually identical sections of code and sections that are the textually identical except for systematic



#	nodeA	nodeB	labelA	labelB	direction
1	0	1	sub	add	out
2	0	2	sub	ldr	out
3	0	3	sub	ldr	in

Figure 7. A small example for a valid DFS-code.

substitution of variable names and constants. Beside the limitations in the clone search, the main focus as described in [4, 6] is to extract the duplicates in a semantic-preserving way into functions or more preferred into macros (to avoid unacceptable performance degradation [29]). As the macros will be expanded by the preprocessor during compilation the resulting binary will not be smaller at all. The main target of clone detection therefor lies more in the field of structuring code during the developing process instead of code size reduction.

Dictionary compression as presented in [11] also starts from DFGs. In this paper a similar approach to [10] by the same authors is used. Dictionary compression mechanisms identify redundant sequences of instructions that occur in a program. The sequences are extracted and copied to a dictionary. Each sequence is then replaced with a codeword that acts as an index into the dictionary, thereby enabling decompression of the program at runtime.

It is interesting to see, that the fields of graph mining and template generation solve similar problems with similar techniques, but the corresponding papers do not cite results of the opposite area. By combining the results of both fields, huge progress seems possible.

3.3 DgSpan: gSpan for directed graphs

As no graph miner was available that fits the requirements of PA we developed a new miner based on the well-known *gSpan* algorithm [45, 46]. It has been shown that gSpan has the best properties of all graph miners [37, 44].

The original gSpan algorithm works on undirected graphs and just counts how many graphs are in the database in which a frequent fragment appears, instead of counting all embeddings. It defines a special canonical form (the so called *DFS-code*) to detect duplicates during the traversal of the lattice. The DFS-code is a sorted list of tuples that describe each edge of the subgraph in the order in which it is attached to the subgraph. gSpan uses a depth-first search. In Fig. 7 a small example for such a code is given. The table contains the DFS-code for the given DFG. Each line in the table is one tuple describing the last edge (and its node) added to the DFS-code. The small numbers attached to the edges of the DFS show the insertion order. The first edge added is the one from `sub` to `ldr`. First the node indices are given where `sub` has index 0 and `ldr` has index 1. The node labels and the edge direction follow. Together they form the DFS code of this edge. Registers are not included for simplicity. Other edges are added in the same way. For ordering the node-labels we define $sub < add < ldr$ and $out < in$. This way different insertion orders and their DFS-codes can be compared lexicographically. As there are different paths to one subgraph in the lattice, there are also different DFS-codes for each fragment. In gSpan, the canonical form is defined as the lexicographic smallest possible list of tuples. By attaching an edge to a graph its list just grows at the end. It can be shown that the parent of each canonical form is still a (smaller) canonical form and that all extensions of a non-canonical code cannot lead to another canonical form. Therefore, the traversal of the lattice can stop as soon as non-canonical form is reached. This speeds up the search. Additionally, extensions that obviously lead to non-canonical codes can also be ignored for further speed-up (except for some special extensions [45]).

The original algorithm is for undirected graphs only, but due to the order of the nodes in the edge-tuples, the direction of an edge can simply be expressed by the label or by an additional flag. Thus the basic algorithm needs only small modifications. In the remainder of this paper, the directed version of gSpan is called **DgSpan**.

3.4 Detection of overlapping embeddings

Embedding-based graph miners may only count non-overlapping embeddings. Fig. 8 shows the example DFG and two embeddings of a subgraph of size 4. It can be seen that both embeddings use the same `ldr`-instruction (both shaded and with a dashed line). Since the two embeddings overlap, only one of them can be extracted from the code. Since the `ldr`-instruction is moved to the procedure, it is missing afterwards, so it can not be extracted again.

In order to detect non-overlapping embeddings we construct, a so-called *collision graph*. Its nodes are the embeddings. An edge connects two embeddings if they have at least one instruction in the DFG in common. For such pairs of nodes only one can be extracted. The challenge is now to find the biggest subset of all embeddings, so that no two embeddings share a node in the DFG. This corresponds to finding a *maximum independent set* of nodes in the collision graph (or a maximum clique in the inverted collision

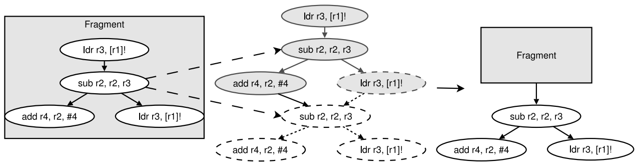


Figure 8. There are two ways to embed the Fragment (left) into the DFG that is shown in the middle. Since the two embeddings overlap, only one can be outlined by PA. When the upper embedding is extracted (right), the other embedding disappears.

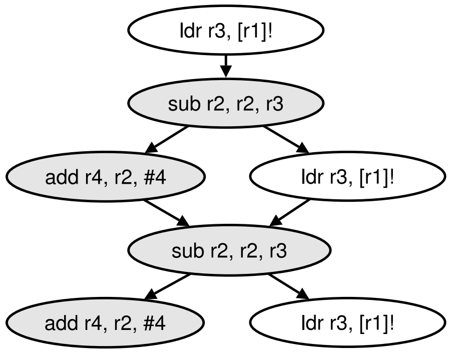
graph). A number of algorithms for this NP-complete problem exists [21, 30]. We combined gSpan with the algorithm described in [30] which at the moment is the fastest known algorithm. The main advantage of this algorithm of Kumlander is its backtracking and its graph coloring pruning heuristics. Nodes of the collision graph are colored heuristically and sorted to determine an order. According to that order the maximum cliques of the induced graphs of the last node, the last two nodes, the last three nodes, and so on are computed iteratively.

Calculating the non-overlapping embeddings of a fragment also leads to a useful pruning technique. Each parent of a frequent graph is still frequent (with respect to the number of embeddings it has). So all children of an infrequent graph can be pruned from the search tree. This is similar to the *frequency antimonotone principle* in frequent item set mining [2, 47].

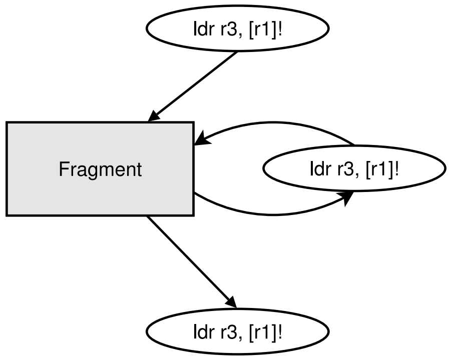
3.5 PA-specific search space pruning

Whereas the traditional suffix trie-based PA can outline any found sequence of sufficient length, in the DFG-based PA, identified frequent code fragments need to survive additional plausibility checks before extraction. Traditional approaches did not touch the instruction order that was generated by the compiler. Our new approach only works on the data dependencies between the instructions. While this increases the number of frequent code fragments, it also causes new problems.

Fig. 9(a) shows the example DFG with a frequent fragment marked in Gray. If this identified embedding would be extracted into a method, the graph would change as displayed in Fig. 9(b). The new graph now contains a cycle. The `ldr`-instruction of the cycle needs to be executed both before and after the fragment is called. But because all instructions can only be executed in sequence, there is no way to satisfy the cyclic data dependency and therefore the em-



(a) Example DFG with a frequent fragment marked in gray. The other embedding into a different basic block is not shown.



(b) Illegal extraction caused by cycle in the data flow.

Figure 9. PA can not extract all the embeddings found.

bedding must not be extracted.

If it can be shown that a branch of the search lattice only contains such unextractable fragments this branch can be pruned to further speed up the search.

In the remainder of this paper, our Embedding-based graph-miner, an extension of DgSpan that consider embeddings, that uses a maximum independent set (MIS) to avoid overlapping embeddings, and that performs PA-specific pruning will be called **Edgar**.

Summing up, our mining approach for procedural abstraction consists of the following steps: First DFG are generated. These DFGs are mined with our new version of gSpan based on directed graphs. The number of non-overlapping embeddings as calculated with [30] after each mining step is used as frequency. After a round of mining and PA specific pruning rules, the fragment that shrinks the binary code most is extracted from the code and the whole process restarts until all frequent fragments have been found.

4 Evaluation

To evaluate our Procedural Abstraction framework, a subset of the MiBench suite [26] is used. As most embedded systems only run one specific application, there is no need for dynamic libraries. Therefore all of the test programs were compiled with the *gcc* to a static binary. All binaries were compiled with the `-Os` switch that tells the compiler to optimize for size: all optimizations that result in faster but larger code (like e.g. loop unrolling and function inlining) are omitted and additional optimizations, that try to further reduce the size of a binary (like e.g. strength reduction or common subexpression elimination), are added. Using the `-Os` option therefore results in the smallest possible code with almost no code redundancy left. In order to even further decrease the size of the binaries, we linked against *dietlibc* [20], a cross platform C run-time library that supports x86, ARM, Sparc, Alpha, PPC, Mips, and s390 architectures. It is compatible to SUVv2 and POSIX and aims to be the smallest compared to *glibc* or *uclibc*. The *dietlibc* is optimized for statical linkage and therefore uses special coding techniques to reduce the code size. For example all modules (e.g. stdio and unistd) are strictly separated and most functions are compiled into separate object-file, allowing the linker to reduce the size of an application by only linking the used functions into the binary. Another very important point is that the *dietlibc* developers reused code wherever possible without using copy-paste or huge C-macros. Because of these techniques the resulting binaries are about 17 to 20 times smaller and contain almost no redundancy compared to binaries produced with *glibc* or *uclibc*. One drawback of using *dietlibc* for compiling the test programs however is that only a subset of the MiBench suite can be compiled and linked. This is because *dietlibc* is not yet fully feature-complete compared to *glibc* and some of the programs rely on special *glibc* features that are not implemented in *dietlibc*. While benchmarking of binaries that have seen a lot of inlining and that are linked with libraries containing redundant code would result in favorable numbers, our performance analysis focuses on the hardest cases, i.e., on code that is already optimized for size and that is linked with the smallest available libraries.

4.1 The ARM Architecture

Although our technique is universally applicable and we work on other platforms as well, this paper only presents results for ARM. Readers familiar with ARM can safely skip this section. The ARM architecture [24] is a popular embedded architecture. The ARM design is highly extendable with a common 32- bit RISC instruction core. Recent models also incorporate a 16- bit instruction set called *Thumb* [1] which we did not yet consider in our prototype implemen-

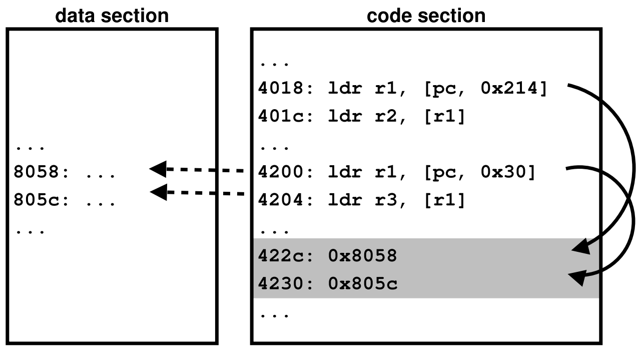


Figure 10. ARM binary code section with indirect loads (straight line arrows) of interwoven addresses (shaded).

tation.

The ARM register set consists of fifteen 32-bit general purpose registers (`r0` to `r14`), a dedicated program counter (`PC`), and the *current program status register* (`cpsr`). There are also eight floating point registers and a corresponding status register. The high number of registers causes the data flow graphs to be more dense than for other architectures. Hence with more registers our new approach to PA is more effective.

Since the ARM architecture with its fixed instructions width does not support 32-bit numbers (like absolute addresses) as immediate operands. Main memory addresses can not be addressed directly. The most effective and common solution, on architectures with an explicit `pc` register (like ARM), for this problem is to load the addresses from memory. This is done by a `pc`-relative indexed load operation. Because of the 12-bit limitation, the data must be somewhere near the current `pc` position. Therefore it must be interwoven with the instructions.

Fig. 10 shows how data can be interwoven in between the assembler instructions in the code section. The instruction at address `0x401c` in the code section of the program needs to access the data stored at address `0x8058` in the data section. Because the compiler only knows that the instruction needs access to that data but not where the instruction is placed in the final program, the compiler adds an literal pool (depicted in gray) to the program code. To access the data at address `0x8058` the compiler has to insert another load instruction (at address `0x4018`), that loads the address of the data (in the data section) from the literal pool into register `r1`. The data in `r1` is dereferenced and the sought-after data is finally loaded into `r2`.

Table 1. Saved instructions in the benchmark suite.

Program	# Instructions	# of saved instructions		
		SFX	DgSpan	Edgar
bitcnts	3946	53	81	83
crc	3584	46	68	119
dijkstra	4632	70	102	164
patricia	5039	70	105	189
qsort	4770	70	105	197
rijndael	7113	70	132	256
search	3717	46	74	110
sha	3897	55	82	120
total	36698	480	749	1238

4.2 Measurements

Table 1 compares the graph-based approach with the traditional suffix trie or fingerprint-based approach (SFX) [16, 23]. The percentile savings we found for SFX are similar to what other researchers have found in their benchmarks for SFX. For graph mining we used two different approaches applied to the set of basic block DFGs. First, we tested DgSpan, our extension of the gSpan algorithm [45] for directed graphs that counts in how many graphs a fragment occurs. Second, we used Edgar, our extension of DgSpan, that is embeddings-based, avoids overlapping embeddings with the detection of maximal independent sets, and applies some PA-specific pruning. For both graph miners, the best fragment found was extracted and the miners were restarted until no further shrinking was possible.

In total, Edgar saves 1238 instead of 480 instructions by SFX which is an improvement by a factor of 2.6 or about 160 % on average; see Fig. 11. Compared with SFX, graph-based PA is more successful in code shrinking. The absolute number of saved instructions may seem small, but remember that they reveal the worst case since (a) we started from the smallest possible inputs (other papers describing procedural abstraction [13] do not start from programs this small) and (b) potential embedded hardware is only available in discrete sizes so even a single saved byte might allow to select a smaller and cheaper chip for a mass-produced item.

The results of Edgar are better than the ones of DgSpan. Obviously, many fragments appear several times in some basic blocks. Since this remains unnoticed by DgSpan it considers these fragments to be infrequent. Edgar's counting of each individual embedding causes detection and hence extraction of these fragments.

For the *rijndael* program, Edgar shows the best improvement over SFX (256 versus 70 saved instructions, improve-

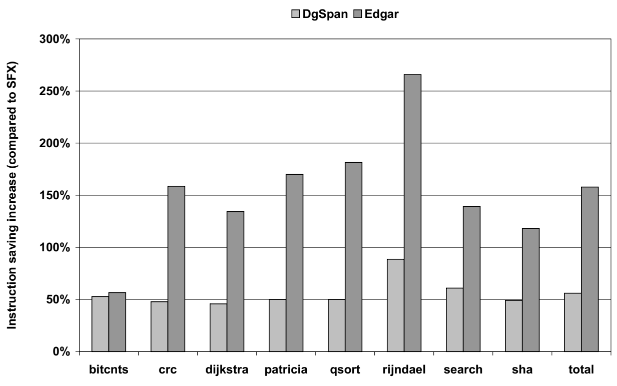


Figure 11. Relative increase of savings of graph-based PA compare to suffix trie; both with DgSpan and Edgar.

Table 2. Number of instructions with $(degree_{IN} \vee degree_{OUT}) > 1$ in all DFGs that are used for mining.

Program	$degree > 1$	$degree \leq 1$
bitcnts	859	2150
crc	730	1966
dijkstra	932	2616
patricia	1010	2851
qsort	980	2702
rijndael	2542	3541
search	776	2081
sha	834	2121
total	8663	20028

ment factor 3.7 or 266 %). Due to the nature of the encryption algorithm, the compiler generates many very similar code sequences. But in order to speed up the execution, these instructions are then reordered and rescheduled to overlap load operations with computation. Hence, the traditional suffix trie and fingerprint approaches cannot identify most of the duplicates.

For the worst program of the benchmark collection, *bitcnts*, Edgar "only" saves 28 more instructions than traditional SFX, which is still an improvement of 52 %. This is more than expected at first, since this program which only processes the given input and calculates the number of bits needed to represent it, does not offer as much optimization potential as other test programs.

To understand why the graph-based PA can outline so much more code compared to the suffix trie approach, it is instructive to study the shapes of the dependency graphs. There would be no potential to reorder instructions at all if the nodes of the dependency graphs formed plain chains, i.e., if the first node of a basic block was followed by a unique second node, which was followed by a unique third node, etc. Thus, all instructions but the first and last instruction of each basic block have exactly one predecessor and one successor, and therefore an in- and outdegree of one (i.e., all the nodes of the dependency graph have an indegree ≤ 1 and an outdegree ≤ 1).[2] In that case, SFX and Edgar would find the same duplicates. In the benchmark programs however, more than one third of the nodes have a higher fan-out or a higher fan-in, see Table 2. This results in varying instruction orders and hence common fragments that can only be detected by the graph-based approach.

On average, DgSpan takes 50 seconds to optimize a program on a typical desktop machine. Because of the embeddings and the computations of the maximal independent set, Edgar takes longer, namely about 90 seconds per program on average.

The optimization time for the *rijndael* program exceeds these averages by far. On the sequential desktop machine DgSpan takes 2 hours and 32 minutes, Edgar works for 4 hours and 22 minutes. With parallel implementations of graph mining that cannot be discussed here, we have achieved significant speedups on SMP machines (up to 10 on 12 processors), see [33]. The high runtimes for bigger and more complex applications like the *rijndael* encryption algorithm are acceptable as PA can by applied as a batch optimization that can be done much easier (e.g. at night time or over weekends) as manual code squeezing that is even more time-consuming and error-prone. As embedded systems are produced in huge amounts of pieces the high optimization effort is acceptable (otherwise there would not be human 'code-squeezers' working on assembly code today). The savings done to lower hardware requirements because of smaller code outweight the costs for optimizing the application code for a few hours.

To explain the huge variations in optimization time, the DFG graphs must be analyzed. Table 3 gives the number of instructions in the DFG representation and their connection to other instructions. As can be seen the benchmark programs have many instructions that are quite independent of other instructions. Hence many nodes ha a higher degree. Due to this high independents a lot of new fragments compared to suffix tries can be found.

Compared to the other programs, *rijndael* has a higher fraction of nodes with higher degree. This higher branching causes a far more complex and bigger search lattice with more paths to the fragments and hence many more duplications that must be identified by costly isomorphism tests.

[2]DFGs of basic blocks with just one instruction have in- and outdegree 0.

Table 3. Indegree and outdegree of all instructions.

Progam	Type	Degree				
		0	1	2	3	≥ 4
bitcnts	**In**	447	2048	404	61	47
	Out	355	2208	350	64	32
crc	**In**	418	1836	341	54	47
	Out	327	1992	291	55	31
dijkstra	**In**	548	2429	444	77	50
	Out	430	2634	382	66	36
patricia	**In**	590	2659	465	91	56
	Out	464	2877	400	77	43
qsort	**In**	574	2511	458	81	58
	Out	449	2734	380	75	44
rijndael	**In**	538	3839	**1410**	**186**	**110**
	Out	408	4120	**1256**	**205**	**94**
search	**In**	435	1955	367	57	43
	Out	343	2118	311	56	29
sha	**In**	451	2004	378	71	50
	Out	355	2174	322	66	38
total	**In**	4001	19281	4267	678	464
	Out	3131	20857	3692	664	347

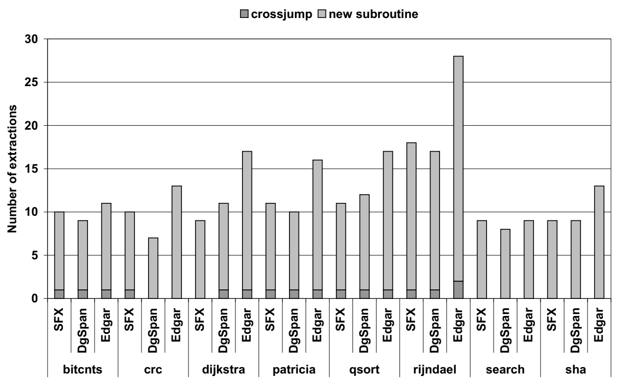


Figure 12. Extraction mechanisms used by SFX, DgSpan, and Edgar.

Additionally the *rijndael* program with 7113 instructions is the biggest program in our test suite[3].

Finally, Fig. 12 displays how often the two extraction methods are used. In all test constellations, *cross jump* extraction occurs seldom since to be applicable, a fragment must end with a (rare) `return` or `jump` instruction. Otherwise the fragment is moved into a new procedure.

5 Conclusion and Future Work

On certain programs (like *rijndael*) graph-based PA saves up to 3.6 times more instructions than traditional suffix trie-based approaches. On average, instruction savings was increased by a factor of 2.6. This is mainly caused by the fact that –in contrast to traditional approaches– graph-based PA can exploit the potential of instruction reordering.

Concluding we can say that for development of embedded systems where a batch optimization is acceptable and where a few more saved bytes might allow to select a smaller chip for mass-produced items, procedural abstraction based on DFGs is superior to traditional approaches.

In future work we will address the long runtimes of the graph-based PA. Although the DFGs used for graph-based PA are directed and acyclic graphs (DAGs), the gSpan algorithm that currently is the basis of our implementation is a general graph miner and does not exploit the special structure and attributes of DAGs. It seems that there are better and faster ways to search through a lattice of DAGs. First, the detection of duplicates might be optimized by a specialized canonical form for DAGs. Second, the computation of the maximum independent set (MIS) can be improved as follows. Since there are no cycles in DAGs, a direction can be assigned to both graphs and frequent fragments. Not only does the number of potential embeddings shrink since embeddings must respect this direction. But in addition, special collision patterns for the MIS-computation result since only "neighboring" embeddings can overlap. A specialized MIS algorithm for collision graphs with these patterns should be much faster than the general algorithm.

Another optimization will be fuzzy instruction matching. In our current implementation the instructions of a frequent fragment's embeddings must be completely identical. Instead of mining for identical instructions one can mine for instructions that are *canonically* equal. In a *canonical* representation two instructions are equal if they have the same *mnemonic* and the same number and types of operands. Fig. 13 shows ARM instructions and their *canonical* representation where `R` denotes a register and `I` an immediate value.

To even further increase the number of saved instructions the search area will be widened. Our prototype is currently restricted to basic blocks. The search area can be increased incrementally. In a first step, instructions of basic blocks that are too small for outlining will be merged with their predecessing basic blocks. In a second step, for each procedure the graph consisting of control and data flow will be considered. Hence, arbitrary segments of procedures that may span several basic blocks might be outlined. Third,

[3]When linked with glibc instead of dietlibc *rijndael* would have 91891 instructions.

```
ldr r3, [r1]!          ldr R, [R]!
sub r2, r2, r3         sub R, R, R
add r4, r2, #4         add R, R, I
```

(a) ARM assembler instructions. (b) Canonical representation.

Figure 13. ARM assembler code and their canonical representation.

with the help of the *call graph* and the CFGs of the individual procedures, an *ICFG* (interprocedural CFG) is built for the whole program. Similar as in the second step the information of the *ICFG* is then used to create only one DFG for the whole program. This DFG then represents the maximum search area, because all instructions in all basic blocks are combined in one graph, in which we will find the maximal number of fragments.

References

[1] Advanced RISC Machines Ltd. (ARM). *An Introduction to THUMB*, March 1995.

[2] R. Agrawal, T. Imielinski, and A. N. Swami. Mining Association Rules between Sets of Items in Large Databases. In P. Buneman and S. Jajodia, editors, *Proc. 1993 ACM SIGMOD Int'l Conf. on Management of Data*, pages 207–216, Washington, D.C., USA, 1993. ACM Press.

[3] aipop. AbsInt Angew. Informatik GmbH, Saarbrücken, http://www.AbsInt.com/aipop.

[4] B. Baker. On finding duplication and near-duplication in large software systems. In *Second Working Conference on Reverse Engineering*, pages 86–95, Los Alamitos, CA, USA, 1995. IEEE Computer Society.

[5] V. Barthelmann. *Advanced Compiling Techniques to Reduce RAM Usage of Static Operating Systems*. PhD thesis, Universität Erlangen–Nürnberg, Erlangen, Germany, 2004.

[6] I. Baxter, A. Yahin, L. De Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the 1998 International Conference on Software Maintenance*, pages 368–377, Bethesda, Maryland, USA, 1998. IEEE Computer Society.

[7] J. Bentley. Programming Pearls: Squeezing Space. *Communications of the ACM*, 27(5):416–421, May 1984.

[8] Á. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto. Survey of code-size reduction methods. *ACM Computing Surveys*, 35(3):223–267, Sept. 2003.

[9] C. Borgelt and M. Berthold. Mining Molecular Fragments: Finding Relevant Substructures of Molecules. In *Proc. IEEE Int'l Conf. on Data Mining ICDM*, pages 51–58, Maebashi City, Japan, 2002. IEEE Computer Society Press.

[10] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *Proceedings of the Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 262–269, New York, NY, USA, 2002. ACM Press.

[11] P. Brisk, J. Macbeth, A. Nahapetian, and M. Sarrafzadeh. A dictionary construction technique for code compression systems with echo instructions. In Y. Paek and R. Gupta, editors, *Proceedings of the Conf. on Languages, Compilers, and Tools for Embedded Systems*, pages 105–114, Chicago, USA, June 2005. ACM.

[12] T. Callahan, P. Chong, A. DeHon, and J. Wawrzynek. Fast module mapping and placement for datapaths in FPGAs. In *ACM/SIGDA Int'l Symp. on Field Programmable Gate Arrays*, pages 123–132, Monterey, CA, 1998.

[13] W. Cheung, W. Evans, and J. Moses. Predicated instructions for code compaction. In *7th Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, number 2826 in Lecture Notes in Computer Science, pages 17–32, 2003.

[14] A. Chowdhary, S. Kale, P. Saripella, N. Sehgal, and R. Gupta. A general approach for regularity extraction in datapath circuits. In *IEEE Int'l Conf. on ComputerAided Design (ICCAD)*, pages 332–339, 1998.

[15] D. J. Cook and L. B. Holder. Graph-based data mining. *IEEE Intelligent Systems*, 15(2):32–41, 2000.

[16] B. De Sutter, B. De Bus, and K. De Bosschere. Sifting out the mud: Low level c++ code reuse. In *Proceedings of the 17th ACM SIGPLAN Conf. on Object-oriented programming, systems, languages, and applications*, pages 275–291, New York, NY, USA, 2002. ACM Press.

[17] B. De Sutter, H. Vandierendonck, B. De Bus, and K. De Bosschere. On the side-effects of code abstraction. In *Proceedings of the 2003 ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems*, pages 244–253, New York, NY, USA, 2003. ACM Press.

[18] S. K. Debray, W. S. Evans, R. Muth, and B. de Sutter. Compiler Techniques for Code Compaction. *ACM Trans. on Progamming Languages and Systems*, 22(2):378–415, Mar. 2000.

[19] Diablo. Parallel Information System Group, University of Gent, Belgien, http://www.elis.ugent.be/diablo.

[20] dietlibc - a libc optimized for small size. http://www.fefe.de/dietlibc/.

[21] F. V. Fomin, F. Grandoni, and D. Kratsch. Measure and Conquer: A Simple $O(n^{0.288n})$ Independent Set Algorithm. In *Proc. of the 17th ACM-SIAM Symp. on Discrete Algorithms: SODA '06*, pages 18–25, Miami, Florida, USA, Jan. 2006. ACM and SIAM.

[22] C. Fraser, E. Myers, and A. Wendt. Analyzing and compressing assembly code. In *Proceedings of the ACM SIGPLAN '84 Symp. on Compiler Construction*, pages 117–121, New York, NY, USA, 1984. ACM Press.

[23] C. W. Fraser, E. W. Myers, and A. L. Wendt. Analyzing and Compressing Assembly Code. In *Proc. 1984 ACM SIGPLAN Symp. an Compiler Construction*, pages 117–121, Montréal, Canada, June 1984.

[24] S. Furber. *ARM System Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[25] S. Ghazizadeh and S. S. Chawathe. Seus: Structure extraction using summaries. In S. Lange, K. Satoh, and C. H. Smith, editors, *5th Int'l Conf. on Discovery Science*, volume 2534 of *Lecture Notes in Computer Science*, pages 71–85, Lübeck, Germany, Nov. 2002. Springer.

[26] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th IEEE Annual Workshop on Workload Characterization*, pages 3–14, Austin, TX, USA, 2001. IEEE Computer Society.

[27] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Proc. of the 3rd IEEE Int'l Conf. on Data Mining ICDM*, pages 549–552, Melbourne, FL, USA, Nov. 2003. IEEE Press.

[28] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD '00: Proceedings of the 4th European Conf. on Principles of Data Mining and Knowledge Discovery*, pages 13–23, London, UK, 2000. Springer.

[29] R. Komondoor and S. Horwitz. Eliminating duplication in source code via procedure extraction. Technical report, Comupter Sciences Department University of Wisconsin-Madison, Madison, WI, USA, 2002.

[30] D. Kumlander. A new exact Algorithm for the Maximum-Weight Clique Problem based on a Heuristic Vertex-Coloring and a Backtrack Search. In T. H. A. Le and D. T. Pham, editors, *5th Int'l Conf. on Modelling, Computation and Optimization in Information Systems and Management Sciences: MCO '04*, pages 202–208, Metz, France, July 2004. Hermes Science Publishing Ltd.

[31] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. In M. W. Berry, U. Dayal, C. Kamath, and D. B. Skillicorn, editors, *Proceedings of the Fourth SIAM Int'l Conf. on Data Mining*, pages 243–271, Orlando, FL, USA, Apr. 2004. SIAM.

[32] T. Meinl and I. Fischer. Subgraph mining. In J. Wang, editor, *Encyclopedia of Data Warehousing and Mining*, pages 1059–1063. Idea Group, Hershey, PA, USA, 2005.

[33] T. Meinl, I. Fischer, and M. Philippsen. Parallel Mining for Frequent Fragments on a Shared-Memory Multiprocessor –Results and Java-Obstacles–. In M. Bauer, A. Kröner, and B. Brandherm, editors, *LWA 2005 - Beiträge zur GI-Workshopwoche Lernen, Wissensentdeckung, Adaptivität*, pages 196–201, 2005.

[34] S. O. Memik, G. Memik, R. Jafari, and E. Kursun. Global resource sharing for synthesis of control data flow graphs on fpgas. In *ACM/IEEE Design Automation Conference (DAC)*, pages 604–609, Anaheim, CA, June 2003.

[35] P. C. Nguyen, K. Ohara, H. Motoda, and T. Washio. Cl-gbi: A novel approach for extracting typical patterns from graph-structured data. In T. B. Ho, D. Cheung, and H. Liu, editors, *Advances in Knowledge Discovery and Data Mining, 9th Pacific-Asia Conference*, volume 3518 of *Lecture Notes in Computer Science*, pages 639–649, Hanoi, Vietnam, May 2005. Springer.

[36] S. Nijssen and J. N. Kok. The Gaston Tool for Frequent Subgraph Mining. *Proc. Int'l Workshop on Graph-Based Tools (Grabats), Electronic Notes in Theoretical Computer Science*, 127(1):77–87, 2004.

[37] S. Nijssen and J. N. Kok. Frequent subgraph miners: Runtime don't say everything. In T. Gärtner, G. C. Garriga, and T. Meinl, editors, *Proceedings of the Int'l Workshop on Mining and Learning with Graphs (MLG 2006)*, pages 173–180, Berlin, Germany, 2006.

[38] W. Pugh. Compressing Java Class Files. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 247–258, Atlanta, GA, USA, May 1999.

[39] D. Rao and F. Kurdahi. Partitioning by regularity extraction. In *ACM/IEEE Design Automation Conference*, pages 235–238, Anaheim, CA, June 1992.

[40] B. D. Sutter, B. D. Bus, and K. D. Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Trans. Prog. Lang. Syst.*, 27(5):882–945, 2005.

[41] F. Tip, P. F. Sweeney, C. Laffra, A. Eisma, and D. Streeter. Practical Extraction Techniques for Java. *ACM Trans. on Programming Languages and Systems*, 24(6):625–666, Nov. 2002.

[42] N. Vanetik, E. Gudes, and S. Shimony. Computing frequent graph patterns from semistructured data. In *Proc. of 2002 IEEE Int'l Conf. on Data Mining*, pages 458–464, Maebashi City, Japan, Dec. 2002. IEEE Computer Society.

[43] T. Washio and H. Motoda. State of the Art of Graph–based Data Mining. *SIGKDD Explorations Newsletter*, 5(1):59–68, July 2003.

[44] M. Wörlein, T. Meinl, I. Fischer, and M. Philippsen. A quantitative comparison of the subgraph miners MoFa, gSpan, FFSM, and Gaston. In A. Jorge, L. Torgo, P. Brazdil, R. Camacho, and J. Gama, editors, *Knowledge Discovery in Database: PKDD 2005*, Lecture Notes in Computer Science, pages 392–403, Porto, Portugal, 2005. Springer.

[45] X. Yan and J. Han. gSpan: Graph–based substructure pattern mining. In *Proceedings IEEE Int'l Conf. on Data Mining ICDM*, pages 721–723, Maebashi City, Japan, 2002. IEEE Computer Society Press.

[46] X. Yan and J. Han. Closegraph: Mining closed frequent graph patterns. In *Proceedings of the 9th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, pages 286–295, Washington, DC, USA, 2003. ACM Press.

[47] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New Algorithms for Fast Discovery of Association Rules. In D. Heckerman, H. Mannila, D. Pregibon, R. Uthurusamy, and M. Park, editors, *Proc. of 3rd Int'l Conf. on Knowledge Discovery and Data Mining*, pages 283–296, Newport Beach, CA, USA, Aug. 1997. AAAI Press.

[48] D. C. Zaretsky, G. Mittal, R. P. Dick, and P. Banerjee. Dynamic template generation for resource sharing in control and data flow graphs. In *Proceedings of the 19th Int'l Conf. on VLSI Design*, pages 465–468, Hyderabad, India, Jan. 2006. IEEE Computer Society.