HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

INSTITUTE FOR
NETWORKED SOLUTIONS

Automatic Refactoring for Parallelization

Master's Thesis

Department of Computer Science
University of Applied Sciences Rapperswil

Spring Term 2017

Author:	Christoph Amrein
Advisor:	Prof. Dr. Luc Bläser, HSR
Project Partner:	Institute for Networked Solutions
External Examiner:	Dr. Felix Friedrich, ETH Zürich

Abstract

In modern software, adaptions like the parallelization are necessary to fully leverage the CPU's capabilities. However, the parallelization introduces a new range of possible software faults. Thus, assisting utilities like static code analyzers are desirable. For instance, ones that inform engineers about the code fragments that can be safely adapted. This thesis focuses on loops and introduces a conservative approach to verify that these can be parallelized. More specifically, it allows proofing that array accesses do not conflict between iterations. The procedure is a data flow analysis, which proofs the absence of conflicts by employing rules for a selection of binary expressions. Furthermore, its design allows it to profit from various code optimizations automatically. Experimental evaluations show that both, the prototype and the data flow analysis itself, do not incorrectly identify loops as parallelizable. Moreover, it pinpoints that the analysis can correctly identify most of the parallelizable loops, and only a negligible amount requires a more mature approach.

1 Introduction

When it comes to improving the performance of an application, one possible enhancement is the parallelization of computationally intensive code fragments. Loops primarily used for badge processing data of an array are excellent candidates for parallelization. These loops shelter the opportunity that each iteration can be executed concurrently which possibly leads to extraordinary speed-up when doing computationally intensive work. However, the parallel code adds a new range of possible errors such as race conditions [2] during program execution.

Therefore, the transformation of a loop into its parallel equivalent requires certain checks, inter alia, the absence of loop carried dependencies. A loop carried dependence is present if a statement of an iteration depends on a statement of another [2]. For example, if there are two or more iterations that access the same array element where at least one is writing. Consider the loop sketched in Listing 1. This loop calculates the sum of all entries up to the current entry and stores the result in the current entry. An array access intersection is present due

```
1  int[] a = new int[] {
2    1, 2, 3, 4, 5, 6
3  };
4  int m = a.Length;
5  for(int i = 1; i < m; i++) {
6    int p = i - 1;
7    a[i] = a[p] + a[i];
8  }
```

Listing 1: Loop that sums-up the array entries with access intersections

to the read and write accesses to the different array indices `i` and `p` on line 7. However, if only one of these two indices was used, there would be no longer an array access intersection. This example underpins the need for a technique to verify that — possibly computed — array indices do not intersect between iterations.

There already exist various approaches to identify whether an access to an array conflicts with other iterations or not, each having its advantages. A mere candidate is the GCD dependence test [6, 37, 45]. This approach allows deciding if array indices computed with linear expressions may intersect with other iterations [37].

A more powerful approach is the polyhedral model (or polytope model) [1, 11]. It allows the transformation of loops which — in their original form — are not parallelizable but are in a restructured way. The detection of intersections is accomplished by the setup of inequalities and solving for an integer solution with the help of an ILP solver. However, as integer linear programming is NP-hard [15], the computation of integer solutions may get computationally intensive.

This thesis introduces a conservative data flow analysis approach. This approach thereby makes use of transfer rules to proof that an index will not intersect between iterations. These rules are conservative in the way they transfer information. More specifically, they only transfer information if it is guaranteed for any situation. With the help of the analysis results, it is then possible to proof the absence of array access intersections. For the sake of simplicity, this thesis concentrates on for loops only. Thus, the complexity required to identify the loop index (or loop counter [3]) can be neglected. In

for loops, the loop index is the variable controlling the iterations of the loop. In the example sketched in Listing 1, the loop index is the variable `i`. The variable `p`, on the other hand, is only a variable that depends on the loop index.

Throughout the following pages, Section 2 introduces the conservative analysis by applying it to a given example code. Section 3 evaluates the usefulness of the introduced analysis. Moreover, it gives a brief overview of the features of the implemented prototype. In Section 4, a survey of related work that support the development of parallel code is given. Ultimately, the results of running the prototype on a collection of open source projects are summarized. Last but not least, Section 5 gives a short reflection and provides an outlook on possible extensions.

```
1  long[] f = new long[] {
2    15, 15, 18, 18, 20, 20
3  };
4  int m = f.Length / 2;
5  for(int i = 0; i < m; i++) {
6    int o = 1;
7    int p = o + i * 2;
8    long s = f[p];
9    long c = 1;
10
11   for(int j = 1; j <= s; j++) {
12     c = c * j;
13   }
14
15   f[p] = c;
16 }
```

Listing 2: Loop which computes the factorial of every odd array index

2 Analysis

This section introduces an approach which reasons the absence of array access intersections by the application of simple transfer rules in the data flow analysis. On the one hand, the design of this approach makes it independent from any other optimization or analysis while on the other hand it still profits from them. In other words, the analysis is correct in any situation while various optimizations can improve the results.

In the upcoming sections, Section 2.1 introduces an example which is used throughout the sections to explain the different steps of the analysis, and Section 2.2 proposes the notation used to encode the information in the analysis. The rules used to derive information from expressions are summarized in Section 2.3, and Section 2.4 describes how the data flow analysis is structured. Conclusively, Section 2.5 describes how the analysis could be adapted for the use with SSA and how the results could be improved further.

2.1 Example

To aid the comprehension of the complete analysis, the example code illustrated in Listing 2 is processed throughout the sections. It computes the factorial of every odd array element. Figure 1 sketches the control flow graph of the outer loop's body. Each block is numbered with a unique label, thus it is label consistent [38].

The illustrated code consists of two nested for loops. The outer declared on line 5 iterates over the array entries, and the inner on line 11 computes the factorial of the entry. As there are neither write accesses to shared memory nor intersecting accesses to the array `f`, the outer loop is parallelizable. The inner loop is not parallelizable because it would have read and write accesses to the same local variable `c` if the loop was parallelized.

The loop index of the outer loop is the variable `i`, and the variable `j` is the loop index of the inner loop.

2.2 Notation

This section introduces a small set of notations. These notations express the different kinds of information encoded within the analysis to simplify the rule application examples.

The analysis employs an embedded reaching definitions analysis [28, 38] to identify possible conflicts of variables. The reaching definitions information is expressed as $x@\ell$ stating the block with the label ℓ defines the variable x. This embedded analysis also removes the need for an intermediate representation in the static single assignment [4, 16, 41] form.

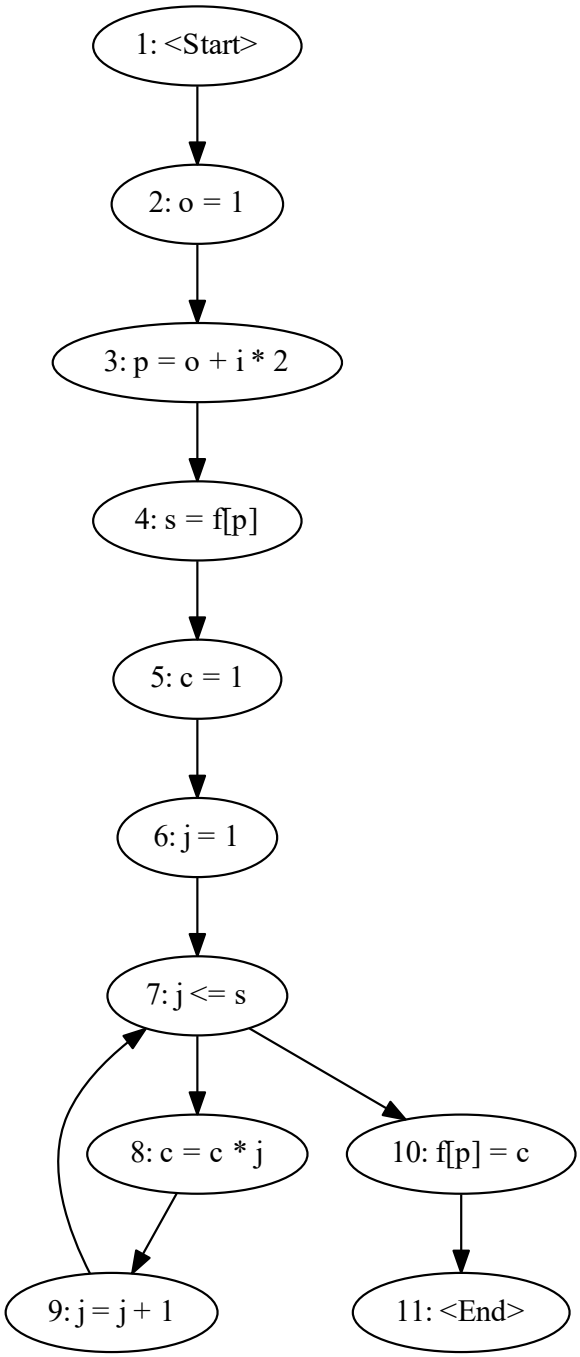

Figure 1: Control flow graph of the code that computes the factorials

Notation	Description
$x@\ell$	x is defined at block ℓ.
$x \mid$	x is loop dependent.
$x \nmid$	x is not loop dependent.
$x = 0$	x is zero.
$x \neq 0$	x is not zero.
$x = 1$	x is one.
$x+$	x is positive.
$x-$	x is negative.

Table 1: Information notation

The most interesting symbol is the loop dependence symbol denoted with a vertical dash[1] $x \mid$. This property identifies that the variable x is the loop index or a variable derived from it. Any access to the same dimension of an array with the variable x and the same definition having the loop dependence symbol is guaranteed to not intersect with itself between iterations. The complement of $x \mid$ is $x \nmid$. It states that the variable x does not depend on the loop index.

The meanings of the remaining symbols are straightforward and are compactly listed in Table 1 for reference.

2.3 Rules

This section introduces the rules to apply to the given integer operation. There is a slight difference to the notation in Section 2.2. The rules do not generate tuples which combine information with a particular variable. Linking the information with a variable is the responsibility of the step explained in Section 2.4.1. This separation allows that the right-hand side of an assignment, or expressions in general, can be computed independently.

2.3.1 Loop Index

(1) specifies the set of information attached to the loop index itself. This set is used as the extremal values of the data flow analysis.

$$\{\mid, \neq 0, +\} \tag{1}$$

[1] The idea for the dash comes from the increasing value of the loop index. It moves from the lower to the upper bound.

Constant	Generated Information
0	$\{\nmid, = 0\}$
1	$\{\nmid, \neq 0, = 1, +\}$
$2, 3, 4, ...$	$\{\nmid, \neq 0, +\}$
$-1, -2, -3, ...$	$\{\nmid, \neq 0, -\}$

Table 2: Information generated by constants

While $\neq 0$ is technically not always correct, it simplifies the other rules and does not void the correctness. The + property denotes that loop index has an increasing value. Hence, it has to be replaced with $-$ for decreasing indices.

2.3.2 Constants

The information generated by integer constants is self-explanatory, thus is only listed in Table 2 for brevity.

2.3.3 Variables

The information a variable provides is retrieved from the state at the entry of the current block ℓ. For instance, consider the entry state[2] of the block 3 sketched in (2).

$$\begin{aligned} Entry(3) = \{&i@?, i\mid, i \neq 0, i+, \\ &o@2, o \neq 0, o = 1, o+, o\nmid\} \end{aligned} \tag{2}$$

The special label ? attached to the loop index i depicts that the variable was defined outside the loop's body.

A variable's information is fetched by filtering all entries by the desired variable. (3) and (4) illustrate the information for the variables i and o respectively. The definition property is omitted as it is irrelevant for the transfers.

$$Information(i) = \{\mid, \neq 0, +\} \tag{3}$$
$$Information(o) = \{\nmid, \neq 0, = 1, +\} \tag{4}$$

2.3.4 Binary Expressions

Tables 7 to 11 of Appendix A list the transfer rules for binary expressions. The columns *Left* and *Right* express the conditions that have to be satisfied when transferring information. Both columns specify a set of properties that has to be attached to the respective operand. For example, if the column *Right* holds the set $\{\mid, +\}$, it requires that the properties $\mid$ and + are attached to the right operand. If this condition is not satisfied, the information of the current row may not be transferred. The column *Transferred* denotes which information is transferred when the associated conditions are fulfilled. The rows captioned with *Copy* identify opportunities where information can be copied from the specified operand safely. To summarize, each row of the transfer rules has to be checked if the property of the row can be transferred. Therefore, it is possible to transfer multiple properties for a single binary expression.

As a general example how to employ the transfer of a binary expression, consider the expression depicted in (5) which is an extract of the node with label 2.

$$o + i * 2 \tag{5}$$

(6) to (8) summarize the information attached to the operands.

$$Information(o) = \{\nmid, \neq 0, = 1, +\} \tag{6}$$
$$Information(i) = \{\mid, \neq 0, +\} \tag{7}$$
$$Information(2) = \{\nmid, \neq 0, +\} \tag{8}$$

The variable i is the loop index, thus it has the $\mid$ information attached. The application of the rules within nested expressions follows conventional evaluation rules; this means that in this example the multiplication is evaluated before the addition. In other words, the corresponding expression tree sketched in Figure 2 is evaluated bottom-up.

As a consequence of the evaluation order, the multiplication transfer rules are applied first. The $\mid$ property is transferred because the left operand contains the properties $\{\mid, +, \neq 0\}$, and the right contains $\{+, \neq 0\}$. Moreover, + and $\neq 0$ are transferred since both operands contain $\{+\}$ and $\{\neq 0\}$ respectively. These transfers ultimately lead to the information as (9) denotes.

$$Information(i * 2) = \{\mid, \neq 0, +\} \tag{9}$$

Next, the transfer rules for additions have to be applied. Since the right operand has the information $\{\mid, +\}$ and the left has $\{+\}$ attached, the $\mid$ property is transferred. Moreover, the properties + and $\neq 0$

[2] More details about the states follow in Section 2.4.

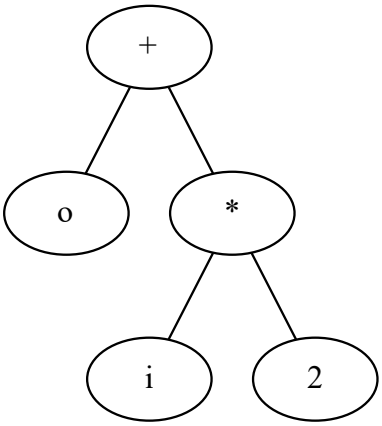

Figure 2: Expression evaluation tree

are transferred because the $\{+\}$ condition for both operands is satisfied. (10) depicts the resulting information after the application of the addition transfer rule.

$$Information(o + i * 2) = \{|, \neq 0, +\} \tag{10}$$

2.3.5 Limitations

Expressions without rules do not generate any information. Furthermore, these rules are only defined for integer operations; thus, operations with any other data type — e.g., decimals — are undefined and do not generate any information as well. However, the rule-set can be enriched to extend the support of binary expressions.

2.4 Data Flow Analysis

The identifiers used to express the states of the data flow analysis are inspired by the ones used by Nielson et al. [38]. $Entry(\ell)$ identifies the set of information when entering whereas $Exit(\ell)$ identifies the set when exiting the block with label ℓ[3]. In other words, these two sets represent the states before and after the execution of the block.

The data flow analysis is a forward analysis [38] with similarities to an instance of the monotone framework [38]. The specialty lies in the merge function. It is neither an intersection nor a unification. While it does employ a unification, it does filter conflicting definitions of the same variables.

In the upcoming sections, Section 2.4.1 describes how the transfer from the entry to the exit state of a block is accomplished, Section 2.4.2 characterizes the merge step, and Section 2.4.3 how the data flow analysis can be solved with the help of a simple iterative worklist algorithm. Last but not least, Section 2.4.4 describes how to interpret the analysis results. It should be noted that some parts of these sections are only in this thesis for the sake of completeness. Therefore, large parts — especially Section 2.4.3 — may be skipped by readers familiar with data flow analysis.

2.4.1 Transfer the Entry to the Exit State

Transferring the data at the entry of a block to construct the exit state is accomplished with the help of the transfer rules introduced in Section 2.3. Therefore, the exit state can be expressed in the form it is illustrated in (11).

$$Exit(\ell) = Transfer_\ell(Entry(\ell)) \tag{11}$$

The mutations of the $Transfer_\ell$ function depend on the block. All blocks but the assignments simply do nothing except copy the data from the entry. Therefore, the exit state will be equal to the entry state. For assignments, transfers are only applied if it is direct to a variable itself. That is why assignments to arrays do not alter the information.

If the block is an assignment of the desired form, the expression on the right-hand side of the assignment is evaluated with the rules introduced in Section 2.3. The resulting information is then linked to the assigned variable x and enriched with the definition tuple $x@\ell$ identifying that the last assignment occurred in the block ℓ. However, before unifying the new information with the entry state, any tuple containing the written variable x has to be removed. To summarize, assignments of the form $x = e$ can be transferred in a way as it is sketched in (12).

$$\begin{aligned} Exit(\ell) = &(Entry(\ell) \setminus \{(x\ i) \mid (x\ i) \in Exit(\ell)\}) \\ &\cup \{(x\ n) \mid n \in Information(e)\} \\ &\cup \{x@\ell\} \end{aligned} \tag{12}$$

In (12), the tuple $(x\ i)$ represents an arbitrary tuple containing information related to the variable x, for example, $x+$.

[3] Another commonly used notation to express these two states are the identifiers $in(\ell)$ and $out(\ell)$.

Reconsider the assignment `p = o + i * 2` of the block 3. The application of the expression rules in Section 2.3.4 resulted in the set $\{\neq 0, +, |\}$. Therefore, the information that will be unified with the truncated entry state is as (13) depicts.

$$\{p@3, p \neq 0, p+, p\,|\} \tag{13}$$

Because the assignment does not overwrite any existing data, no tuples are removed, and the exit state is as (14) shows. For readability, the newly added properties are highlighted.

$$\begin{aligned} Exit(3) = \{&i@?, i\,|, i \neq 0, i+, o@2, o \neq 0, o\,\text{!}, \\ &o = 1, o+, \boxed{p@3, p \neq 0, p+, p\,|}\,\} \end{aligned} \tag{14}$$

2.4.2 Merge the Predecessors' Exit States

The initial step when merging information at the entry of the block ℓ is unifying the exit values of its predecessors ℓ'. After the unification was applied, it removes any conflicting information of variables. This removal is achieved with the help of the embedded reaching definitions analysis. If there is a variable that has multiple distinct definitions, it is a conflicting variable that has to be removed. Practically, the entry of the block ℓ can be constructed in a way it is demonstrated in (15) whereas $Conflicts(\ell)$ is the set of conflicting tuples to be removed.

$$Entry(\ell) = \left(\bigcup_{\ell' \in Pred(\ell)} Exit(\ell') \right) \setminus Conflicts(\ell) \tag{15}$$

However, it is crucial for the data flow analysis to retain the reaching definitions tuples $x@\ell$. For example, if there are two nested if statements where each assigns a different value to a variable. If the merge operation would remove the reaching definitions tuples as well, the conflict of the variable definitions may no longer be identified when merging the outer if statement.

As an example application of the merge logic, consider the block with the label 7. This block has two predecessors: 6 and 9, whereas (16) and (17) depict their exit states[4]. The relevant properties of this example are highlighted.

$$\begin{aligned} Exit(6) = \{&i@?, i\,|, i \neq 0, i+, o@2, o \neq 0, o\,\text{!}, \\ &o = 1, o+, p@3, p \neq 0, p+, p\,|, \\ &s@4, \boxed{c@5, c \neq 0, c\,\text{!}, c = 1, c+}, \\ &j@6, j \neq 0, j\,\text{!}, j = 1, j+\} \end{aligned} \tag{16}$$

$$\begin{aligned} Exit(9) = \{&i@?, i\,|, i \neq 0, i+, o@2, o \neq 0, o\,\text{!}, \\ &o = 1, o+, p@3, p \neq 0, p+, p\,|, \\ &s@4, \boxed{c@8, c \neq 0, c\,\text{!}, c = 1, c+}, \\ &j@9, j \neq 0, j\,\text{!}, j = 1, j+\} \end{aligned} \tag{17}$$

Obviously, there are distinct definitions of the variable c. One by the block with the label 5 and one by the block with the label 8. Therefore, it is required to remove any information—except the reaching definitions—attached to the variable c. This situation also applies to the variable j. Hence its information has to be removed too. (18) illustrates the resulting entry state of the block 7.

$$\begin{aligned} Entry(7) = \{&i@?, i\,|, i \neq 0, i+, o@2, o \neq 0, o\,\text{!}, \\ &o = 1, o+, p@3, p \neq 0, p+, p\,|, \\ &s@4, \boxed{c@5, c@8}, j@6, j@9\} \end{aligned} \tag{18}$$

Because of the restrictive merge operation, the analysis allows safe usage of subtractions when calculating the position of an array index. It is not possible to encode loop dependence information without the usage of variables marked as loop dependent. The same applies when removing loop dependence information from a variable. Consider the example illustrated in Listing 3. The variable `i` is the loop index and initially assigned to the variable `a` on line 2. In practice, the illustrated loop resets the value of `a` to 0 with consecutive subtractions. Because of the merge operation, the loop dependence information is removed from the variable `a` since there are two distinct assignments. Therefore, the assignment to variable `b` on line 6 does not provide any information.

The same applies if a variable may hold distinct values due to branches. Listing 4 sketches a branch where the variable a may either hold the value 2 or 3, depending on the current iteration. If the variable `a` was used to compute an index for array access after this branch, there is the possibility of

[4]The exit state of block 9 represents the first iteration of the solving algorithm. Later iterations will have different information due to the change of the exit state of block 7.

```
1  // 'i' is the loop index
2  var a = i;
3  for(var j = 0; j < i; ++j) {
4    a = a - 1;
5  }
6  var b = a;
```

Listing 3: Removal of the loop dependence

```
1  // 'i' is the loop index
2  var a = 3;
3  if(i%2 == 0) {
4    a = 2;
5  }
```

Listing 4: Distinct values because of branch

intersections between the iterations. Again, this case is prevented by the merging rule.

2.4.3 Solve the Data Flow Analysis

The data flow analysis can almost be solved identically to conventional instances of the monotone framework, except for the merge operation. Algorithm 1 depicts one possible approach.

The inputs of the iterative worklist algorithm [28, 37, 38] are the control flow graph Cfg and the loop index i. The first loop initializes the entry state of the start node of the control flow graph with the extremal values[5]—the information about the loop index. All other entry states are initialized with an empty set. After the initialization of the entry states, all exit states are initialized with an empty set and the nodes are added to the worklist.

The computation runs until the worklist is empty. The worklist is not empty as long as there was a change to an exit state. If a change occurred, all the successors of the currently processed block ℓ are added to the worklist.

The merge step merges the exit states of all predecessors for any but the start node. Therefore, the entry state of the start node stays untouched. After the states were merged, the algorithm discards variables with conflicting definitions from the set.

[5]It is possible to enrich the extremal values with information about variables declared outside of the loop to improve the results of the analysis.

Data: Cfg, i
Result: $Entry$, $Exit$
$worklist \leftarrow \emptyset$;
foreach $\ell \in blocks(Cfg)$ **do**
  if $\ell = start(Cfg)$ **then**
    $Entry(\ell) \leftarrow \{i@?, i\,|, i \neq 0, i+\}$;
  else
    $Entry(\ell) \leftarrow \emptyset$;
  end
  $Exit(\ell) \leftarrow \emptyset$;
  $worklist \leftarrow worklist \cup \{\ell\}$;
end
while $worklist \neq \emptyset$ **do**
  $\ell \leftarrow head(worklist)$;
  $worklist \leftarrow tail(worklist)$;
  $oldExit \leftarrow Exit(\ell)$;
  if $\ell \neq start(Cfg)$ **then**
    $Entry(\ell) \leftarrow \bigcup\{Exit(\ell') \mid \ell' \in predecessors(\ell)\}$;
  end
  $Entry(\ell) \leftarrow Entry(\ell) \setminus Conflicts(\ell)$;
  $Exit(\ell) \leftarrow Transfer_\ell(Entry(\ell))$;
  if $Exit(\ell) \neq oldExit$ **then**
    $worklist \leftarrow worklist \cup successors(\ell)$;
  end
end
return $Entry$, $Exit$

Algorithm 1: Iterative worklist algorithm to solve the data flow analysis

Finally, $Transfer_\ell$ transfers the entry state to the exit state.

The result of the algorithm are the computed entry and exit states of all blocks within the control flow graph. The final states of the example are listed in Tables 12 and 13 of Appendix B.

2.4.4 Interpret the Results

The use of the analysis results is straightforward. However, without the utilization of an alias analysis [37], any array has to be treated as if one single array was accessed. This conservative simplification is possible because it implicitly covers all possibilities of array aliasing.

The first step is the collection of any array access and the node where this access been identified. In the initially introduced example, that would be the reading from `f[p]` in block 4 and writing to `f[p]`

```
1 int m = a.Length - 1;
2 for(int i = 0; i < m; i++) {
3   var x = i;
4   var o = a[x + 1];
5   a[x + 1] = o + 1;
6 }
```

Listing 5: Array access with offset

```
1 int m = a.Length - 1;
2 for(int i = 0; i < m; i++) {
3   var x = i;
4   var t1 = x + 1;
5   var o = a[t1];
6   var t2 = t1;
7   a[t1] = o + 1;
8 }
```

Listing 6: Optimized array access with offset

in block 10. For both blocks, the entry state of the respective block is queried for the array accessor p. (19) sketches the result of the two state queries that happens to be the same for both as the variable p is not written between the blocks 4 and 10.

$$\{p@3, p \neq 0, p+, p\,|\} \tag{19}$$

With the retrieved information, three checks are essential. The first one ensures that each accessor has the loop dependence property | attached. Next, it is necessary to prove that the accessor is always targeting the same dimension. The third and last check is ensuring that all have the same defining block. It is unnecessary to verify that the same variable was used because this is implicitly given by the definition check. If all three checks are successful, there are no array access intersections between the loop iterations.

The factorial example has obviously no intersecting accesses to arrays. Furthermore, the loop does not contain any other kind of write access to shared memory. Consequently, the loop is parallelizable.

2.5 Opportunities

Adapting the analysis to an implementation for the use with an intermediate representation that is in the static single assignment form drastically simplifies the analysis. First of all, the embedded reaching definitions analysis becomes obsolete. The transfers can be executed top-down, definition by definition. Moreover, the merge function is no longer necessary as it can be expressed as a ϕ-function which does not produce any information.

As initially mentioned, a variety of optimization techniques can improve the results of the introduced analysis. For example, consider the situation sketched in Listing 5 which constructs the array indices with an offset. The lines 4 and 5 both access the array at the same position. Nevertheless, they compute the index in two distinct locations. Therefore, there is no guarantee that the value of the variable x did not change in-between. One possibility would be to check the definitions of all the accessed variables, and that the expressions are equal. A more sophisticated approach is the application of a common subexpression elimination [37] and copy propagation [37]. Listing 6 illustrates a possible result of the utilization of these two optimizations. Both accesses are made with the same variable having the same definitions. Thus the loop can safely be parallelized and is also detected by the prototype introduced in Section 3.1.

3 Evaluation

To verify the usefulness of the introduced analysis, a prototype which makes use of it has been implemented. Section 3.1 summarizes further details about said prototype. The results of the experimental evaluation are outlined in Section 3.2.

3.1 Prototype

The prototype is a static code analyzer implemented in C# with the help of the .NET Compiler Platform (Roslyn) [33]. It is a Visual Studio [35] plugin that automatically scans the for loops within the documents and reports opportunities for parallelization. For the sake of simplicity, the interprocedural analysis is only context insensitive. To avoid cases where methods are overridden with polymorphism, only non-virtual methods are supported. The prototype works conservatively with the goal of guaranteed correctness if a loop is parallelized.

Internally, the prototype transforms the provided

C# code into a three-address intermediate representation. During the conversion process, semantic checks prevent ambiguities of shadowed variables. Of course, the conversion fails for unsupported language features, and the loop is not analyzed further. Moreover, the prototype applies a copy propagation and common subexpression elimination to improve the results of the loop dependence analysis. An alias analysis is used to support accesses to distinct arrays.

Besides a small set of white-listed static methods of the .NET Framework, the prototype supports the following language features: 1) auto-properties 2) binary and unary expressions 3) coalesce expressions 4) compound assignments 5) conditional expressions 6) continue and break statements for inner loops 7) for-, while-, and do-statements 8) method invocations 9) multi-dimensional arrays 10) string interpolation.

3.2 Results

The evaluation was made by analyzing 21 open source projects available on GitHub [27] with a total of 33,893 for loops. The selected projects are listed in Appendix C and are considered mature regarding size, age, and the number of given stars. The majority of the projects were chosen because they provide functionality in either data processing, image processing, or machine learning; expecting a a better chance for parallelizable loops. Throughout this section, a loop is considered parallelizable if it can be transformed without any further code adjustments except for the `Parallel.For` [3, 34] instruction itself.

Table 3 illustrates the summary of seven open source projects. Each of the 299 present loops inside these projects were manually reviewed. About 18.7% of all loops are parallelizable whereas 12.5% of them have been reported by the prototype. Moreover, Table 3 highlights the reliability of the prototype, as it did not report any of the 243 loops that cannot be executed in parallel.

Further data was collected by adapting the prototype in a way that it also indicates loops that may have array access intersections. This adaption allows quick identification of loops that are parallelizable but would require a stronger analysis than the DFA introduced in this thesis. However, it should be remembered that it is still limited by the prototype's capabilities. That is why a selection of projects received the full review previously. Twenty of the scanned projects account for 5,359 loops whereas the relaxed prototype reported 223 of them. Twenty-four of these loops have intersecting array accesses; thus, are not parallelizable. Table 4 depicts that 92% of all the identified loops only use the loop index to access an array element. Only eight percent compute the position to access a particular array element. However, only three percent of the loops would require a stronger analysis than the introduced one. On the one hand, this observation shows that the introduced analysis might be too powerful, but on the other hand, it appears that the application of more sophisticated techniques bears an even higher overhead.

Parallelizable	Total	Reported	%
No	243	0	0.0
Yes	56	7	12.5

Table 3: Ratio of parallelizable loops

Array Access Kind	#	%
Direct Access with Loop Index	183	92
Data Flow Analysis Sufficient	10	5
Stronger Analysis Necessary	6	3

Table 4: Identification of array accesses

To identify the possible gain of the parallelization, Table 5 categorizes the identified parallelizable loops. If a body of a loop does not execute more than five instructions, it is considered as a loop that either initializes an array or copies the data from another array. Out of all the identified loops, a third does more than initializing or copying an array. Therefore, there is a good possibility to profit from the parallelization of these loops as long as these loops work with sufficiently sized arrays. Moreover, it can be expected that there exist more computational intensive loops, which are beyond the capabilities of the prototype.

The last scanned project is the machine learning framework Accord.NET [21] which comprises 28,535 for loops. This project is used to evaluate the reliability of the prototype and the analysis on a

Loop Kind	#	%
Initialization	77	39
Copy	54	27
Computation	68	34

Table 5: Loop classification

	Sequential	Parallel	Speedup
Mandelbrot	30,336.2	5,460.6	5.6
Convolution	4,329.6	1,420.8	3.0
Genetics	7,578.0	5,082.0	1.5

Table 6: Average execution time (in ms) of five runs and the resulting speedup factor

large scale project. The prototype claims that 3,650 (12.8%) loops are parallelizable. However, instead of a manual review of these loops, the refactoring was applied. Unfortunately, the refactoring transformed only 800 in total. This limitation is reasoned to the fact that the other loops are all located in generated code[6]. The execution of the 3,000 unit tests—with a code coverage of roughly 34.2%—on the refactored code showed that the framework still functions as expected. This application underpins the correctness and stability for both the prototype and the analysis. However, as the unit test execution required about one and half times as much time as before, there was no benefit in applying the refactoring throughout the application.

Nonetheless, a speedup from parallelizing for loops is possible. To evaluate this statement, a collection of different problems have been implemented. This collection incorporates the mandelbrot set [31], various convolution filters [24], and a genetic algorithm [7] to solve the max-one problem. The tests were run on an Intel Core i7-4770k CPU (3.50 GHz, 4 Cores) [17]. Table 6 summarizes the results (more details can be found in Tables 14 and 15 of Appendix D). It can be seen that the speedup factor ranges from 1.5 up to 5.6, and thus the parallelization of the loops was beneficial. However, the prototype was only able to detect the parallelization opportunity for the mandelbrot and convolution implementations. The implementation of the genetic algorithm was not reported because it uses language features which are not supported by the prototype.

To sum up, this section illustrated the reliability of the introduced analysis and the implemented prototype. This reliability raises the question why it was not used to automate the conversion of for loops. While this extension is certainly possible, the benefits are limited. A significant part of the parallelizable loops are not computationally intensive, and thus the parallelization of these would lead to reduced performance. The other loops would only benefit if they do heavy work or process a significant amount of data. Therefore, an approach that estimates the gain is desirable. However, the goal of the analysis is to proof the absence of array access intersections and not the profit of the parallelization.

[6]It appears that the .NET Compiler Platform or Visual Studio prevent the refactoring of the generated code.

4 Related Work

There already exists a variety of utilities aiding engineers in the development of parallel code. This section briefly introduces a selection of different approaches. Unfortunately, none of these projects actively informs the user about the parallelization opportunities.

Pluto [8, 9, 10, 39] is a C source to source transformer that makes use of the polyhedral model. Although it does its transformations on the source code, it is more of a pre-compiler because some restructurings Pluto provides yield loops that can be very different to the original implementation. Loops meant for parallelization have to be marked with a special compiler directive. If the code is parallelizable, Pluto may apply loop transformations and enriches the code with suitable OpenMP [2, 18] directives.

Another interesting approach is Hydra [13]. It operates on the intermediate representation of LLVM [12, 29, 30]. Its underlying idea can be seen as C#'s async/await [3, 32, 36] principle. However, instead of manually declaring fragments with `async` and `await` respectively, it does so automatically. To accomplish its job, Hydra searches for parallelizable code fragments and computes the cost of the sequential and the parallel code. If the cost of the parallel code is lower than the original sequential,

the function invocation is offloaded to a thread pool.

Sambamba [43, 44] also tries to automatize the parallelization process. However, besides the conservative static code analysis on LLVM's bitcode at compile time, it also incorporates a runtime system. During the application's execution, the runtime system further collects information and exchanges methods where appropriate. Information that was collected at runtime is stored in a persistent storage so it can be reused in later program executions. Moreover, Sambamba involves a software transactional memory system to guard parallelized code where needed. This STM system allows, different to the previous two approaches, the parallelization of code fragments that require synchronization.

Baar [5, 19, 20] is an approach that transparently offloads computationally intensive program segments to a server. It provides an environment to allow the execution of programs in LLVM's IR. The goal is the identification of hotspots at runtime. The identified and suitable hotspots are offloaded to a server, which applies further processings such as parallelization and vectorization. The automatic parallelization is thereby accomplished with the LLVM subproject Polly [26, 40], which makes use of the polyhedral model. The Message Passing Interface (MPI) [25] is used to automatically use the most suitable communication channel; reducing the negative impact of expensive data transfers between client and server.

Last but not least, Concurrencer [14, 22] is an Eclipse [23] plugin to refactor Java code. Its most interesting feature is the ability to transform a divide and conquer based algorithm into a parallel implementation, wich uses Java's fork-join pool. This transformation is accomplished with the algorithm introduced by Rugina et al. [42]. Unfortunately, Concurrencer does not guarantee the correctness of the refactored code.

5 Conclusion

This thesis introduced a conservative data flow analysis to aid the parallelization of loops. Nonetheless, there are far more code constellations that can be parallelized than just loops. The analysis applies various transfer rules for different expressions. The resulting information can then be used to proof the absence of array access intersections. Although it is possible to use this technique for automatic conversion, a benefit can only be expected when estimating the possible speedup due to the high number of non-intensive loops. That is why the prototype works in a suggestive way with guaranteed correctness. This suggestive approach is also the main difference to related work which tend to automatize the parallelization or just simplify the refactoring process. However, the data flow analysis cannot reasonably handle transfers from branches. This is lost parallelization potential if all loop iterations take the same execution path. Moreover, the analysis is incapable of identifying distinct accesses to arrays where two or more loop indices are used to compute the index. Nevertheless, the data flow analysis as well as the prototype can be extended further. The analysis could be extended with additional transfer rules, for example, one that handles array size queries. An extension opportunity for the prototype would be informing the engineers about the code fragments that prevent the parallelization of a loop.

References

[1] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, and tools, 2/e*. Addison wesley Boston, 2003.

[2] Shameen Akhter and Jason Roberts. *Multi-core programming*, volume 33. Intel press Hillsboro, 2006.

[3] Joseph Albahari and Ben Albahari. *C# 6.0 in a Nutshell: The Definitive Reference*. O'Reilly Media, Inc., 2015.

[4] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 1–11, New York, NY, USA, 1988. ACM.

[5] Baar (binary acceleration at runtime). `https://github.com/pc2/baar`. Accessed: 2017-05-30.

[6] Uptpal Banerjee. Dependence testing in ordinary programs. Master's thesis, University

of Illinois, Department of Computer Science, 1976.

[7] Ulrich Bodenhofer. *Genetic Algorithms: Theory and Applications, 3/e*. Lecture notes, Fuzzy Logic Laboratorium Linz-Hagenberg, Winter, 2003.

[8] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction*, pages 132–146. Springer, 2008.

[9] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Notices*, volume 43, pages 101–113. ACM, 2008.

[10] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer. Technical Report OSU-CISRC-10/07-TR70, The Ohio State University, October 2007.

[11] Uday Kumar Bondhugula. *Effective automatic parallelization and locality optimization using the polyhedral model*. PhD thesis, The Ohio State University, 2008.

[12] Robert Chansler, Russell Bryant, Roy Bryant, Rosangela Canino-Koening, Francesco Cesarini, Eric Allman, Keith Bostic, and Titus Brown. The architecture of open source applications. 2011.

[13] James Chicken. Hydra: Automatic parallelism using llvm. 2014. Homerton College.

[14] Concurrencer. `http://refactoring.info/tools/Concurrencer`. Accessed: 2017-04-25.

[15] Michele Conforti, Gérard Cornuéjols, and Giacomo Zambelli. *Integer programming*, volume 271. Springer, 2014.

[16] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[17] Intel Corporation. Intel core i7-4770k processor. `http://ark.intel.com/products/75123/Intel-Core-i7-4770K-Processor-8M-Cache-up-to-3_90-GHz`. Accessed: 2017-05-30.

[18] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[19] Marvin Damschen and Christian Plessl. Easy-to-use on-the-fly binary program acceleration on many-cores. *arXiv preprint arXiv:1412.3906*, 2014.

[20] Marvin Damschen, Heinrich Riebler, Gavin Vaz, and Christian Plessl. Transparent offloading of computational hotspots from binary code to xeon phi. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, pages 1078–1083. IEEE, 2015.

[21] César Roberto de Souza. The accord .net framework. `http://accord-framework.net`, 2014. Accessed: 2017-05-11.

[22] Danny Dig, John Marrero, and Michael D Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *Proceedings of the 31st International Conference on Software Engineering*, pages 397–407. IEEE Computer Society, 2009.

[23] The Eclipse Foundation. Eclipse. `http://www.eclipse.org`. Accessed: 2017-04-26.

[24] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing, 3/e*. Prentice Hall International, Jul 2007.

[25] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[26] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, 2011.

[27] GitHub Inc. Github. `https://github.com`. Accessed: 2017-05-11.

[28] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data flow analysis: theory and practice*. CRC Press, 2009.

[29] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[30] The llvm compiler infrastructure. `http://llvm.org`. Accessed: 2017-04-26.

[31] Benoit Mandelbrot. *Fractals and chaos: the Mandelbrot set and beyond*. Springer, 2004.

[32] Microsoft. Asynchronous programming with async and await (c# and visual basic). `https://msdn.microsoft.com/library/hh191443(vs.110).aspx`. Accessed: 2017-05-23.

[33] Microsoft. .net compiler platform (roslyn). `https://github.com/dotnet/roslyn`. Accessed: 2017-04-21.

[34] Microsoft. Parallel.for method. `https://msdn.microsoft.com/en-us/library/system.threading.tasks.parallel.for(v=vs.110).aspx`. Accessed: 2017-05-23.

[35] Microsoft. Visual studio. `https://www.visualstudio.com`. Accessed: 2017-04-21.

[36] Microsoft. C# language specification version 5.0. `https://www.microsoft.com/en-us/download/details.aspx?id=7029`, 2015. Accessed: 2017-03-06.

[37] Steven S Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[38] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[39] Pluto: A polyhedral automatic parallelizer and locality optimizer for multicores. `http://pluto-compiler.sourceforge.net`. Accessed: 2017-04-24.

[40] Polly - llvm framework for high-level loop and data-locality optimizations. `https://polly.llvm.org`. Accessed: 2017-05-30.

[41] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM.

[42] Radu Rugina and Martin Rinard. Automatic parallelization of divide and conquer algorithms. In *ACM SIGPLAN Notices*, volume 34, pages 72–83. ACM, 1999.

[43] Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. Sambamba - adaptive optimization and parallelization of general purpose programs. `http://www.sambamba.org`. Accessed: 2017-05-29.

[44] Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. Sambamba: Runtime adaptive parallel execution. In *Proceedings of the 3rd International Workshop on Adaptive Self-Tuning Computing Systems*, ADAPT '13, pages 7:1–7:6, New York, NY, USA, 2013. ACM.

[45] Ross Albert Towle. *Control and Data Dependence for Program Transformations.* PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1976. AAI7624191.

A Transfer Rules

Left	Right	Transferred
$\{\mid\}$	$\{\nmid\}$	$\mid$
$\{\mid, +\}$	$\{+\}$	$\mid$
$\{+\}$	$\{\mid, +\}$	$\mid$
$\{\nmid\}$	$\{\mid\}$	$\mid$
$\{\mid, -\}$	$\{-\}$	$\mid$
$\{-\}$	$\{\mid, -\}$	$\mid$
$\{\nmid\}$	$\{\nmid\}$	$\nmid$
$\{+\}$	$\{+\}$	$+$
$\{-\}$	$\{-\}$	$-$
$\{+\}$	$\{+\}$	$\neq 0$
$\{-\}$	$\{-\}$	$\neq 0$
Covered by copy rules.		$= 0$
Covered by copy rules.		$= 1$
$\emptyset$	$\{= 0\}$	Copy from left
$\{= 0\}$	$\emptyset$	Copy from right

Table 7: Transfer rules for additions

Left	Right	Transferred
$\{\mid\}$	$\{\nmid\}$	$\mid$
$\{\mid, +\}$	$\{-\}$	$\mid$
$\{\mid, -\}$	$\{+\}$	$\mid$
$\{\nmid\}$	$\{\mid\}$	$\mid$
$\{-\}$	$\{\mid, +\}$	$\mid$
$\{+\}$	$\{\mid, -\}$	$\mid$
$\{\nmid\}$	$\{\nmid\}$	$\nmid$
$\{+\}$	$\{-\}$	$+$
$\{-\}$	$\{+\}$	$-$
$\{+\}$	$\{-\}$	$\neq 0$
$\{-\}$	$\{+\}$	$\neq 0$
Covered by copy rule.		$= 0$
Covered by copy rule.		$= 1$
$\emptyset$	$\{= 0\}$	Copy from left

Table 8: Transfer rules for subtractions

Left	Right	Transferred
$\{\mid, +, \neq 0\}$	$\{-, \neq 0\}$	$\mid$
$\{\mid, -, \neq 0\}$	$\{+, \neq 0\}$	$\mid$
$\{\nmid\}$	$\{\nmid\}$	$\nmid$
$\{+\}$	$\{+\}$	$+$
$\{-\}$	$\{-\}$	$+$
$\{-\}$	$\{+\}$	$-$
$\{+\}$	$\{-\}$	$-$
$\{\neq 0\}$	$\{\neq 0\}$	$\neq 0$
$\{= 0\}$	$\emptyset$	$= 0$
$\emptyset$	$\{= 0\}$	$= 0$
Covered by copy rules.		$= 1$
$\emptyset$	$\{= 1\}$	Copy from left
$\{= 1\}$	$\emptyset$	Copy from right

Table 9: Transfer rules for multiplications

Left	Right	Transferred
Covered by copy rules.		$\mid$
$\{\nmid\}$	$\{\nmid\}$	$\nmid$
Covered by copy rules.		$= 0$
Covered by copy rules.		$= 1$
$\emptyset$	$\{= 1\}$	Copy from left
$\{= 0\}$	$\emptyset$	Copy from left

Table 10: Transfer rules for divisions

Left	Right	Transferred
$\{\nmid\}$	$\{\nmid\}$	$\nmid$
$\{= 0\}$	$\emptyset$	$= 0$
$\emptyset$	$\{= 1\}$	$= 0$

Table 11: Transfer rules for modulo

B State Solutions

ℓ	$Entry(\ell)$
1	$\{i@?, i\,|, i \neq 0, i+\}$
2	$\{i@?, i\,|, i \neq 0, i+\}$
3	$\{i@?, i\,|, i \neq 0, i+, o@2, o\,\restriction, o \neq 0, o = 1, o+\}$
4	$\{i@?, i\,|, i \neq 0, i+, o@2, o\,\restriction, o \neq 0, o = 1, o+, p@3, p\,|, p \neq 0, p+\}$
5	$\{i@?, i\,|, i \neq 0, i+, o@2, o\,\restriction, o \neq 0, o = 1, o+, p@3, p\,|, p \neq 0, p+, s@4\}$
6	$\{i@?, i\,|, i \neq 0, i+, o@2, o\,\restriction, o \neq 0, o = 1, o+, p@3, p\,|, p \neq 0, p+, s@4, c@5, c\,\restriction, c \neq 0, c = 1, c+\}$
7	$\{i@?, i\,|, i \neq 0, i+, o@2, o\,\restriction, o \neq 0, o = 1, o+, p@3, p\,|, p \neq 0, p+, s@4, c@5, c@8, j@6, j@9\}$
8	$\{i@?, i\,|, i \neq 0, i+, o@2, o\,\restriction, o \neq 0, o = 1, o+, p@3, p\,|, p \neq 0, p+, s@4, c@5, c@8, j@6, j@9\}$
9	$\{i@?, i\,|, i \neq 0, i+, o@2, o\,\restriction, o \neq 0, o = 1, o+, p@3, p\,|, p \neq 0, p+, s@4, c@8, j@6, j@9\}$
10	$\{i@?, i\,|, i \neq 0, i+, o@2, o\,\restriction, o \neq 0, o = 1, o+, p@3, p\,|, p \neq 0, p+, s@4, c@5, c@8, j@6, j@9\}$
11	$\{i@?, i\,|, i \neq 0, i+, o@2, o\,\restriction, o \neq 0, o = 1, o+, p@3, p\,|, p \neq 0, p+, s@4, c@5, c@8, j@6, j@9\}$

Table 12: Entry state solution of the data flow analysis

ℓ	$Exit(\ell)$
1	$\{i@?, i\,|, i \neq 0, i+\}$
2	$\{i@?, i\,|, i \neq 0, i+, o@2, o\,\restriction, o \neq 0, o = 1, o+\}$
3	$\{i@?, i\,|, i \neq 0, i+, o@2, o\,\restriction, o \neq 0, o = 1, o+, p@3, p\,|, p \neq 0, p+\}$
4	$\{i@?, i\,|, i \neq 0, i+, o@2, o\,\restriction, o \neq 0, o = 1, o+, p@3, p\,|, p \neq 0, p+, s@4\}$
5	$\{i@?, i\,|, i \neq 0, i+, o@2, o\,\restriction, o \neq 0, o = 1, o+, p@3, p\,|, p \neq 0, p+, s@4, c@5, c\,\restriction, c \neq 0, c = 1, c+\}$
6	$\{i@?, i\,|, i \neq 0, i+, o@2, o\,\restriction, o \neq 0, o = 1, o+, p@3, p\,|, p \neq 0, p+, s@4, c@5, c\,\restriction, c \neq 0, c = 1, c+,$ $j@6, j\,\restriction, j \neq 0, j = 1, j+\}$
7	$\{i@?, i\,|, i \neq 0, i+, o@2, o\,\restriction, o \neq 0, o = 1, o+, p@3, p\,|, p \neq 0, p+, s@4, c@5, c@8, j@6, j@9\}$
8	$\{i@?, i\,|, i \neq 0, i+, o@2, o\,\restriction, o \neq 0, o = 1, o+, p@3, p\,|, p \neq 0, p+, s@4, c@8, j@6, j@9\}$
9	$\{i@?, i\,|, i \neq 0, i+, o@2, o\,\restriction, o \neq 0, o = 1, o+, p@3, p\,|, p \neq 0, p+, s@4, c@8, j@9\}$
10	$\{i@?, i\,|, i \neq 0, i+, o@2, o\,\restriction, o \neq 0, o = 1, o+, p@3, p\,|, p \neq 0, p+, s@4, c@5, c@8, j@6, j@9\}$
11	$\{i@?, i\,|, i \neq 0, i+, o@2, o\,\restriction, o \neq 0, o = 1, o+, p@3, p\,|, p \neq 0, p+, s@4, c@5, c@8, j@6, j@9\}$

Table 13: Exit state solution of the data flow analysis

C Scanned Projects

Projects with a full manual review:

- GeneticSharp
- ImageProcessor
- LiteDB
- Microsoft BotBuilder
- Nancy
- Popcorn
- SignalR

Projects where primarily the reported loops have been manually reviewed:

- AForge.NET
- BrainSimulator
- CSCore
- C-Sharp-Algorithms
- ImageSharp
- Inbox2 desktop client
- Naiad
- NHibernate
- RavenDB
- SharpBrain
- Spring.NET
- Structure.Sketching
- Veldrid

Scan-only projects with the refactoring application:

- Accord.NET

D Speedup Evaluation

The mandelbrot set was run with 10,000 iterations to generate an image with a resolution of 2560x1440. A 9x9 motion blur convolution filter was used on a 2560x1440 image. The genetic algorithm used for the max-one problem was set up for 100 generations, 1000 individuals, 1000 crossovers and 1000 mutations per generation, and 1000 genes per individual.

	Run 1	Run 2	Run 3	Run 4	Run 5
M	30,307	30,328	30,365	30,373	30,308
C	4,293	4,346	4,337	4,332	4,340
G	7,355	7,450	7,937	7,622	7,526

Table 14: Results (in ms) of the sequential runs of mandelbrot (M), convolution (C), and genetics (G)

	Run 1	Run 2	Run 3	Run 4	Run 5
M	5,522	5,418	5,479	5,453	5,431
C	1,487	1,438	1,390	1,393	1,396
G	5,064	5,122	5,090	5,082	5,052

Table 15: Results (in ms) of the parallel runs of mandelbrot (M), convolution (C), and genetics (G)