A Refactoring Tool for the Safe Parallelization of Array-Centric For-Loops

Christoph Amrein and Luc Bläser

HSR Hochschule für Technik Rapperswil, Switzerland
christoph.amrein@hsr.ch, luc.blaeser@hsr.ch

Abstract. We present a software refactoring tool for loop parallelization. By using a light-weight static analysis, the tool conservatively detects sequential for-loops with mutually independent loop steps that can be safely transformed to corresponding parallel for-loops. We implemented this tool for C# as a Visual Studio IDE plugin and applied it to several open-source projects. Our evaluation shows that the analysis is fast and effective in detecting parallelizable array-centric for-loops.

Keywords: Concurrency · Parallelization · Refactoring · For-Loops · Loop-Carried Dependence · C# · Dataflow Analysis

1 Introduction

Modern software is still unnecessarily sequentially programmed, even if today's programming languages feature light-weighted parallelization constructs and efficient parallelization libraries, such as for example the task parallel library (TPL) [20] in .NET with its parallel invocation or parallel loop statements. A reason for this is that programmers have to explicitly opt-in for parallelization by selecting the corresponding parallel program idioms, instead of having a system or IDE tool that automatically assists in parallelizing the right code fragments. For this reason, various research has been conducted on the topic of automatic parallelization, at the level of the compiler [1, 5, 4, 23, 28] or the runtime system [26, 27].

In this work, we focus on a specific case of automatically inferred parallelization, namely *for-loop parallelization*. Our goal to is develop an IDE tool that interactively assists programmers in identifying for-loops that can be safely parallelized, i.e., without altering the functional behavior of the loop, except its execution speed. The tool eventually transforms such for-loops to their corresponding parallel versions. We concentrate on for-loops that have mutually independent loop steps and are therefore straightforward to be replaced by their parallel counterpart, without introducing synchronization in the parallel loop bodies.

To identify parallelizable loops, we focus ourselves on independent loops. This is also known as proving the absence of loop-carried dependencies [2]. A loop-carried dependency is given if an instruction of an iteration depends on an

instruction of another iteration. Conversely, loop iteration steps are independent if they do not access the same memory locations except for read-only accesses, and if they do not perform any side effects via external function calls, such as I/O. In many programming languages, including C#, memory locations are defined at the granularity of variables or array elements. Therefore, loops are guaranteed to be independent if the write accesses per loop step target distinct memory locations, and if none of the loop steps calls side-effecting functions.

A particular challenge in the analysis of loop independence is the verification that there is no array access intersection when at least one element of the array is written in the loop, i.e., each iteration accesses a distinct set of elements of the array. For example, consider the code sketched in Listing 1. This example computes the sum of all entries of the array a up to the position i and writes it to the position i. A loop-carried dependency occurs in the line 6 because it depends on the current iteration i and the previous iteration $i-1$. This loop hence has dependent iterations steps, and will not be parallelized by our tool.

```
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    a[i] = a[i - 1] + a[i];
7  }
```

Listing 1. Example of a dependent for-loop that will not be parallelized.

```
1  int[] a = new int[] {
2    1, 2, 3, 4, 5, 6
3  };
4  int m = a.Length;
5  for(int i=0; i < m; i++) {
6    a[i] = a[i] * a[i];
7  }
```

Listing 2. Example of an independent loop that can be parallelized.

In contrast, Listing 2 sketches an independent and thus parallelizable loop. This for-loop squares the value at the array position i and stores it again at the same position. All accesses in the line 6 access a distinct region of the array a per iteration, such that this loop is free of array access intersections.

There exist well-known static loop dependency analysis methods, such as the polyhedral model [1, 5], or the GCD dependence test [4, 23, 28]. While these methods can detect specific complex cases, our motivation was to design a simple alternative algorithm, that is both fast and effective in practice. This is particularly important in an IDE refactoring tool that should respond fast to code changes. Our method also supports parallelization of array-centric loops with non-linear array index functions. For this purpose, we designed a light-weight dataflow analysis that infers potential array access intersections, with a relatively limited, cheap context information. This analysis has been integrated into our parallel refactoring tool for C# that works as a plugin in Visual Studio IDE. It

highlights for-loops that can be safely parallelized, and on the consent of the developer, parallelizes these for-loops, by using the .NET TPL's `Parallel.For`.

In summary, this paper makes the following contributions:

- Description of a new dataflow analysis to determine the absence of array access intersections, that experimentally proves to be fast and effective for array-centric for-loops.
- Report of the implementation and evaluation of a practical refactoring tool for loop parallelization in C# and Visual Studio based on this analysis.

The remainder of this paper is structured as follows: Section 2 presents the dataflow analysis method. Section 3 descibes the implementaton of our refactoring tool based on this analysis. Section 4 reports on the experimental evaluation of the analysis and the refactoring tool. Section 5 discusses related work. Section 6 finally concludes this paper.

2 Analysis

Instead of applying potentially expensive computations, the analysis uses a set of simple transfer rules on the static single assignment (SSA) form. These transfer rules are designed to provide sufficient information to decide whether an array index expression is an injective function. In other words, it helps deciding if each iteration in the loop computes unique array indices. One essential property of the analyzed array index expressions is that they are dependent on the loop index. For instance, if the function $f(x)$ is injective, using the loop index as function argument yields a unique value for each iteration. Therefore, an array access such as $a[f(i)]$, with i representing the loop index, is independent in each loop step. Consequently, the identity function $f(x) = x$ is the simplest function that satisfies this property.

This section outlines our dataflow-based loop dependency analysis: Section 2.1 introduces an example that is used throughout the sections to explain the different steps of the analysis. Section 2.2 specifies requirements for the for-loops that are covered by the analysis. Section 2.3 introduces our notation to encode analysis properties. Section 2.4 summarizes the property inference used in the dataflow analysis. Section 2.5 specifies the dataflow analysis that computes the properties for loop code. Section 2.6 explains how the dataflow analysis output allows determining potential array access intersections. Section 2.7 finally combines the computed information to determine whether the loop is independent and can be safely parallelized. Section 2.8 describes further static analyses that can increase the precision in our loop dependency analysis.

2.1 Example

The analysis procedure is introduced with the help of the example sketched in Listing 3. The illustrated code computes the factorial of every odd array index and stores the factorial at this position. The outer loop declared in line 5 iterates

```
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 3. A loop which computes the factorial of every odd array index.

```
1   o1 ← 1
2   p1 ← o1 + i * 2
3   s1 ← f[p1]
4   c1 ← 1
5   j1 ← 1
6   label 1
7   if φ(j1, j2) <= s1 goto 2
8   goto 3
9   label 2
10  c2 ← φ(c1, c2) * φ(j1, j2)
11  j2 ← φ(j1, j2) + 1
12  goto 1
13  label 3
14  f[p1] ← φ(c1, c2)
```

Listing 4. Body of the loop that computes the factorials in the SSA form.

over every second element, while the inner loop in line 11 computes the factorial of the current element.

Listing 4 shows the body of the loop that iterates over the array elements in the SSA form. The loop index i and the array f have no value numbering since their write accesses and definitions only occur outside of the body. Instead of denoting the SSA form as a graph, we here sketch the code with labels and goto instructions. The line numbers are later used as labels to refer to the specific basic blocks.

2.2 Prerequisites

For our analysis, we focus on for-loops that have the form as sketched in Listing 5. Conceptually, other loop patterns, e.g. reverse or multi-step loops, could be handled as well, by transforming them to our required form. The `type` of the loop index has to be an integer, such as `int` or `long`. The identifier `low` defines the lower bound and `up` the upper bound of the loop. The implementation also supports the logical operator `<=` to define the upper bound as well as `i+=1` for the single step increment. An iteration may start with a negative value and end with a positive one. We currently concentrate on this specific template of single-step forward for-loop as this corresponds to the only supported variant of the `Parallel.For` program construct of .NET. Moreover, we require that the analyzed sequential for-loops are correct, e.g., have no integer overflows, no concurrency issues, etc. inside the loop bodies. We need this assumption, as we

```
1  for(type i = low; i < up; i++)
```

Listing 5. Definition of the for-loop structure.

do not prove these elementary correctness conditions and could otherwise effect subsequent errors through the parallelization.

2.3 Notation

This section introduces some elementary notation that we use in our analysis and it is outlined in Table 1. We generally denote sets with capital letters. The identifier ℓ stands for the label of a particular basic block of the loop body in the SSA form and s_ℓ the basic block with this label. The set S represents the current state of the analysis. This set contains tuples of the form $(x\ p)$, stating that the variable x has the property p.

Table 1. Analysis notation

Notation	Description
ℓ	The label ℓ of the basic block.
s_ℓ	The basic block with the label ℓ.
S	The current analysis state.
x	The variable x of the SSA form.
p	The property p.
$(x\ p)$	The variable x has the property p.

Table 2. Property notation

Notation	Description
$x\mid$	x is loop-dependent.
$x\nmid$	x is not loop-dependent.
$x = 0$	x is zero.
$x \not\simeq 0$	x is at most once zero.
$x = 1$	x is one.
$x+$	x is positive.
$x-$	x is negative.

Table 2 introduces the set of properties a variable may have. A variable can have multiple properties associated with it. Only a few properties are mutually exclusive, e.g., a positive variable cannot be negative at the same time. Moreover, the absence of a property does not mean its negation, it only means that the property has not been proved by the conservative analysis. The primary property is the loop dependence denoted as the vertical bar[1] $x\mid$. This expresses that the variable x is either the loop index itself or a variable computed with an injective function making use of the loop index. Therefore, these variables have unique values for each iteration. This property eventually helps to prove the absence of array access intersections, along with other rules that we describe in subsequent sections. The counterpart $x\nmid$ expresses that the variable x will have the same value for all iterations. The property $x = 0$ states that the value is exactly 0,

[1] The idea for the bar comes from the moving value of the loop index. It moves from one bound to the other.

whereas $x \not\simeq 0$ specifies that the value may only be zero for a single iteration. This may be the case if the variable x is loop-dependent. $x = 1$ means that x is exactly 1. Property $x+$ denotes that the value of x is positive or increases in each loop step, if x is loop-dependent. Analogously, $x-$ has the opposite characteristics as $x+$, i.e., represents negative values or a loop-decreasing value.

2.4 Property Inference

This section introduces the rules to derive property sets from expressions and statements in the program code.

Loop Index The loop index has a set of properties, comprising the loop-dependence property | and the at-most-once-zero property $\not\simeq 0$. As we currently only address increasing loop directions, the additional + property is also associated. If we support decreasing loop directions in future, we would have to associate − for them instead. As for the introductive example of Section 2.1, the loop index has the property set of (1).

$$Prop(i) = \{|, \not\simeq 0, +\} \tag{1}$$

Constants The properties derived from integer constants are self-explanatory, thus only listed in Table 3 for brevity.

Table 3. Properties generated by constants

Constant	Generated Properties
0	$\{\nmid, = 0\}$
1	$\{\nmid, \not\simeq 0, = 1, +\}$
$2, 3, 4, ...$	$\{\nmid, \not\simeq 0, +\}$
$-1, -2, -3, ...$	$\{\nmid, \not\simeq 0, -\}$

Phi-Functions The ϕ-functions conservatively yield the empty property set. As ϕ represents multiple potential values and the analysis aims to identify injective index expressions, we cannot take the intersection. For example, if $x_1 = 2$ and $x_2 = 3$, the intersection of the properties would be $\{\nmid, \not\simeq 0, +\}$. However, this information would wrongly classify an assignment like $a[i + \phi(x_1, x_2)] \leftarrow i$ as not intersecting between the iterations which is not necessarily the case. Therefore, we stick to the conservative rule of (P-PHI).

$$Prop(\phi(x_1, x_2, ..., x_n)) = \emptyset \tag{P-PHI}$$

Variables The dataflow analysis keeps track of all variables and the associated properties and stores these tuples in the analysis state S. This state is queried to retrieve all state tuples with the desired variable to get the associated properties as specified by (P-VAR).

$$Prop(x) = \{p \mid (x\ p) \in S\} \tag{P-VAR}$$

For example, consider the analysis state sketched in (2).

$$S = \{i\ |, i \not\simeq 0, i+, o_1\ \nmid, o_1 \not\simeq 0, o_1 = 1, o_1+\} \tag{2}$$

(3) and (4) illustrate the properties of the variables i and o_1, respectively.

$$Prop(i) = \{|, \not\simeq 0, +\} \tag{3}$$
$$Prop(o_1) = \{\nmid, \not\simeq 0, = 1, +\} \tag{4}$$

Binary Expressions Tables 4 to 8 list the property inference rules for binary expressions. The columns *Left* and *Right* express the conditions that have to be satisfied when deriving the properties. More specifically, both columns specify a set of properties that have to be associated with the corresponding operand as sketched in (P-BINEXP).

$$Prop(e_1\ \text{op}\ e_2) = \bigcup_{r \in Rows(\text{op})} \{Result_r \mid Left_r \subseteq Prop(e_1) \wedge Right_r \subseteq Prop(e_2)\} \tag{P-BINEXP}$$

For example, if the column *Right* has the set $\{|, +\}$, it requires that the properties $|$ and $+$ are associated with the right operand. If this condition is not satisfied, the properties of the current row cannot be derived. The column *Result* denotes which properties are derived if the associated conditions are fulfilled. The rows captioned with *Copy* identify opportunities where the properties can be safely copied from the specified operand. In summary, each row of the rule table has to be checked whether the corresponding result property of the row can be derived. Therefore, it is possible to derive multiple properties for a single binary expression that are eventually unified the property set of that binary expression.

The defined rules follow arithmetics. For example, the addition of two positive numbers results in a positive number. Similarly, a multiplication with zero results in zero. However, these rules do not respect integer overflows/underflows and may yield incorrect properties in such a case. As mentioned in Section 2.2, we presume that the sequential implementation is free of overflows, or alternatively, suggest enabling overflow runtime checks in .NET.

As a general example of how to infer the properties of a binary expression, consider the expression in (5), an extract of the statement in line 2 of the example's SSA form.

$$o_1 + i * 2 \tag{5}$$

Table 4. Inference rules for subtractions

Left	Right	Result
$\{\mid\}$	$\{\nmid\}$	$\mid$
$\{\nmid\}$	$\{\mid\}$	$\mid$
$\{\mid, +\}$	$\{-\}$	$\mid$
$\{-\}$	$\{\mid, +\}$	$\mid$
$\{\mid, -\}$	$\{+\}$	$\mid$
$\{+\}$	$\{\mid, -\}$	$\mid$
$\{\nmid\}$	$\{\nmid\}$	$\nmid$
$\{+\}$	$\{-\}$	$+$
$\{-\}$	$\{+\}$	$-$
$\{+\}$	$\{-\}$	$\not\simeq 0$
$\{-\}$	$\{+\}$	$\not\simeq 0$
Covered by copy rule.		$= 0$
Covered by copy rule.		$= 1$
$\emptyset$	$\{= 0\}$	Copy from left

Table 5. Inference rules for additions

Left	Right	Result
$\{\mid\}$	$\{\nmid\}$	$\mid$
$\{\nmid\}$	$\{\mid\}$	$\mid$
$\{\mid, +\}$	$\{+\}$	$\mid$
$\{+\}$	$\{\mid, +\}$	$\mid$
$\{\mid, -\}$	$\{-\}$	$\mid$
$\{-\}$	$\{\mid, -\}$	$\mid$
$\{\nmid\}$	$\{\nmid\}$	$\nmid$
$\{+\}$	$\{+\}$	$+$
$\{-\}$	$\{-\}$	$-$
$\{+\}$	$\{+\}$	$\not\simeq 0$
$\{-\}$	$\{-\}$	$\not\simeq 0$
Covered by copy rules.		$= 0$
Covered by copy rules.		$= 1$
$\emptyset$	$\{= 0\}$	Copy from left
$\{= 0\}$	$\emptyset$	Copy from right

Table 6. Inference rules for multiplications

Left	Right	Result
$\{\mid\}$	$\{\not\simeq 0\}$	$\mid$
$\{\not\simeq 0\}$	$\{\mid\}$	$\mid$
$\{\nmid\}$	$\{\nmid\}$	$\nmid$
$\{+\}$	$\{+\}$	$+$
$\{-\}$	$\{-\}$	$+$
$\{+\}$	$\{-\}$	$-$
$\{-\}$	$\{+\}$	$-$
$\{\not\simeq 0\}$	$\{\not\simeq 0\}$	$\not\simeq 0$
$\{= 0\}$	$\emptyset$	$= 0$
$\emptyset$	$\{= 0\}$	$= 0$
Covered by copy rules.		$= 1$
$\emptyset$	$\{= 1\}$	Copy from left
$\{= 1\}$	$\emptyset$	Copy from right

Table 7. Inference rules for divisions

Left	Right	Result
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 8. Inference rules for modulo

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

(6) to (8) show the properties of the operands.

$$Prop(o_1) = \{\nmid, \not\simeq 0, = 1, +\} \tag{6}$$
$$Prop(i) = \{|, \not\simeq 0, +\} \tag{7}$$
$$Prop(2) = \{\nmid, \not\simeq 0, +\} \tag{8}$$

Since the variable i is the loop index, it features the loop-dependence property $|$. The application of the rules within nested expressions follows conventional evaluation rules of the programming language. In this example, the multiplication is evaluated before the addition, i.e., the corresponding expression tree is evaluated bottom-up. As a consequence of the evaluation order, the multiplication rules are applied first. Table 6 denotes the necessary inference rules for multiplications. The $|$ property is derived because the left operand contains the property $\{|\}$, and the right contains $\{\not\simeq 0\}$. Moreover, $+$ and $\not\simeq 0$ are derived by the fact that both operands contain $\{+\}$ and $\{\not\simeq 0\}$. This finally leads to the properties of (9).

$$Prop(i * 2) = \{|, \not\simeq 0, +\} \tag{9}$$

Subsequently, the rules for additions listed in Table 5 have to be applied. Since the right operand has the properties $\{|, +\}$ and the left has an associated $\{+\}$, the $|$ property is obtained. Moreover, the properties $+$ and $\not\simeq 0$ are derived because the $\{+\}$ condition for both operands is satisfied. (10) denotes the resulting properties after the application of the addition rule.

$$Prop(o_1 + i * 2) = \{|, \not\simeq 0, +\} \tag{10}$$

Other Operations As a conservative approach, expressions without rules generate the empty property set. For example, floating point operations are not supported and yield no properties. Since our analysis is inter-procedural, the use of method invocations is however supported and propagates properties.

2.5 Dataflow Analysis

The previous sections described how to infer properties for a given expression or statement. This section now describes the application of these properties in the dataflow analysis. The properties have to be computed per basic block. Anything but variable assignments does not generate new properties; thus, the corresponding *Gen* functions are equal to the empty set as (11) to (14) show.

$$Gen(x[e_1] \leftarrow e_2) = \emptyset \tag{11}$$
$$Gen(\text{label } n) = \emptyset \tag{12}$$
$$Gen(\text{goto } n) = \emptyset \tag{13}$$
$$Gen(\text{if } e \text{ goto } n) = \emptyset \tag{14}$$

Assignments combine the properties retrieved from the expression e with the variable x to form the state tuples as illustrated in (G-ASGMT).

$$Gen(x \leftarrow e) = \{(x\ p) \mid p \in Prop(e)\} \tag{G-ASGMT}$$

The dataflow transfer function in (S-TRANS) eventually merges the current analysis state with the generated states of the statement s_ℓ by applying a unification.

$$Transfer(s_\ell) = Gen(s_\ell) \cup S \qquad \text{(S-TRANS)}$$

One way to solve the analysis is with the help of an iterative worklist algorithm [15, 23, 24]. The worklist can be initialized with all basic blocks of the SSA form. Although the initial order of the statements is irrelevant, a top-down initialization requires fewer iterations to converge. The algorithm will terminate in any case because of the limited size of the monotonously growing analysis state. This is because the transfer function introduced in (S-TRANS) does not discard any information, the number of state tuples may only increase. At most, it will have the size of the number of variables times the number of properties.

2.6 Array Intersection Analysis

Before answering the general question of the loop independency, we first have to determine potential array intersections. The output of the preceding dataflow analysis provides corresponding information, namely whether array index expressions are disjoint across all iteration steps of a loop. We additionally need an alias analysis [23] to identify potential aliases of an array, since this may be additional source for array access intersections. To deal with array aliases, we conservatively treat potential array aliases as if they would refer to the same array instance.

For each array and its potential aliases, we collect all array element access expressions. In the introduced example, this are the read from $f[p_1]$ in line 3 and the write to $f[p_1]$ in line 14. We then concentrate on the index expressions of the array accesses: If an array has accesses with different index expressions, we conservatively consider these accesses as potentially intersecting. We later relax this condition in Section 2.8. If the same index expression is used in all accesses of an array, we continue the intersection analysis for this array: In our example, this is the case since each access to the array f is made with the same index expression p_1.

In the case of a common access index expression, we look up the property set of this expression and check whether it contains the loop dependency property. This is the case for the index expression p_1 since the state S contains the tuple p_1 |. Due to the presence of the loop dependency property, it is ensured that each iteration computes a distinct index and thus no access intersections happen on the array f.

2.7 Loop Dependency Analysis

If a for loop contains I/O operations, potentially side-effecting calls, or writes to variables (not array elements!) declared outside the loop body, they are likely to be loop-dependent and are immediately rejected as parallelization candidates. Otherwise, we examine array accesses in the loops: We check for each array (and its aliases), whether it is read-only accessed or the array accesses do not intersect.

The latter information is obtained from the preceding array intersection analysis. If all these checks are fulfilled, the loop is guaranteed to be independent and can be parallelized.

The factorial example has obviously no array access intersections. Moreover, the loop performs neither I/O calls, nor side effects, nor writes to variables declared outside the loop body. Consequently, the loop is parallelizable.

2.8 Accuracy Improvements

A variety of optimization techniques can improve the results of the introduced analysis. For example, consider the situation sketched in Listing 6 that constructs the array indices with an offset. The lines 4 and 5 both access the same array

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

Listing 6. Indices as common sub-expressions

```
1  var m = a.Length - 1;
2  var b = ...; // computed
3  for(var i=0; i < m; i++) {
4    if(b) {
5      a[i] = a[i] * 2;
6    } else {
7      var x = i + 1;
8      a[x] = a[x] * 3;
9    }
10 }
```

Listing 7. Intersecting array access with branches

element. However, 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. To cover such situations, we implement a common sub-expression elimination [23] together with a copy propagation [23] prior to the loop dependency analysis.

Another situation in which we miss the parallelization opportunity is sketched in Listing 7. In this example, two different branches are present. Each branch uses a different function to compute the array indices. One single branch would be detected as parallelizable but not their combination. However, since the condition and its result is the same for every iteration, the loop itself is parallelizable. One approach to aid the analysis to handle such situations is the application of program slicing [6, 11, 29] and analyzing each slice separately. Although, our tool does currently not implement this.

3 Implementation

The refactoring tool[2] is a static code analyzer implemented in C# with the help of the .NET Compiler Platform (Roslyn) [19]. It is a Visual Studio 2017 Version 15.6 [21] plugin that collects all for-loops within the source code and reports opportunities for parallelization. Our array access intersection analysis is based on interprocedural context-insensitive dataflow analysis as described in the previous section. To avoid cases where methods are overridden with polymorphism, our inter-procedural analysis only follows non-virtual methods.

Internally, the C# code is transformed into a three-address intermediate representation. During the conversion process, semantic checks prevent ambiguities of shadowed variables. Moreover, a copy propagation and common subexpression elimination are used to improve the precision of the array access analysis. We also implemented an alias analysis to detect potential array aliases. Although an SSA form is the best baseline for our analysis method, we did not implement SSA transformation for performance reasons (short refactoring tool reaction time) but used an over-approximation, namely reaching definition analysis instead. This simplification does not violate soundness of array access intersection analysis.

The implementation currently 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. The unlisted language features are not implemented at this time and conservatively prevent parallelization of the for-loop.

Moreover, we support a small set of white-listed methods of the .NET framework that are side-effect-free and can be safely parallelized, such as `System.Math.Abs`. Our analysis currently analyzes each source code file in isolation and does not perform an inter-document full-solution analysis.

4 Experimental Evaluation

The refactoring tool was applied to a selection of open source projects available on GitHub. The list comprises well-known C# projects in general as well as projects by search strings such as image, algorithms, machine learning, and database are included. A project must have had at least five for-loops to be taken into account. Different software maturity levels are considered by selecting the projects with different user rankings (measured as the number of stars awarded by users in GitHub): from 56 up to 7,063. It is important to note that test code of the selected projects was excluded from the analysis to only consider production code in the evaluation.

Each scan was run on a computer with an Intel Core i7-7700HQ CPU with four 2.80 GHz cores and 16 GB RAM. The source code of the projects was located on an SSD (SK hynix SC308). On average, a scan of a project took 4.59

[2] The source code is available on `https://github.com/camrein/RefactorToParallel`

seconds. The measured times are the average out of three subsequent runs. The analysis automatically skips files without for loops hence the scan time does not necessarily correlate with the lines of code. Moreover, large files require more time because the analysis eagerly collects and optimizes methods for the interprocedural analysis even if they are not used by the for-loops.

During this evaluation, a for-loop was considered parallelizable if it can be replaced with its `Parallel.For` counterpart without breaking the as-if-serial property as well as without changing its body, i.e., not introducing synchronization in the loop body of the parallel counterpart. We focus on a specific type of for-loops where we apply array intersection analysis, so-called array-centric loops. These are for-loops with their bodies accessing arrays and engaging the elementary language features as listed in Section 3. We, therefore, exclude for-loops that work on collections, provoke virtual method calls, or use object accesses (except arrays and strings). Our motivation is to assess the quality of the array intersection analysis, by concentrating on mere array-accessing for-loops with language features implemented by our analysis. Twenty projects were analyzed, as listed in Table 9.

Table 9. Projects used for the analysis coverage evaluation

Project	LoC (C#)	Stars	Avg. Time (s)
AForge.NET	20,684	405	4.04
BrainSimulator	189,858	224	5.34
CSCore	45,325	860	4.75
C-Sharp-Algorithms	16,921	1,657	2.52
GeneticSharp	13,581	320	1.74
ImageProcessor	18,974	1,859	4.27
ImageSharp	27,966	1,790	2.13
Inbox2 desktop client	88,731	435	5.07
LiteDB	15,404	2,806	1.99
Microsoft BotBuilder	41,998	5,037	4.08
Naiad	28,886	371	4.59
Nancy	34,874	6,034	4.57
NHibernate	360,776	1,423	8.08
Popcorn	18,971	601	2.30
RavenDB	638,467	1,898	13.25
SharpBrain	1,330	53	2.11
SignalR	16,941	7,063	2.89
Spring.NET	52,208	456	9.18
Structure.Sketching	17,536	56	2.48
Veldrid	17,496	361	2.02

We scanned all projects for the described type of for-loops and counted 223 such array-centric for-loops. Ten of these loops make use of a computed array index and the other 183 use the loop index itself. We eventually run our loop

independence analysis on all these projects. The tool proves to be relatively fast, taking about 4.5 seconds on average per project. Our tool reported 193 of the selected 223 for-loops to be independent and parallelizable. 30 of 223 for-loops could not be identified as independent and would therefore not be parallelized. We eventually manually reviewed all the 223 loops for loop independence. As expected, all 193 reported parallelizable loops were indeed independent (no false positives). As for the 30 for-loops that the tool did not approve for parallelization, we identified six loops to be independent, i.e., 3% false negatives). To support these remaining 3%, we observed that an analysis supporting accesses to array ranges — such as the GCD dependence test — would be necessary. In summary, our tool shows effective detection for the case of "low-level" array-centric loops.

5 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.

Hydra [8] operates on the intermediate representation of LLVM [7, 16, 17]. Its underlying idea can be seen as C#'s async/await [18, 22] programming model. However, instead of manually declaring fragments with `async` and `await` respectively, it infers them automatically. To accomplish this, Hydra searches for parallelizable code fragments and computes an estimated cost of the sequential and the parallel code. If the estimated cost of the parallel code is lower than the original sequential, the function invocation is offloaded to a thread pool.

Sambamba [26, 27] aims at large-scoped automatic parallelization. Besides a conservative compile-time code analysis on LLVM's bitcode, it also incorporates a runtime system that optimistically parallelizes execution. For this purpose, the runtime system collects further information and replaces method code where appropriate. The collected information is thereby stored in a persistent storage that can be reused in later program executions. At runtime, Sambamba involves a software transactional memory [14] system to guard optimistically parallelized code. This STM system allows, different to the previous two approaches, the parallelization of code fragments that require synchronization.

Baar [3, 9, 10] is an approach that transparently offloads compute-intense program parts to a server. It provides an environment to allow the execution of programs in LLVM's IR. The goal is the runtime identification of code hotspots. Identified hotspots are offloaded to a server that applies additional processing such as parallelization and vectorization. The automatic parallelization is thereby accomplished with the LLVM subproject Polly [13, 25], based on the polyhedral model. The Message Passing Interface (MPI) [12] is used for inter-process communication; reducing the negative impact of expensive data transfers between client and server.

In summary, none of the introduced projects actively informs the user about the parallelization opportunities. In contrast, our implementation aids the developer with the parallelization by actively informing them inside the IDE, through-

out the entire source code. The developer may then decide if the parallelization is justified and apply the refactoring directly in the source code.

Besides the already introduced tools, there are also various analysis approaches. For example, the GCD dependence test [4, 23, 28] is a technique that allows deciding if array indices computed with linear expressions may intersect with other iterations. Although it does not support non-linear expressions like our approach, it is capable of identifying non-intersecting ranges.

Another powerful analysis technique is the polyhedral model (or polytope model) [1, 5]. 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 which can become very expensive for complex programs.

6 Conclusion

This paper introduced a light-weight conservative dataflow analysis to prove the absence of array access intersections in for-loops, covering linear and non-linear array element access patterns. It proves to be effective in terms of speed and accuracy for "low-level" for-loops that merely work on arrays. For this frequent case of for-loops, we achieved 97% precision with 4.5 second analysis time per multi KLOC project. We integrated the analysis in a practical Visual Studio IDE plugin operating on C# code. The tool actively informs the user about safe parallelization opportunities in the context of array-centric for-loops.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools, 2/e. Addison Wesley Boston (2003)
2. Akhter, S., Roberts, J.: Multi-core programming, vol. 33. Intel press Hillsboro (2006)
3. Baar (binary acceleration at runtime). `https://github.com/pc2/baar`, accessed: 2019-06-03
4. Banerjee, U.: Dependence testing in ordinary programs. Master's thesis, University of Illinois, Department of Computer Science (1976)
5. Bondhugula, U.K.: Effective automatic parallelization and locality optimization using the polyhedral model. Ph.D. thesis, The Ohio State University (2008)
6. Canfora, G., Cimitile, A., De Lucia, A.: Conditioned program slicing. Information and Software Technology **40**(11-12), 595–607 (1998)
7. Chansler, R., Bryant, R., Bryant, R., Canino-Koening, R., Cesarini, F., Allman, E., Bostic, K., Brown, T.: The architecture of open source applications (2011)
8. Chicken, J.: Hydra: Automatic parallelism using llvm (2014), homerton College
9. Damschen, M., Plessl, C.: Easy-to-use on-the-fly binary program acceleration on many-cores. arXiv preprint arXiv:1412.3906 (2014)
10. Damschen, M., Riebler, H., Vaz, G., Plessl, C.: Transparent offloading of computational hotspots from binary code to xeon phi. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015. pp. 1078–1083. IEEE (2015)

11. Danicic, S., Fox, C., Harman, M., Hierons, R.M.: Consit: A conditioned program slicer. In: IEEE International Conference on Software Maintenance (ICSM'00). pp. 216–226 (2000)
12. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: portable parallel programming with the message-passing interface, vol. 1. MIT press (1999)
13. Grosser, T., Zheng, H., Aloor, R., Simbürger, A., Größlinger, A., Pouchet, L.N.: Polly-polyhedral optimization in llvm. In: Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT). vol. 2011 (2011)
14. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann (2011)
15. Khedker, U., Sanyal, A., Sathe, B.: Data flow analysis: theory and practice. CRC Press (2009)
16. Lattner, C., Adve, V.: 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. pp. 75–. CGO '04, IEEE Computer Society, Washington, DC, USA (2004)
17. The llvm compiler infrastructure. `http://llvm.org`, accessed: 2019-06-03
18. Microsoft: Asynchronous programming with async and await (c# and visual basic). `https://msdn.microsoft.com/library/hh191443(vs.110).aspx`, accessed: 2019-06-03
19. Microsoft: .net compiler platform (roslyn). `https://github.com/dotnet/roslyn`, accessed: 2019-06-03
20. Microsoft: Task parallel library (tpl). `https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl`, accessed: 2019-06-03
21. Microsoft: Visual studio. `https://www.visualstudio.com`, accessed: 2019-06-03
22. Microsoft: C# language specification version 5.0. `https://www.microsoft.com/en-us/download/details.aspx?id=7029` (2015), accessed: 2019-06-03
23. Muchnick, S.S.: Advanced compiler design implementation. Morgan Kaufmann (1997)
24. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (2015)
25. Polly - llvm framework for high-level loop and data-locality optimizations. `https://polly.llvm.org`, accessed: 2019-06-03
26. Streit, K., Hammacher, C., Zeller, A., Hack, S.: Sambamba - adaptive optimization and parallelization of general purpose programs. `http://www.sambamba.org`, accessed: 2019-06-03
27. Streit, K., Hammacher, C., Zeller, A., Hack, S.: Sambamba: Runtime adaptive parallel execution. In: Proceedings of the 3rd International Workshop on Adaptive Self-Tuning Computing Systems. pp. 7:1–7:6. ADAPT '13, ACM, New York, NY, USA (2013)
28. Towle, R.A.: Control and Data Dependence for Program Transformations. Ph.D. thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA (1976)
29. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering. pp. 439–449. ICSE '81, IEEE Press, Piscataway, NJ, USA (1981)