Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

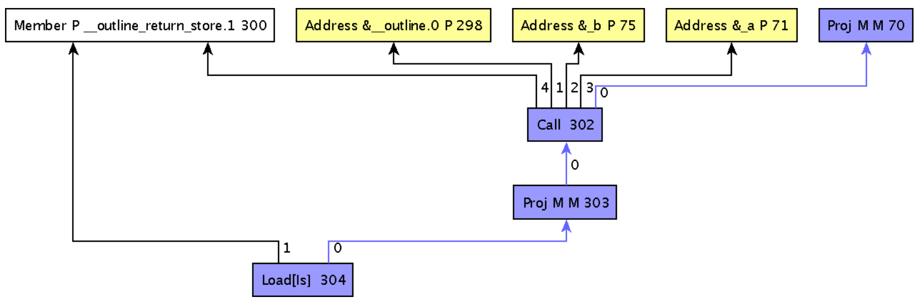
Lehrstuhl Prof. Dr.-Ing. Snelting

Identifying and Extracting Recurring Program Structures

Bachelorarbeit von

Petar Heyken

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuender Mitarbeiter:	M.Sc. Andreas Fried

Bearbeitungszeit: 21. November 2017 – 14. März 2018

Zusammenfassung
Abstract

Die Outline-Optimierung sucht nach ähnlichen Codesegmenten und wandelt diese in eine eigene Funktion um. Der hier vorgestellte Ansatz führt die Outline-Optimierung auf Compilerebene durch und sucht dabei nach isomorphen Subgraphen in einem graphgestützten Zwischencode und extrahiert die gefundenen Subgraphen in neue Funktionen. Danach werden die Vorkommen mit einem Aufruf der neuen Funktion ersetzt.

In dieser Arbeit wird ein einfacher, aber langsamer Algorithmus zum Finden von isomorphen Subgraphen, die Bedingungen, die für ein outlining-geeignetes Muster gelten müssen, sowie die Umstrukturierung des Graphen, um eine Funktion aufzurufen, vorgestellt.

Mithilfe der libFIRM C Test Suite und des SPEC2000 Benchmarks wird gezeigt, warum dass Outlining kaum die Laufzeit verbessert oder die Binarygröße verringert, aber die Compilezeit drastisch erhöht.

The outline optimization searches for similar code segments and transforms these into separate functions. Our approach performs the outline optimization on compiler level by searching isomorphic subgraphs in an graph-based intermediate representation and extracting the found subgraphs into new functions. Then, it replaces the occurrences with calls to the new function.

In this thesis, we describe a simple but slow algorithm for finding isomorphic subgraphs, the conditions that have to be met for a pattern to be eligible for outlining, as well as the rearrangements of the graphs for calling an outlined function.

Using the libFIRM C test suite and the SPEC2000 benchmark, we show that outlining hardly improves run time or decrease the binary size while increasing the compile time drastically.

Contents

1. Introduction

Even though it is in theory possible for a developer to write machine code, it is hardly practiced nowadays. High level languages are easier to read for a human. These high level source files have to be translated into machine code though; this is where a compiler takes over.

Although it is considered bad practice, source files tend to have duplicated code, as outlined in [1]. It is possible to either find duplicates at source code level or at compiler level, using the compiler's internal representation. We want to find duplicates at compiler level and take it a step further extracting the identified duplicates into their separate function and replacing the occurrences with a call to the function.

Our optimization is called outlining, in contrast to inlining. Inlining [2] is an optimization which takes a function whose size is under a certain threshold and replaces the call to the function with the content of the function. This improves the performance of the compiled binary since the overhead associated with a function call is eliminated and enables further optimizations. The increase in binary size is accepted since it is usually not of concern.

With outlining, we want to implement an optimization which creates a smaller binary for use cases where the binary size is a concern. We use the internal graph representation of libFIRM to find similar structures in the graph which have the same functionality.

This work is structured as follows: In Chapter 2, we give an overview of the compiler we want to extend as well as some other basics needed. The design and implementation is presented in Chapter 3. In Chapter 4, we give an overview of the testing conducted and the results. Finally, we draw a conclusion in Chapter 5.

2. Basics and Related Work

A compiler is used to translate high-level, human-readable programing languages into low-level machine code, which the computer can then execute. Usually, a compiler includes certain optimization routines to achieve faster execution times, or, like the outline optimization presented in this work, to achieve smaller binaries. In order to implement the optimizations only once and not separately for every programing language, an *intermediate representation* (IR) is used. A compiler using an IR is usually divided into three parts:

- Front end: takes the program's source, parses it, checks it for semantic errors, and then translates it into the IR
- Middle end: applies optimizations to the IR
- Back end: translates the IR to machine code

2.1. SSA form

Nowadays, IRs use *static single assignment* (SSA) form [3] to represent values. Every time a value is assigned to a variable, the variable is renamed and all following references to that variable are also renamed accordingly. Renaming the variables makes it visible on which values an operation depends, thus making it easy to convert code in SSA form into a data dependency graph, where every node represents a value. However, a variable can have different values depending on which control flow branch is taken. In order to be able to have branching control flow, IRs in SSA form use a ϕ-function which chooses the value according to the control flow.

Figure 2.1 shows an example of code that is not in SSA form and the code transformed into SSA form.

```
x = 3
x += 2
if a > x:
  x = 5
else:
  x = 6
a = x
```

a Code before transformation into SSA

```
x1 = 3
x2 = x1 + 2
if a1 > x2:
  x3 = 5
else:
  x4 = 6
a2 = φ(x3, x4)
```

b Code after transformation into SSA

Figure 2.1.: SSA example

2.2. CFG

A *control flow graph* (CFG) consists of basic blocks containing instructions. If control flow enters a basic block, all instructions in the block are executed in order and it is ensured that no jump to another part of the program happens during execution. Figure 2.2 shows the code introduced in Figure 2.1 separated into basic blocks and thus forming a CFG.

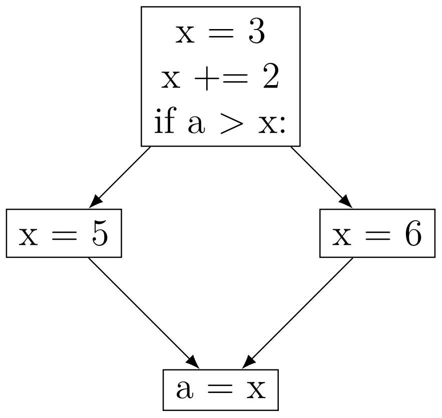


Figure 2.2.: CFG example

2.3. libFirm and cparser

libFIRM is a compiler library written in C that uses a graph-based IR [4, 5]. *cparser* [6] is a front end for compiling C programs which uses libFIRM. libFIRM implements common optimizations used in compilers such as constant folding, if conversion, and inlining.

2.3.1. libFirm's design

libFIRM combines the SSA form and a CFG in one graph, referred to as *ir_graph*. The ir_graph consists of nodes and edges. Nodes that are the result of the SSA form transformation are placed in blocks, where the blocks are nodes themselves. The edges represent dependencies among nodes. Furthermore, the paths through the ir_graph do not go from start to end, but from end to start since libFIRM implements a dependency graph. Every node has certain dependencies which are represented by the edges to the nodes.

Memory state The current state of the memory is represented by a node in libFIRM. Nodes whose operation requires memory access, such as load and store operations, need a memory state as an input, and they output a new memory state for the successors. The memory state depends on the control flow as well, therefore the ϕ-function is necessary for memory values as well.

Edges libFirm has three types of edges: control flow, data flow, and memory dependency. The type is implicit, libFirm reasons the edge type using the edge index and the source and target node. A special case is the edge with index $= -1$, it points to the block the node belongs to.

Modes Modes are primitive data types that can usually be matched directly to a target machine. libFirm has modes for different integers and floats, boolean, and pointer. Special modes for internal use include a memory mode to represent a memory state, control flow mode, internal boolean mode for conditional jumps, and tuple mode to represent that a node has more than one value.

Types libFirm types are created from the source language's data types. Primitive data types like integers, floats, and pointers can be directly transformed into modes. More complex data types like compound types (structs and classes) and arrays can be represented with libFirm's types.

2.3.2. Nodes in libFirm

Every node has an opcode which defines the node's operation. Every edge going out of a node represents a dependency on another node's value. In libFirm's implementation, the struct representing a node is called *ir_node*.

Block A block is a node which contains a set of nodes. Blocks have control flow edges pointing to the nodes that transfer the control flow to the block.

Start and End node In each ir_graph, there is exactly one Start and one End node. They represent the entry and exit of the ir_graph. The Start node has the initial memory, pointer to the frame base, and the function's arguments as its output. The End node has a set of keep-alive edges which point to and therefore hold on to parts of the graph that would otherwise not be connected, e.g. an endless loop which would otherwise not have a path to the end node.

Proj node In libFirm, a node represents exactly one value. Since nodes can have more than one value resulting from their operation, nodes with multiple values output a tuple with all values. Proj nodes are used to extract one value from the tuple for further use as input for other nodes.

Memory nodes *Memory nodes* is a term we use in this work. It includes all nodes in an ir_graph that have mode M and therefore only output a memory state. They handle the current memory state according to the operation of the node. Most common is the Proj M node, which extracts the memory state from the result of a memory operation. *Memory operation nodes* (like Store and Load) are not included in the memory nodes since they output the memory state in a tuple.

Phi node Phi nodes are the implementation of the SSA form's ϕ-function. The Phi node outputs the value corresponding to the control flow taken. It has to have the same number of arguments as control flow predecessors of the block it belongs to. The Phi node is also used to choose a memory state depending on the control flow.

Call node The Call node transfers control flow to another function. It has a memory state and a pointer to the function as its input, as well as the arguments for the called function. The arguments have to match the function's type. The Call node outputs the inputs of the Return node inside the called function, i.e. the last memory state and the return values.

Figure 2.3 shows a C source code file (Figure 2.3a) which is transfered into a libFIRM ir_graph (Figure 2.3b), and the resulting assembler output (Figure 2.3c).

2.4. Frequent subgraph mining

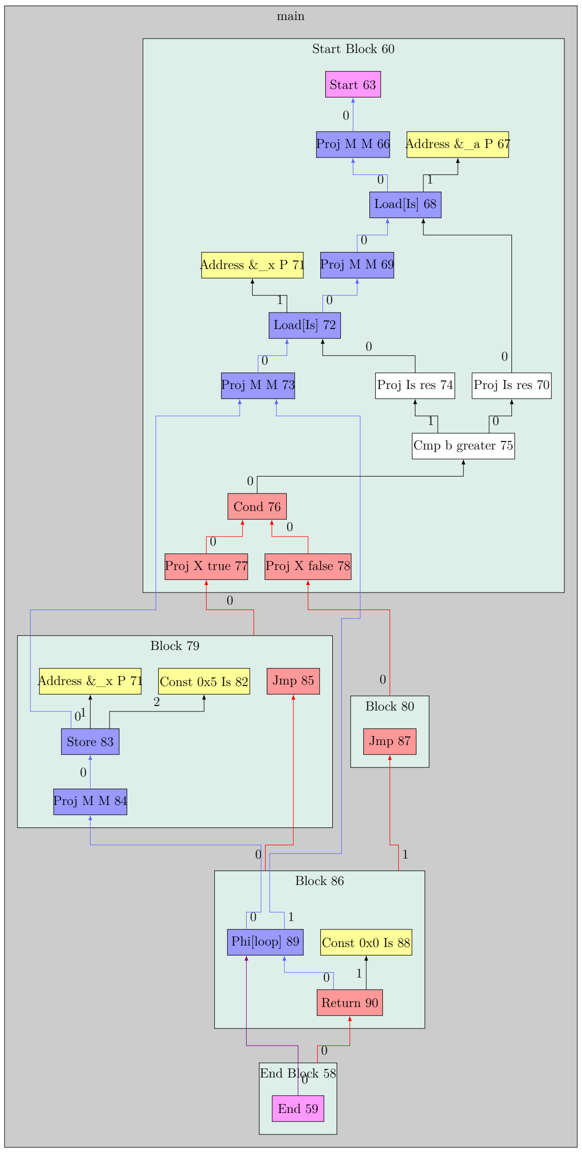
Jiang, Coenen, and Zito [7] give an overview of different *frequent subgraph mining* (FSM) algorithms, which are used to find isomorphic subgraphs. First, we need to introduce a few definitions though:

- *graph*: A graph can be represented as $G(V, E)$ with V being a set of vertices and $E \subseteq V \times V$ a set of edges.
- *labeled graph*: In addition, the graph has a set of labels for vertices L_V and edges L_E as well as a label function φ that defines the mappings $V \to L_V$ and $E \to L_E$.
- *subgraph*: $G_1(V_1, E_1, L_{V_1}, L_{E_1}, \varphi_1)$ is a subgraph of $G_2(V_2, E_2, L_{V_2}, L_{E_2}, \varphi_2)$ if:
  - $V_1 \subseteq V_2$
  - $\forall v \in V_1 : \varphi_1(v) = \varphi_2(v)$
  - $E_1 \subseteq E_2$
  - $\forall (u, v) \in E_1 : \varphi_1((u, v)) = \varphi_2((u, v))$
- *graph isomorphism*: G_1 is isomorphic to another graph G_2 if a bijection $f : V_1 \to V_2$ exists such that:
  - $\forall u \in V_1 : \varphi_1(u) = \varphi_2(f(u))$
  - $\forall (u, v) \in E_1 \iff (f(u), f(v)) \in E_2$
  - $\forall (u, v) \in E_1 : \varphi_1((u, v)) = \varphi_2((f(u), f(v)))$
- *subgraph isomorphism*: G_1 is subgraph isomorphic to G_2 if $\exists g \subseteq G_2 : G_1$ isomorphic to g

```
int a = 4;
int x = 0;

int main(void) {
    if (a > x)
        x = 5;
    return 0;
}
```

a C source code



b ir_graph before any optimization, Figure A.1 shows a bigger version

```
_main:
L0:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    xorl %eax, %eax
    movl _x, %edx
    cmpl _a, %edx
    jl L1
L2:
    movl %ebp, %esp
    popl %ebp
    ret
L1:
    movl $5, _x
    movl %ebp, %esp
    popl %ebp
    ret
```

c Assembler

Figure 2.3.: Example of a C source with its ir_graph and assembler output

It is not yet known whether graph isomorphism is solvable in polynomial time nor whether it is NP-complete, whereas subgraph isomorphism is known to be NP-complete. Notable algorithms which check for subgraph isomorphism are Ullmann [8], which uses backtracking and a lookahead function, and VF [9], which uses a depth-first-search strategy and feasibility rules.

There are two types of FSM algorithms, transaction-based and single graph-based. Single graph-based FSM algorithms only work on one graph, transaction-based ones can detect isomorphic subgraphs in independent graphs. Depending on the application of the FSM, a subgraph g is only considered frequent if either the occurrence-based or the transaction-based count is greater than the defined threshold. Occurrence-based counting counts every occurrence of the subgraph, transaction-based counting counts every graph in which there is at least one isomorphic subgraph. FSM algorithms are categorized by their approach as well. On the one hand, there are Apriori-based approaches which are based on a breadth-first-search-like traversal of the graph, e.g. gSpan [10]. On the other hand, pattern growth-based approaches traverse the graph in a depth-first-search-like manner, e.g DPMine [11].

DPMine seems to suit our use case best, however, we decided to implement a brute-force like method to begin with first in order to reduce the complexity of our outlining algorithm.

2.5. Identifying duplication in source code

In [12], Komondoor and Horwitz outline a method of finding duplicates in source code using forward and backward slicing, which seems to be the most promising approach. They use program dependence graphs (PDGs), which combine data and control dependencies into one graph, to find isomorphic PDG subgraphs. Nodes are program statements and predicates, edges are data and control dependencies. Their approach is divided into three steps: finding pairs of duplicates, removing subsumed duplicates, and combining pairs of duplicates into larger groups. In the first step, they partition all nodes into equivalence classes based on the nodes' syntactic structures. They call two nodes in the same class *matching nodes*. Backward slicing is used to extend the isomorphic subgraphs, which have only one node each in the beginning. Forward slicing is only used when the node is a predicate. With their slicing approach and the use of PDGs, they are able to find non-contiguous, reordered, as well as intertwined duplicates. Their testing showed, that their implementation found the clones a human would find as well, though it found many variants of the ideal clone. Their goal for the future is to find heuristics that reduce the number of variants, to improve the run time, and to implement an extraction of the found duplicates into a new procedure.

Our approach is similar to theirs since we search for isomorphic subgraphs in a likewise manner. However, we are able to find patterns which are not present at source code level. The compiler already applies some optimization before the outline optimization is run which can lead to other patterns. Furthermore, we implement an extraction of the found duplicates into a new procedure.

2.6. Terminology used

Before we present the design and implementation, we introduce the terminology used in the next chapter:

- *similar nodes*: Two nodes with the same operation that have the same certain attributes depending on the nodes' operation. The attributes usually include the mode and the number of predecessors. For some nodes though, more attributes have to be checked, e.g. two Cmp nodes[1] have to have the same comparison relation in order to be considered similar. The similar nodes relation is transitive, meaning if node A and node B as well as node B and node C are similar, then node A and node C are also similar.

- *isomorphic subgraph pair (ISP)*: Two sets of nodes which are both subgraphs of an ir_graph. In every subgraph, all nodes are reachable from the exit node. Every node in one subgraph has a corresponding node in the other subgraph, which is similar. If there is an ISP with the node sets A and B, and an ISP with the node sets B and C, then there is an ISP with A and C as well.
- *isomorphic subgraph group (ISG) / pattern*: One or more ISPs combined. The ISPs' transitivity is used to group them together into a pattern.
- *entry, exit*: We define the entry and exit of a pattern according to the control flow and not in reversed order like the edges in libFirm. This means that the algorithm for finding ISPs starts at the exit node and works its way up to the pattern's entry. When talking about edges, we will use the same direction as libFirm though, meaning that the path through the pattern goes from the exit to the entry.

[1]compares two inputs according to the nodes relation attribute, outputs a value of the internal boolean mode

3. Design and Implementation

This chapter explains the steps that the outline optimization goes through. First, patterns and their arguments have to be found. Then, three checks are run to ensure that the pattern is eligible. Finally, the pattern is extracted into a new function and replaced with a call to the function at every occurrence.

3.1. Finding the patterns

We divide finding possible patterns into two steps. Firstly, the algorithm searches for an ISP. The algorithm takes two similar nodes and then tries to expand the subgraphs by looking at the predecessors. Secondly, the optimization tries to group ISPs together in order to have a larger number of isomorphic subgraphs in one group, which are usually the preferred patterns, and in the course of grouping ISPs together transforms them into ISGs.

3.1.1. Finding the ISPs

The outline algorithm for finding ISPs uses a data structure as shown in Figure 3.1. It has two pointer sets, *graph1* and *graph2*, containing pointers to ir_nodes. Edges are saved implicitly, meaning if node A and node B are in the pointer set, the edge between those two nodes is part of the subgraph as well. *outer_nodes_graph1* and *outer_nodes_graph2* are arrays which contain nodes that are entry nodes to the subgraph during the expansion of the subgraph. *change* signals whether the subgraphs changed during the previous iteration or not and is therefore used as the criterion for terminating the algorithm. *deleted* shows whether the subgraphs are still considered for outlining or if the algorithm found a cause that makes the subgraphs ineligible for outlining.

The goal of the first step taken by the pattern finding algorithm is to create all ISPs with the minimal node count of eight since smaller patterns would lead to more nodes when outlined. To achieve this, all nodes in all ir_graphs are placed into pointer sets, each pointer set only containing nodes that are similar. Then, the algorithm creates the ISPs which always consist of two nodes that are in the same pointer set. The implementation of the algorithm is optimized though, as shown in Figure 3.2a. We use the same strategy that saves all similar nodes in a list of pointer sets (lines 2-4, 10-12), but the algorithm already creates and tries to expand the ISPs (lines 5-9) while the *similar_nodes_list* is still being populated. Therefore, the algorithm can disregard most of the ISPs immediately (lines 8-9), since the pattern

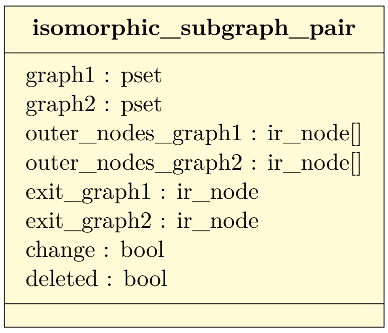


Figure 3.1.: isomorphic subgraph pair struct used by the algorithm

size is too small. This saves a huge amount of memory since most of the ISPs are smaller than the minimum node count.

The modus operandi of the expansion algorithm is shown in Figure 3.3. The graph is expanded in a breadth-first-search like manner. The nodes A_1 and A_2 are the exit nodes of the ISP as well as the outer nodes before the expansion of the ISP is started. Then, the algorithm checks whether the corresponding predecessors are similar and adds them to the pattern, as shown in Figure 3.3b. The B and C nodes are outer nodes and are considered in the next iteration. They add their similar predecessors in Figure 3.3c. Since there are no more nodes to expand the subgraphs after the second iteration, the D nodes are marked and the algorithm terminates as shown in Figure 3.3d. The pseudo code for this algorithm is outlined in Figure 3.2b (lines 2-6, 8, 21, 24).

Since a libFIRM ir_graph is far more complex than the example, we have to add additional constraints and functionality to the ISP expansion algorithm.

It is important that the nodes are exclusive to one subgraph and are not shared by the two subgraphs except for constlike[1] nodes. In Figure 3.4, one subgraph consists of the first two lines, the other subgraph consists of line two and three. If the two subgraphs were outlined, a would be incremented four times, which would obviously alter the semantics of the program and is therefore not desired.

If one predecessor node is already part of its subgraph but the other predecessor is not part of its subgraph, the ISP will be discarded. The subgraph where the predecessor is already part of it contains a cycle where the other does not. Therefore, they are not an ISP and are discarded by setting the deleted flag (Figure 3.2b, lines 9-11).

We have to check the relation of the current nodes' blocks and the predecessors' blocks. Nodes can only be added to the subgraph if they are either in the same block as the successors (lines 22-24) or the nodes' blocks are connected directly to the blocks of nodes' successors via control flow edges (lines 12-13, 15-18) since we cannot skip blocks that are in between. The current blocks have to have the same number of control flow edges going into them (line 14), otherwise the subgraphs cannot be

[1]Node flag which indicates that the node always outputs the same value when it has the same attributes, therefore these nodes are part of the ir_graph only once

```
function init_isomorphic_subgraph_pair_and_expand:
    foreach node in all_ir_graphs_nodes:
        if exists entry in similar_nodes_list
              where is_similar_node(node, entry.first_node):
            foreach similar_node in entry.nodes:
                isp = init_new_isp(node, similar_node)
                maximum_expand_isp(isp)
                if pattern_size(isp) >= 8 and !isp.deleted:
                    isp_array.append(isp)
            entry.nodes.add(node)
        else:
            add_new_entry_to_similar_nodes_list(node)
```

a init a isomorphic subgraph pair structure and calling expand

```
function maximum_expand_isp(isomorphic_subgraph_pair isp):
    while isp.change and !isp.deleted:
        foreach (node1, node2) in
                isp.outer_nodes_graph1 zipWith isp.outer_nodes_graph2:
            foreach (pred_node1, pred_node2) in
                    node1.pred_nodes zipWith node2.pred_nodes:
                if are_nodes_exclusive(pred_node1, pred_node2, isp)
                        and is_similar_node(pred_node1, pred_node2):
                    if in_subgraph(pred_node1, isp.graph1)
                            xor in_subgraph(pred_node2, isp.graph2):
                        isp.deleted = true
                    if node1.block != pred_node1.block
                            and node2.block != pred_node2.block
                            and is_similar_block(node1.block, node2.block)
                            and exists x in node1.block.cfg_preds.block
                                where x == pred_node1.block
                            and exists y in node2.block.cfg_preds.block
                                where y == pred_node2.block:
                        add_blocks(pred_node1.block, pred_node2.block, isp)
                        add_phis(node1.block, node2.block, isp)
                        add_nodes(pred_node1, pred_node2, isp)
                    else if node1.block == pred_node1.block
                            and node2.block == pred_node2.block:
                        add_nodes(pred_node1, pred_node2, isp)
```

b expanding a similar subgraphs structure

Figure 3.2.: Pseudo code for finding ISPs

expanded to the new blocks. If the blocks have more than one control flow input, all Phi nodes have to be added to the subgraphs as well since the outlined function is not able to indicate which control flow is chosen to the caller and therefore the values have to be chosen inside the function.

3.1.2. Grouping ISPs together

After all isomorphic subgraph pairs are expanded to their maximum size, the next step of the outline optimization tries to group ISPs together into ISGs. These data structures contain all isomorphic subgraphs in an array (the pattern) as well as arrays of the pattern arguments and return values, which are calculated later. Figure 3.5 outlines how the grouping is implemented.

First, the ISPs are sorted by their number of nodes in descending order (line 2).

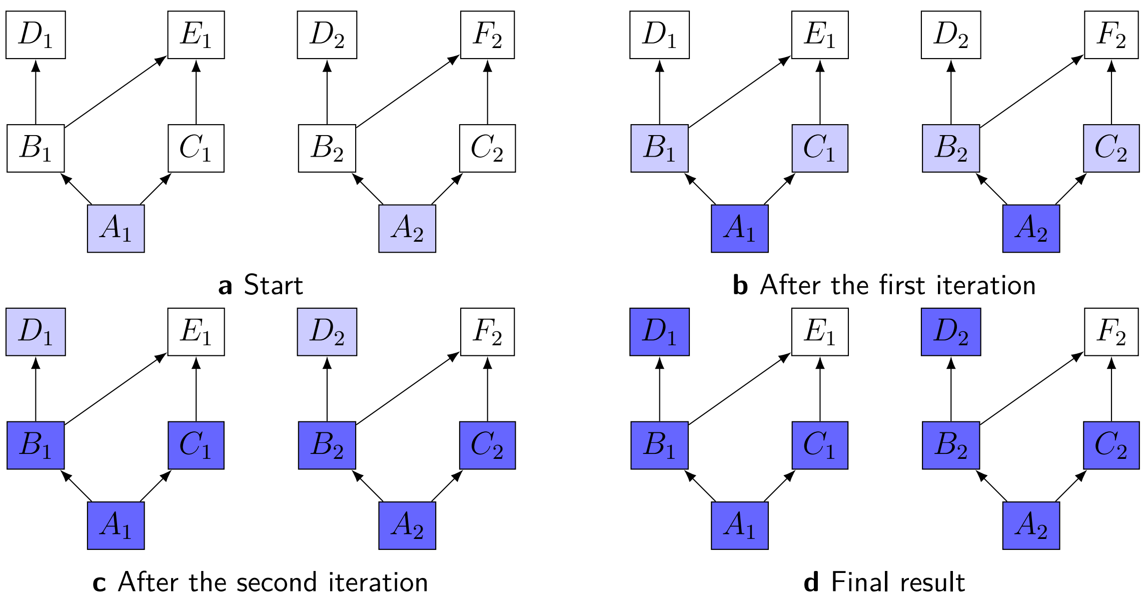
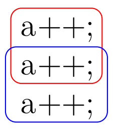


Figure 3.3.: Schematic example of the expansion algorithm, light blue nodes are the outer nodes of the ISP



```
a++;
a++;
a++;
```

Figure 3.4.: Example why nodes have to be exclusive to a pattern

Then, the optimization iterates over the sorted array (line 3). If the current ISP is not marked as deleted and all checks are successful (lines 5-6), a new ISG is created and the two subgraphs are copied to the new data structure (line 7). Next, the optimization tries to match other ISPs to the ISG. Since the ISPs are sorted by their pattern size, the algorithm tries to match the array elements starting at index plus one up until the pattern size changes (lines 8-9). It is sufficient to only look at those ISPs in this range in order to make sure that if there is a ISP that matches the ISG, it will be matched and added to the ISG. We take advantage of lazy evaluation and only check whether the pattern is the same (line 15), which compares the entries of the pointer sets, when one exit node is the same as in the ISG (lines 13-14). Furthermore, we already run a few checks on the pattern occurrence (line 16) which will be discussed later. If the match is successful, the pattern not yet contained in the ISG will be added to it and the ISP will be marked as deleted (lines 17-18).

```
function group_isps_to_isgs:
    sort_isp_array()
    for int i = 0; i < isp_array.length; i++:
        isp = isp_array[i]
        if !isp.deleted:
            if run_checks_on_isp(isp):
                isg = init_new_isg_with_isp(isp)
                match_i = i + 1
                while match_i < isp_array.length and
                      pattern_size(isp) == pattern_size(isp_array[match_i]):
                    isp_to_match = isp_array[match_i]
                    if !isp_to_match.deleted:
                        if (isp.exit_graph1 == isp_to_match.exit_graph1 or
                                isp.exit_graph1 == isp_to_match.exit_graph2) and
                                is_same_pattern(isp, isp_to_match) and
                                run_checks_on_isp(isp_to_match):
                            add_isp_to_isg(isg, isp_to_match)
                            isp_to_match.deleted = true
                    match_i++
            isp.deleted = true
            isg_array.append(isg)
```

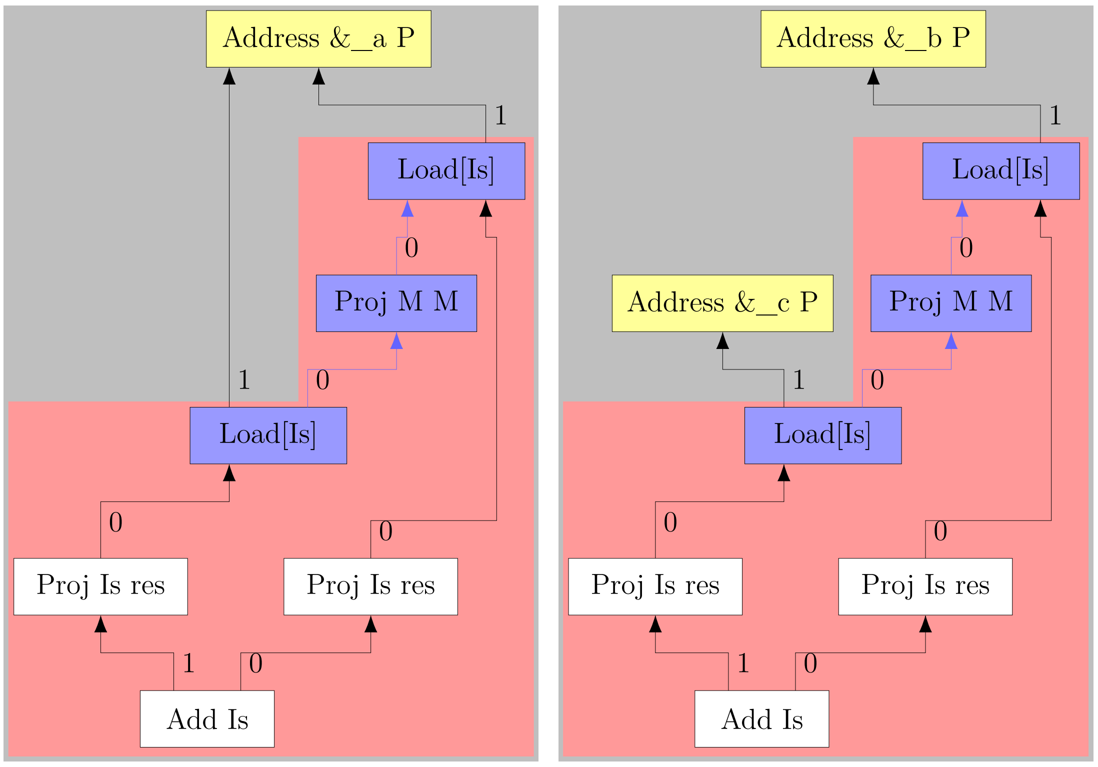
Figure 3.5.: algorithm for grouping ISPs into ISGs

3.2. Calculating the arguments

A pattern has pattern arguments and return values. Pattern arguments are values outside of the pattern that are needed inside of it, return values are values inside the pattern which are needed outside of it. They are calculated in two separate steps and saved into the corresponding arrays for each pattern.

3.2.1. Pattern arguments

Nodes inside of the pattern can depend on values which are outside of the pattern. Consider an ir_node n which is part of the pattern. n depends on a value v that is not part of the pattern. We call v a *pattern argument* and n an *argument dependent*. In order to find all pattern arguments, it is sufficient to only iterate over all nodes in the pattern and to look for data dependency edges where the destination is not part of the pattern. Checking for pattern arguments is done for every occurrence of the pattern since it is possible that in one pattern occurrence two different argument dependents point to the same pattern argument whereas in the other occurrence the argument dependents point to two different pattern arguments as shown in Figure 3.6. However, if two or more different argument dependents in the pattern point to the same pattern arguments in every occurrence, the pattern arguments are combined and the node's value is only passed once to the outlined function. Primitive types like integers and floats are passed as values, more complex types like arrays and structs are passed to the outlined function as pointers.



a Same value twice **b** Two different values

Figure 3.6.: Two patterns that are similar, but Figure 3.6a uses variable a twice whereas Figure 3.6b uses variable b and c

3.2.2. Return values

Like pattern arguments, there can also be nodes outside of the pattern which depend on values inside the pattern. Consider an ir_node n which is not part of the pattern, but depends on the value v which is part of the pattern. We call v a *return value* and n a *return value dependent*. In order to find all return values, the outline optimization checks all data dependencies in the ir_graph. If the optimization finds a data dependency that goes from outside the pattern into it, the return value node will be added to the array for each pattern occurrence. Every occurrence has to be checked for the data dependencies since for one occurrence there might be a return value dependent whereas for another occurrence there might not be.

libFIRM can only return a limited number of values with the built-in Return node depending on the back end. For example, the x86 back end has only two registers for returning values. Therefore, if the function has more return values than supported by the back end, another approach has to be used. In the ir_graph where the pattern occurs, memory is allocated for every additional return value. We call the allocated chunk of memory the *return value store*. The pointers to the return value stores are then passed as arguments to the outlined function. The outlined function saves the values from the return value nodes to the given addresses. After the call to the outlined function, these values are loaded from the return value store and the data dependencies going into the pattern are rerouted to the corresponding Proj nodes, which output the loaded return values.

3.3. Checking the patterns

Before a pattern can be outlined, it must be checked whether outlining is possible or not. Three criteria have to be met:

- The pattern is a *single entry single exit* (SESE) region [13] meaning that the pattern has exactly one entry and exactly one exit for the control flow since it is not possible to pass control flow to a function nor return it.
- There is no path out and back into the pattern since a pattern argument would depend on a return value then.
- The memory chain is not separated since it is not possible to return nor inject a memory state mid-function.

If all of the checks are successful, the pattern is eligible for outlining and replacement with calls to the function.

3.3.1. SESE region

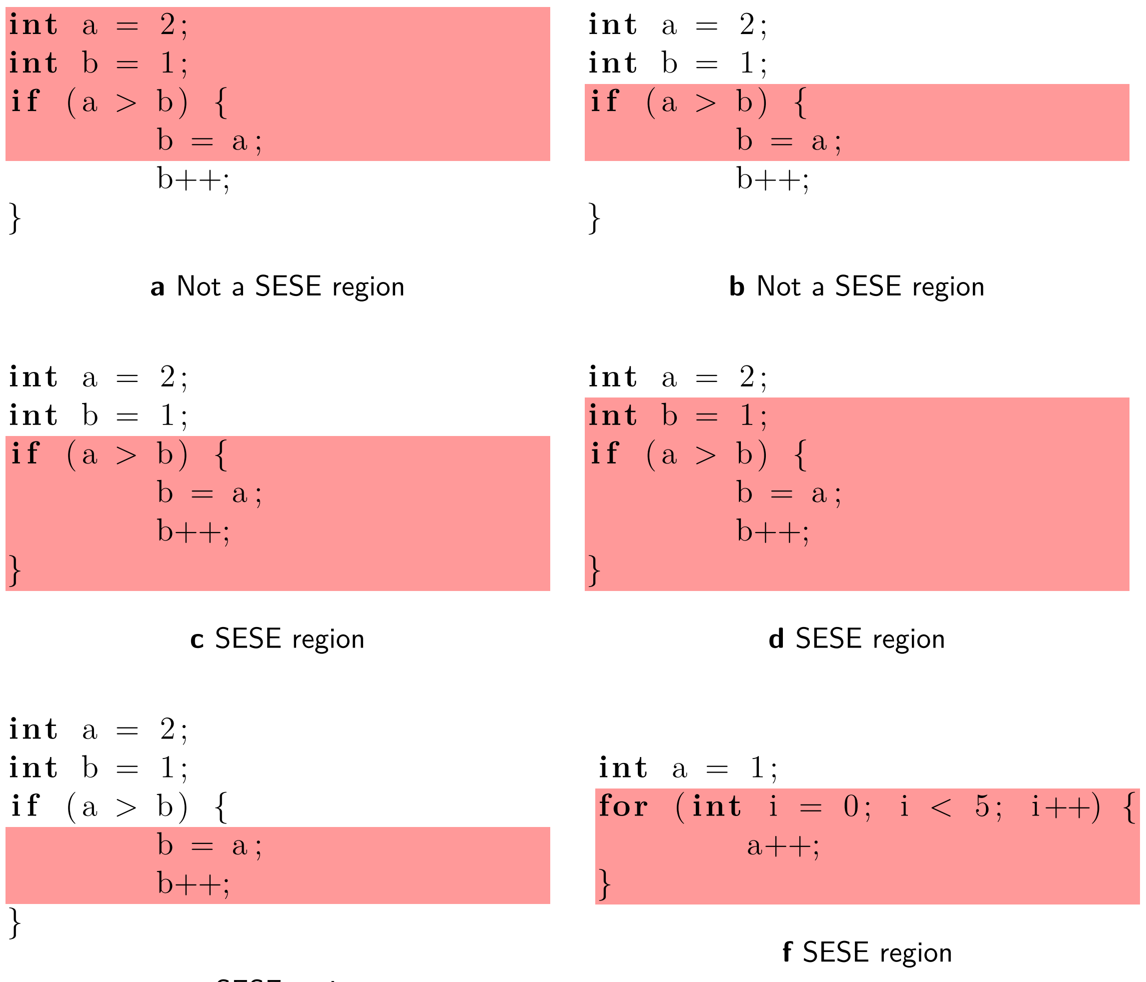
In order for a pattern to be considered for outlining, it has to be a SESE region. To understand what a SESE region is, a few simple C examples are given in Figure 3.7. The regions highlighted in Figure 3.7a and Figure 3.7b are not SESE regions in the control flow graph since they do not include the whole if body. Figure 3.7c takes the if clause and the body, Figure 3.7d additionally takes a term before the if clause, and Figure 3.7e only takes the if body making them a SESE region. Figure 3.7f shows that loops can also be SESE regions.

According to Johnson, Pearson, and Pingali [13] the following criteria have to be met for the entry edge a and exit edge b for a region to be a SESE region:

1. a dominates b
2. b postdominates a
3. every cycle containing a also contains b and vice versa

Dominance and postdominance is defined as follows:

- an edge x dominates the edge y if every path from the start node to y includes x
- an edge x postdominates the edge y if every path from y to the end node includes x



```
int a = 2;
int b = 1;
if (a > b) {
        b = a;
        b++;
}
```

a Not a SESE region

```
int a = 2;
int b = 1;
if (a > b) {
        b = a;
        b++;
}
```

b Not a SESE region

```
int a = 2;
int b = 1;
if (a > b) {
        b = a;
        b++;
}
```

c SESE region

```
int a = 2;
int b = 1;
if (a > b) {
        b = a;
        b++;
}
```

d SESE region

```
int a = 2;
int b = 1;
if (a > b) {
        b = a;
        b++;
}
```

e SESE region

```
int a = 1;
for (int i = 0; i < 5; i++) {
        a++;
}
```

f SESE region

Figure 3.7.: Four examples of a SESE region and two examples that are not a SESE region

The first condition makes sure that every path that enters the pattern does so via the entry edge. The second condition makes sure that every path that exits the pattern exits it via the exit edge. The third condition is needed for eliminating patterns where backedges either enter or exit the pattern.

Since libFIRM calculates the dominance informations using blocks, the entry and exit block are used instead of the edges. Johnson et al.'s criteria are not sufficient for our use case. In addition to the entry and exit, we have to consider the blocks in between as well. The outline optimization adds two further constraints to the SESE definition:

- all blocks between the entry and exit block (*inner blocks*) have to be part of the pattern
- all nodes belonging to an inner block are part of the pattern

3.3.2. Pattern exit and reentry

We will use an example to explain the problem which arises from a pattern exit and reentry. In Figure 3.8, the nodes `Address a`, `Address b`, and `Mul` are arguments of the highlighted pattern. The value loaded by the `Load` node is a return value. Since the `Mul` node depends on the return value, outlining this pattern would result in the Call node depending on a value that is calculated by the called function. Thus, the pattern is not eligible for outlining.

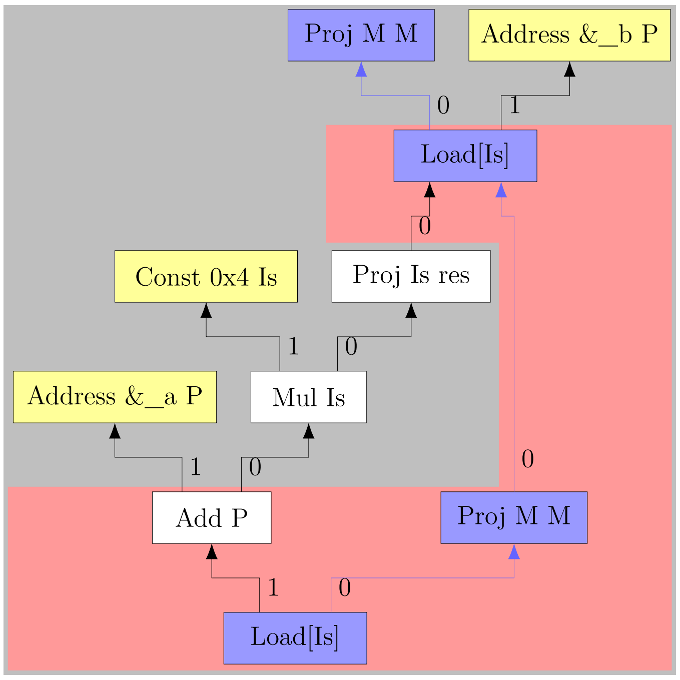


Figure 3.8.: Pattern exit and reentry

The outline optimization checks for every argument if there is a path to the start node that enters the pattern. If such path exists, the argument will have a dependency on a return value and the pattern will therefore be excluded from outlining.

3.3.3. Memory chain

Since a memory state cannot be returned nor injected mid-function, the memory nodes inside a pattern have to form a complete chain without interruption. This is similar to the pattern exit and reentry, however we check for an uninterrupted chain differently. First, the number of memory nodes in the pattern is counted. Then, starting from the last memory node, the check walks the memory chain until a node is not part of the pattern anymore counting the memory nodes throughout. If the two counts match, the memory chain inside the pattern is uninterrupted.

3.4. Extracting the patterns

If all checks on the pattern are successful, we will copy the pattern to a new function first and then replace all occurrences of the pattern with a call to the new function.

3.4.1. Creating a new function

First, a new ir_graph representing the new function has to be created. A newly created method already has some basic nodes like the Start and End node. Then, the outline optimization creates the Proj nodes for the arguments as seen in Figure 3.9.

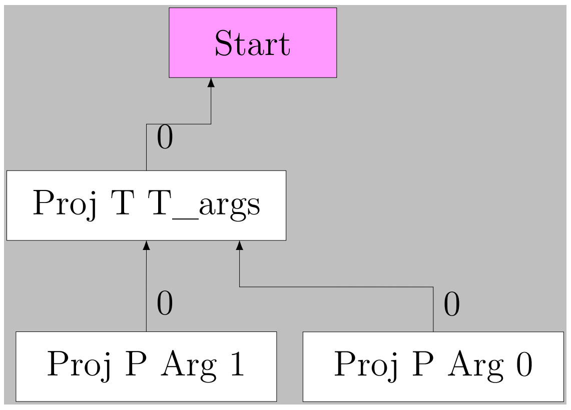


Figure 3.9.: Proj nodes to a function's arguments

One pattern occurrence is picked for copying the nodes which has to be done in two steps. First, the optimization iterates over all nodes in the pattern and creates a copy in the new function. A link from every original node to the new node is set. The edges of the new method's nodes are set using the edges of the original pattern and the link to the new nodes in the second iteration.

Next, the argument dependents' edges which point to a pattern argument are set to the corresponding Proj nodes.

Then, if the start block[2] is not part of the pattern, the control flow will be fixed since the copied blocks are not connected to the new function's start block. A Jmp node to the entry block is added to the start block. The cfopt optimization will merge the blocks together later.

Lastly, the last memory node in the pattern has to be found. It is either directly attached to the Return node or used for the storing of return values. Every return value of a pattern as described in Section 3.2.2 has to be either returned by the built-in Return node or stored at the passed address at the end of the new function. An example for storing one return value is given in Figure 3.10. In this example, there are more return values than the back end can handle. One value can be returned by the built-in Return node, the other has to be stored in the return value store.

[2]block containing the Start node

The second argument of the outlined function is the address where the return value should be stored, thus the `Store` node having the `Proj 1` as the pointer input. The `Return Value 1` node holds the value which is needed outside of the pattern and is therefore the value input for the `Store` node.

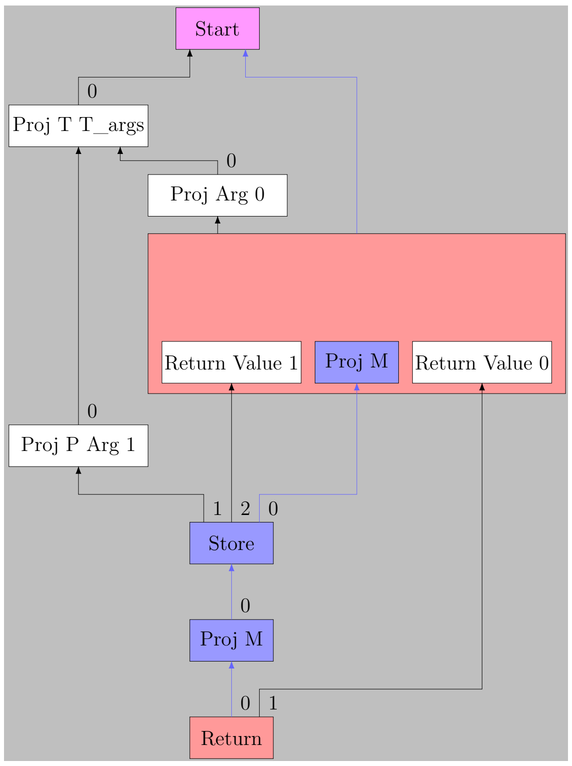


Figure 3.10.: Storing a return value to the passed address, pattern represented by red box

3.4.2. Replacing the pattern with call to function

After a new function for the pattern is created, the pattern can be replaced by a call to the function at every occurrence. First, a new entity is created for every additional return value over the limit given by the back end, so that the address of the entity can be passed to the function. Then, the outline optimization looks for the memory input for the call. There are two possibilities: either the pattern contains memory nodes or it does not. If the pattern contains memory nodes, the memory node that has an edge going out of the pattern is chosen as the memory input for the call node. It is the first memory node in the pattern. If the pattern does not contain memory nodes, we still have to pass a memory state to the function since storing a return value can add memory nodes. The last memory node in the entry block that is not reachable from any argument is chosen as the memory input since otherwise arguments that depend on a value loaded from memory would cause a memory loop.

We need to fix control flow as well. A Jmp node to the exit block is added to the entry block.

Finally, the return values have to be connected to the nodes that depend on them. Return values which are returned by the Return node are extracted by Proj nodes. A Load node is placed after the call for every additional return value and the edges that exit the pattern are attached to the loaded values.

Figure 3.11 shows a call to an outlined function with the parameters pointer to variable *a* and pointer to variable *b*. A pointer to where the return value should be stored is passed as well and the value is loaded directly after the call. To simplify matters, we do not use the built-in Return node in this example and therefore no return value is extracted from the output of the call node.

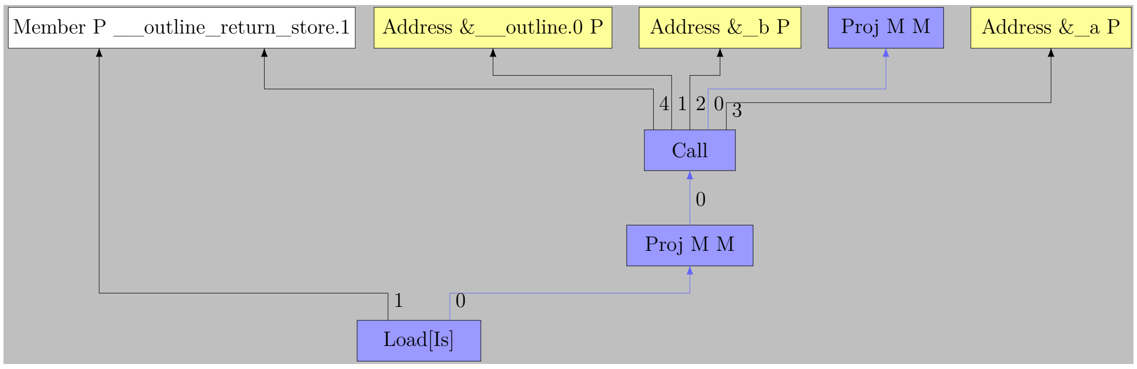


Figure 3.11.: Call to the outlined function, built-in Return node is not used

3.5. Performance tuning

Since the brute-force algorithm used in the outline optimization compares every node with all other nodes and only creates pairs to begin with, the worst-case time and space complexity is $\mathcal{O}(n!)$. Therefore, we added certain mechanisms so that the optimization is still feasible on some larger graphs.

3.5.1. Leaving out certain nodes

The time and the memory consumption can be reduced by skipping certain nodes and not considering them as the start for an ISP. For example, it is not necessary for Proj nodes to be the start of a pattern, because if the predecessors would form a isomorphic subgraph pair, the Proj node would be a return value anyway. Therefore, it is sufficient to just check the predecessor, saving one subgraph expansion and the memory that would otherwise be used by the ISP.

3.5.2. Aborting when limit is reached

Since it depends on the source code how long the algorithm takes and it cannot be predicted beforehand, the user has the option to specify two limits after which the outline optimization will stop if one of them is exceeded. The first limit is a timeout after which the algorithm stops looking for further isomorphic subgraph pairs. The second limit is a maximum number of ISPs to look for. Both limits have in common that after the limits are exceeded, the optimization still has to group the subgraphs together, run the checks on the pattern, and replace the patterns with a function, thus still taking more time and allocating more memory.

4. Evaluation

In order to evaluate our outlining implementation, we use two test suites. We use libFIRM's own set of test files, the libFIRM C test suite [14], as well as the SPEC CPU2000 benchmark [15]. Testing is conducted on a 3.40GHz Intel® Core™ i7-6700 CPU with 32 GB of system memory running Ubuntu 16.04.5. We used the x86 back end because it is considered stable.

4.1. libFirm's C test suite

libFIRM's test suite consists of 1783 test files, which test various aspects of the compiler. We extended the test runner to accept an additional set of compiler flags in order to compare two binaries to each other. Table 4.1 gives an overview of the data collected. Unfortunately, there are hardly any test cases where the outline

# of test cases		1783
	that fail	106
	that fail with outlining	109
	where pattern exist	65
# of patterns	Ø	1.431
	max	9
pattern size	Ø	17.817
	max	375
# of pattern occurrences	Ø	4.226
	max	24
# of pattern arguments	Ø	6.075
	max	128
# of return values	Ø	0.892
	max	3
size difference of the text sections in bytes	Ø	121.19
	max	832
	min	−16

Table 4.1.: Results of the outline optimization being applied to the libFIRM test suite

optimization reduces the size of the text section[1]. Only one test case has 16 bytes less than the reference text section, two test cases have the exact same size. We want to point out the maximal size difference we observed where the outline optimization adds over 50% in size to the text section. This test case[2] is the one that has one pattern with 128 pattern arguments. We therefore conclude that pattern arguments have a huge overhead.

4.2. SPEC CPU2000

We use all C benchmarks in the SPEC CPU2000 benchmark, which means we use eleven integer benchmarks and three floating point benchmarks. We are able to run all benchmarks without a timeout except for the *perlbmk* benchmark which ran with a 30 minute timeout for the outline optimization.

Table 4.2 gives an overview of the data collected applying the outline optimization to the SPEC CPU2000 benchmark. A more detailed overview is given in Table A.1 and Table A.2 in the Appendix. The outline optimization is able to find some patterns in all benchmarks except *art* and *mcf*. The results show that the outline optimization has a huge impact on the compile times and that it is not feasible to run it during development in real world scenarios. Each benchmark was run twenty times. Most benchmark run times do not differ in a significant way, except for *vpr*, *mesa*, *parser*, *perlbmk*, and *vortex*. These benchmarks do not only add an overhead in the text section size, but a measurable overhead in running time as well. The benchmarks that have the most patterns have the biggest difference in text section size.

4.3. Overhead analysis

We want to take a closer look at the overhead a function has. Therefore, we manually generate test files where we once have duplicates in the source file and once we re-factor these duplicates into a separate function.

Figure 4.1 shows one test case that we tried which has a smaller text section when the minimum function is in-line. It is worth noting that the outline optimization does not find any patterns in this example because when transformed into an ir_graph, not all SESE requirements are met, however it is sufficient for showing the overhead that comes along with a function call.

We take another, simple test file consisting of a hundred *asm("nop")* calls. We increase the minimum pattern size to 14 and run the outline optimization. The outlined pattern has a size of 21 with ten occurrences and zero pattern arguments as well as zero return values. The text size section is 16 bytes less when the outline optimization is applied to this test case. However, with the increased pattern size,

[1]part of the binary where the instructions are located

[2]opt/fehler216.c, initialization of an array with 250 elements with the same value

# of patterns	Ø	20.769
	max	66
pattern size	Ø	12.54
	max	75
# of pattern occurrences	Ø	3.646
	max	32
# of pattern arguments	Ø	4.16
	max	31
# of return values	Ø	1.426
	max	10
compile time Δ in %	Ø	955.50
	max	3253.93
	min	100.00
run time Δ in %	Ø	2.16
	max	18.07
	min	−0.83
size difference of the text sections in bytes	Ø	2509.538
	max	7728
	min	112

Table 4.2.: Overview of the results of the outline optimization being applied to the SPEC CPU2000 benchmark

the libFIRM's C test suite average difference of the text sections increases to 267.52 bytes, with only one test case reducing the size of the text section by −208 bytes.

We compare the assembler outputs of a test file with and without outlining in order to find the number of bytes added to the text section. The result is presented in Table 4.3. We cannot predict the exact amount of bytes a certain structures will need since it depends on factors which the middle end does not know. The values could be used to approximate the size overhead and to improve the pattern choosing, however, they only apply to the x86 back end. Furthermore, it is not possible to know how many nodes account for one instruction (and therefore the instruction size), making it impossible to define an exact threshold for outlining.

```
int main() {
  int min_ab;
  int min_ac;

  if (a < b)
    min_ab = a;
  else
    min_ab = b;

  if (a < c)
    min_ac = a;
  else
    min_ac = c;

  return min_ab + min_ac;
}
```

a text section has 402 bytes

```
int min(int a1, int b1) {
  if (a1 < b1)
    return a1;
  else
    return b1;
}

int main() {
  return min(a,b) + min(a,c);
}
```

b text section has 450 bytes

Figure 4.1.: First manual test

	caller	callee
call / function overhead	5 bytes	10 bytes
pattern argument	at least 3 bytes	3 bytes
return value with built-in Return node	0 or 3 bytes	3 bytes
return value with return value store	at least 6 bytes	5 bytes

Table 4.3.: Overhead of outlining

5. Conclusion

Our implementation of the outline optimization shows that it is possible to identify and automatically extract patterns in an IR in theory. However, the evaluation of the outlining optimization reveals that the optimizations falls short of the anticipated behavior. It hardly decreases the size of the binary while increasing compile time by up to 33 times.

There is still room for improvement though. Better heuristics could lead to better pattern selection, resulting in more test cases with reduced binary size. Furthermore, we only consider the maximal pattern, so reducing a pattern and only testing a subpattern might lead to better candidates. Finally, using a better FSM algorithm would probably improve the compile time significantly.

We assume that finding duplicates using PDGs and then either displaying or outlining them at source code level is the preferred option. It eliminates duplicates and makes the source code more error proof.

However, the pattern finding algorithm at compiler level could be used for finding small procedures that might be interesting for creating a new instruction at processor level. Designing and implementing an optimized instruction for a pattern would not only decrease the binary size, but furthermore might be able to improve run time.

Bibliography

[1] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, pp. 176–192, March 2006.

[2] P. P. Chang and W.-W. Hwu, "Inline function expansion for compiling c programs," *SIGPLAN Not.*, vol. 24, pp. 246–257, June 1989.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, Oct. 1991.

[4] "libFIRM Main Page." `https://libfirm.github.io`. Last checked: 2018-03-14.

[5] M. Braun, S. Buchwald, and A. Zwinkau, "Firm — a graph-based intermediate representation," Tech. Rep. 35, Karlsruhe Institute of Technology, 2011.

[6] "cparser GitHub repository." `https://github.com/libfirm/cparser`. Last checked: 2018-03-14.

[7] C. Jiang, F. Coenen, and M. Zito, "A survey of frequent subgraph mining algorithms.," *Knowledge Eng. Review*, vol. 28, no. 1, pp. 75–105, 2013.

[8] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, pp. 31–42, Jan. 1976.

[9] L. P. Cordella, P. Foggia, C. Sansone, F. Tortorella, and M. Vento, "Graph matching: a fast algorithm and its evaluation," in *Proceedings. Fourteenth International Conference on Pattern Recognition (Cat. No.98EX170)*, vol. 2, pp. 1582–1584 vol.2, Aug 1998.

[10] X. Yan and J. Han, "gspan: graph-based substructure pattern mining," in *2002 IEEE International Conference on Data Mining, 2002. Proceedings.*, pp. 721–724, 2002.

[11] E. Gudes, S. E. Shimony, and N. Vanetik, "Discovering frequent graph patterns using disjoint paths," *IEEE Trans. on Knowl. and Data Eng.*, vol. 18, pp. 1441–1456, Nov. 2006.

[12] R. Komondoor and S. Horwitz, “Using slicing to identify duplication in source code,” in *Proceedings of the 8th International Symposium on Static Analysis*, SAS ’01, (London, UK, UK), pp. 40–56, Springer-Verlag, 2001.

[13] R. Johnson, D. Pearson, and K. Pingali, “The program structure tree: Computing control regions in linear time,” *SIGPLAN Not.*, vol. 29, pp. 171–185, June 1994.

[14] “libFIRM’s C Testsuite GitHub repository.” `https://github.com/libfirm/firm-testsuite`. Last checked: 2018-03-14.

[15] “SPEC CPU2000 Main Page.” `https://spec.org/cpu2000`. Last checked: 2018-03-14.

Erklärung

Hiermit erkläre ich, Petar Heyken, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift

A. Appendix

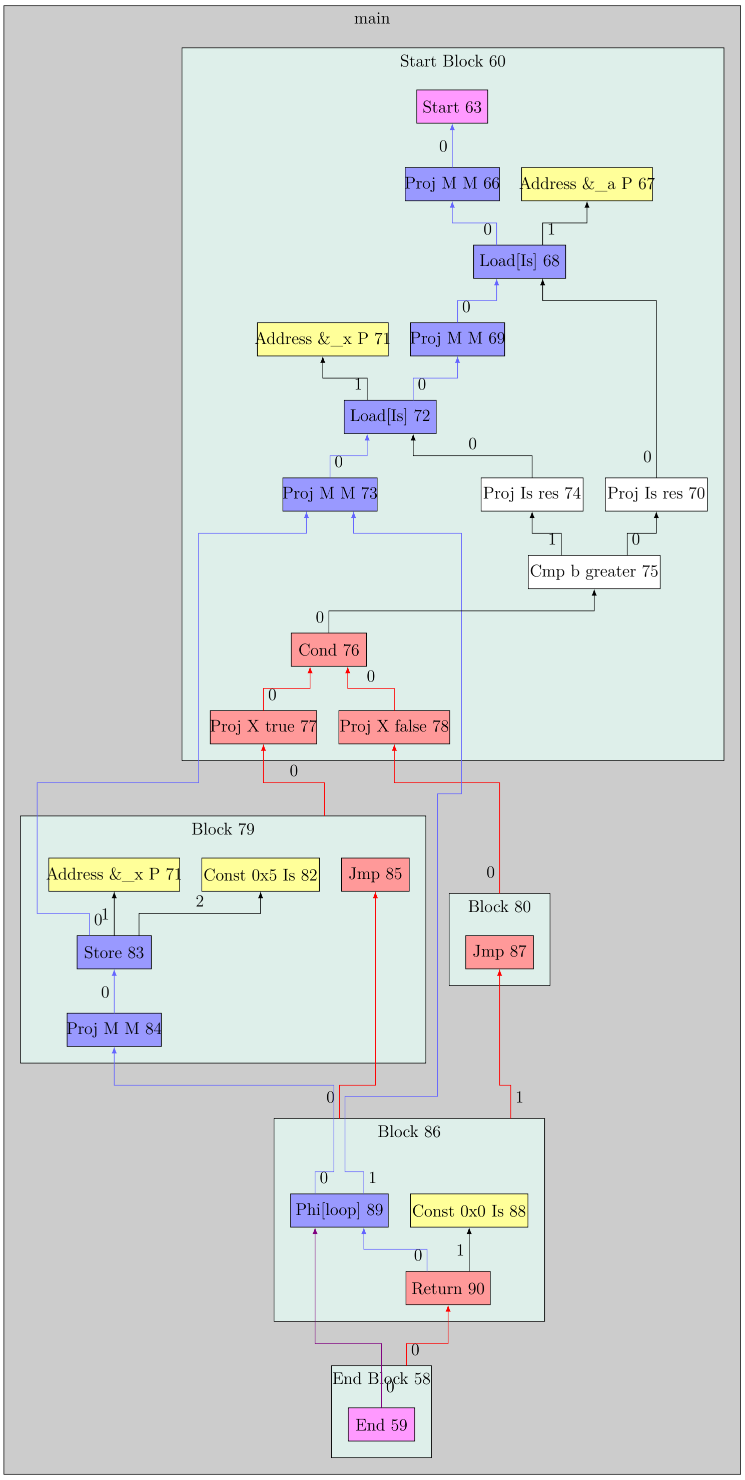


Figure A.1.: ir_graph from Figure 2.3b

		pat. size		pat. occurrences		pat. arguments		return values	
	# of pat.[1]	Ø	*max*	Ø	*max*	Ø	*max*	Ø	*max*
gzip	2	11	14	3.5	4	2.5	3	0.5	1
vpr	12	9.417	13	2.333	5	2.917	4	1.833	4
gcc	57	14.404	71	3.754	29	4.14	31	1.421	4
mesa	66	11.985	32	3.106	16	5.091	17	1.258	4
art	0								
mcf	0								
equake	3	12	16	5	10	5.667	11	1	2
crafty	3	12	15	2	2	6	10	1.333	3
ammp	4	14.25	15	3.25	5	3	4	2.25	3
parser	4	9.75	10	3.5	7	3.25	6	1	2
perlbmk[2]	45	13.756	75	4.511	32	3.044	23	2.244	10
gap	14	11.571	35	5.714	21	4.929	10	0.786	3
vortex	49	11.837	30	3.265	21	4.102	13	1.082	4
bzip2	1	15	15	3	3	8	8	0	0
twolf	10	9.875	15	3.375	6	3.375	5	1.5	3

[1] pattern

[2] run with a timeout of 30 minutes

Table A.1.: The different SPEC CPU2000 benchmarks with the statistics of the outline optimization

			run time							
	compile time		Ø in seconds			σ_M		text section size in bytes		
	outline	reference	outline	reference	Δ	outline	reference	outline	reference	Δ
gzip	40s	10s	57.608	58.088	-0.83%	0.063	0.083	43426	43298	128
vpr	3m11s	29s	44.536	43.519	2.34%	0.046	0.021	154738	153746	992
gcc	2h44min8s	6min42s	18.904	18.909	-0.02%	0.005	0.005	1820275	1813507	6768
mesa	21min54s	1min57s	41.058	40.082	2.44%	0.014	0.025	696914	689186	7728
art	16s	3s								
mcf	4s	2s								
equake	25s	3s	20.881	20.947	-0.32%	0.031	0.028	17602	17138	464
crafty	3m54s	55s	23.815	23.555	1.11%	0.007	0.016	183570	182754	816
ammp	1m53s	22s	78.786	78.809	-0.03%	0.178	0.115	126866	126114	752
parser	7m09s	37s	56.059	54.864	2.18%	0.043	0.035	183266	182738	528
perlbmk	1h39m30s	2m58s	45.138	43.977	2.64%	0.015	0.021	856355	849699	6656
gap	30m27s	2m12s	22.939	22.894	0.19%	0.020	0.024	561138	558354	2784
vortex	13m10s	1m38s	39.991	33.872	18.07%	0.020	0.059	742242	738178	4064
bzip2	2m16s	8s	43.417	43.310	0.25%	0.038	0.037	47170	47058	112
twolf	2m55s	1m0s	60.734	60.654	0.13%	0.115	0.066	189330	188498	832

Table A.2.: The different SPEC CPU2000 benchmarks with their compile times, run times, and the text section sizes