Finding Lightweight Opportunities for Parallelism in .NET C#

Version 1.0

13 January 2014

By Richard Bos, Master's degree student at University of Amsterdam

Contact: Mail@RichardBos.net

Supervised by Jurgen Vinju, CWI.

Contact: Jurgen.Vinju@cwi.nl

Abstract

Adding parallelism to .NET C# software programs has become a great option that can be used to enable performance. But how can we find out if the existing large complex software programs are even suitable for parallelism, other than investing a lot of time by checking it by hand? By using static analysis to find dependencies in the source code of software programs, we are able to find actual opportunities for parallelism. This list of generated opportunities for parallelism provides information needed to make the decision whether it is worth the time and effort to implement parallelism to a software program, and also provides guidance for the programmers when parallelism is implemented.

Acknowledgements

First of all I would like to thank my tutor Jurgen Vinju. I have enjoyed our collaboration and many discussions about the topic. Without this help and guidance I would not have been able to write this thesis.

I would also like to thank my fellow students of the Software Engineering master at the University of Amsterdam. We have shared many project hours, classes, laughs and beers.

My thanks for making it possible for me to do this master, goes out to my employer, Herke ICT Group, my supervisor, Peter Overmeer, and my colleague's for supporting me and conversing with me about my topics.

Finally I would like to thank my girlfriend, my family and my friends for their mental support during this master.

Contents

1. Introduction

During the last decade, CPU's have stopped increasing in CPU frequency. Instead, CPU's are now equipped with more and more cores. To take full advantage of these many cores and to speed up the software, there has to be support built into software programs to consume them. Unfortunately, manual parallelization of software is time-consuming and error-prone for all but the most advanced programmers. In this study, we will try to assist programmers that want to add parallelization to their software.

1.1 The Problem

The problem we are trying to solve with this project, is that there is no quick way to determine if a non-parallel .NET C# program could be transformed to use parallelism. Experience shows that many programs in C# are not developed to use parallelism. Currently there are not many ways to check if a program could be transformed to use parallelism other than doing it manually, (another) time consuming and error-prone task that not all companies can afford to do.

1.1 Parallelism

To elaborate why adding parallelism to software is good, consider the following. A car has to be assembled from parts and everything needed to assemble the car is present. Imagine if there was one worker assembling the car. It would take the worker an certain amount of time. If there were 2 workers, they could both be working on the car at the same time. Implying the 2 workers could finish the job twice as fast, since they can both assemble different parts of the car. However, if the car would need new paint, all other assembling has to stop and wait. This means that one of the workers will be waiting for the paint job to complete. Another thing that comes into play when 2 workers are used is some management on what the workers will be doing, so they will not interfere with each other. The point of this story is that with more workers, the car can be finished faster. However, it is likely that sometimes workers have to wait for each other. And there is also an overhead(management) when more than one worker is used. This can be translated into software and CPU's. In software there are a lot of tasks that have to be done by the CPU's. If the software only lets one CPU do the tasks, it would take longer than when it uses 2 CPU's(but not twice as fast!).

Parallelism can be defined as follows:

"Parallelism is executing multiple operations at the same time."

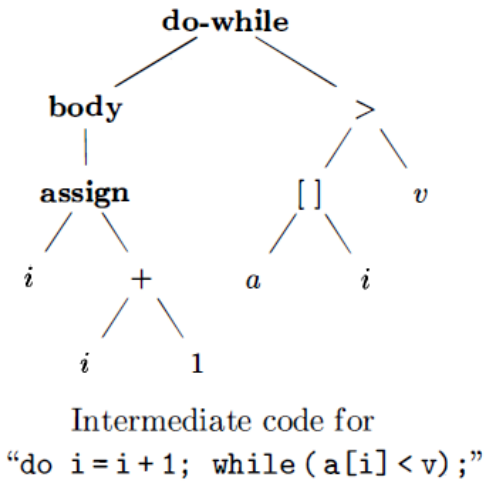

Figure 1: An example of an AST[24]

The operations in this definition can origin from the same program or from multiple programs. Finding if it is possible to achieve parallelism, and guiding programmers to implementing parallelism, is the goal of this study. In .NET C#, this is done by creating threads[1] in the code and running these threads at the same time. A thread can be seen as an execution queue, the software programs adds operations to the various threads that it uses. The operating system then divides these queued operations across the various CPU's. The biggest advantage of parallelism is that the program can be executed in less time. The biggest pitfall of parallelism is corruption of shared memory by usage of the parallel threads, causing incorrect behavior. This makes a program unreliable and therefore unusable.

Threads

There are various ways to achieve parallelism in a software program. Message Passing Interface (MPI) [3] is an example of another implementation of parallelism. In this study, we will be focusing on parallelism that will be implemented by using threads.

1.2 Dependency

Before we go into what dependencies are, we first need to understand a few other topics. We will use Figure 1 to explain these topics.

AST - Abstract Syntax tree[24]

Figure 1 is an example of an AST. An AST is a way to represent a part(or all) of the source code of a program. In an AST every element is shown as a **child** of another element, the **parent**. Every AST has a single root. The root element is the only element that does not have a parent. The root of the abstract syntax tree in Figure 1 represents an entire do-while loop. As we can see, the root element 'do-while' is

[1] An extended description is found here: http://www.microsoft.com/en-us/download/details.aspx?id=7029

parent of 2 other elements. The left child of the root represents the body of the loop, and the right child of the root represents the condition.

Statements

Statements in the source code can be compared to the sentences of a book. The statements in the source code define what the source code does when what is executed. A statement can contain other statements, we can see an example of this in Figure 1. The root of the AST is an do-while statement. The do-while statement contains another statement (‘`do i=i+1;`’) in its body (the left child).

Dependencies

Now that we know about ASTs and statements, we can go into dependencies. In any programming language there are a lot of dependencies between the statements in the source code of any given program. Software programs contain various processes, and performing any kind of process means doing it from start to end. Starting somewhere in the middle would mean the process did not perform the actions from the beginning, and could mean that the middle can’t be performed at all because these actions are required.

This translates into dependencies. A dependency in the source code of a software program is a relationship between 2 statements, in which it is required for one of the statements to be executed before the other statement can execute in order to guarantee the workings of the program. This means all of the dependencies of a given statement have to be executed before the statement can be executed itself. If all statements and their dependencies are taken into consideration, this can be seen as a mandatory sequence of execution of the source code statements.

There are different types of dependencies;

- **Call Dependency[1]**
  A Call Dependency represents the relationships between **routines**. A routine can be either a method, a function or a constructor. This relationship is formed when a routine is invoked inside the body of another routine. For example, if routine A has an invocation of routine B inside it’s body, routine A has a call dependency to routine B.
- **Data Dependency**
  Data dependency is about the **values** stored in variables in the program. Data dependencies can be further divided in the following types[6]:
  - Anti-dependency, occurs when Statement A reads a memory location that Statement B later writes.
  - Output dependency, occurs when Statement A writes a memory location that Statement B later writes.
  - Input dependency, occurs when Statement A reads a memory location that Statement B later reads.
  - True (flow) dependence, occurs when Statement A writes a memory location that Statement B later reads.

```
S1    var a = 1;
S2    var b = 1;
S3    if (a == 1)
S4    {
S5        b += 2;
S6    }
S7    b = b / 2;
```

Code block 1

For these descriptions of the types of data dependencies, we presume the use of a single thread and no other statements interfere with the memory locations.

Anti-dependence, Output dependence, and True (flow) dependence are the types of data dependence that we will keep track off. If two statements that share a dependency of either one of these dependency types are executed in parallel it could break the workings of a program.

The only dependency type that would not break the workings of a program in this situation is Input dependence, and because of this will not be relevant for this study.

- **Control Dependency**

  A Control dependency is about the **structure** of a program, and occurs if statements depend on the outcome of other statements whether they will be executed or not. This will be elaborated in the following example.

For an example of Data and Control dependencies, consider code block 1. It starts with 2 variable declarations, followed by an if-statement. The 2 variable declarations have no dependency, they could be placed anywhere in the code and they would still do the same, declare a variable and initialize it with a static value. However, the if-statement has a condition in which it uses variable 'a'. In order to execute the if-statement, variable 'a' has to be declared. In other words, the if-statement **depends on** variable declaration 'a', this is a **Data Dependency**. In the body of the if-statement, there is an assignment-statement of variable 'b'. In order to execute the assignment, variable 'b' has to be declared, the assignment-statement **depends on** the declaration of 'b'. The if-statement can only be executed if all of the dependencies of the statements in the body of the if-statement have been executed. In other words, the if-statement **depends on** its body statements. So the if-statement depends on the assignment statement 'b += 2', which is in turn depended on the declaration of 'b'. This means the if-statement is **indirectly dependent on** variable declaration 'b'. The assignment statement 'b += 2' will only be executed if the condition of the if-statement holds. This means that it **depends on** the if-statement, this is a **Control Dependency**. Another example of the control dependency is statement '`b = b / 2;`',the right-hand side of this statement depends on the value of 'b'. The value of 'b' depends on both the declaration statement '`var b = 1;`'(**Data Dependency**) and the if-statement(**Control Dependency**), because 'b' could be assigned in the body of the if-statement.

1.3 Opportunities for parallelism

During this study we will be trying to assist programmers who want to add parallelism to their software. We will try to do this by creating a way for programmers to know if and where programs can be parallelized. By doing this a programmer can save the time spend into looking where a program could be parallelized, or even if a program can be parallelized at all. In other words, we will be looking for **opportunities for parallelism**. These opportunities will tell if and where a given program can be parallelized. It is important to note that we will not try to change the source code to eliminate any dependencies. We will try to find the opportunities in the source code as it is given, without any transformation of any kind. We define an opportunity as follows:

"An opportunity for parallelism are two or more blocks of source code from a program that can be run in parallel without changing the output of the program."

In this definition, a block of code is a number of statements that have the same parent. A parent can be any kind of routine, property(getter or setter), if-statement, switch statement etc., all of these have child statements. For example in code block 1, statement 5 is a child statement of the if-statement. The statements of both blocks of code have to have the same parent. This is because we know that these statements would normally be executed in sequence. However, this does not limit the statements that could be run in parallel to only the statements with the same parent. For example, it can be possible for the blocks of code to contain invocation statements to other routines or to contain statements which have child statements. So this would mean that all of the statements in the called routine would also be run in parallel.

The parallelization of the blocks may not change the output of the program. If they would change the output of the program, the program could become unreliable and therefore unusable. In order to ensure this we have to make sure there are no dependencies between both blocks of source code.

1.4 Goals and structure of the thesis

This study presents a technique to find opportunities for parallelism in .NET C# software programs. But before we go into that, we will first discuss the background information in Chapter 2, handling topics like pitfalls and existing solutions. After that we will work towards our solution by achieving the following goals. Our main goal and solution to the problem is:

- Generate a list of valid opportunities for parallelism, that could improve performance, to guide programmers while implementing parallelism.

To achieve our main goal, we will need to achieve the following sub-goals:

- Extract a dependency model from .NET C# code (Chapter 3).
- Analyze the dependencies and determine which elements in the source code are independent (Chapter 3).
- Generate a list of opportunities for parallelism based on the independent elements (Chapter 3).

- Validate if the found opportunities can improve performance (Chapter 4).
- Validate that the found opportunities do not change the output of the program (Chapter 4).

In the final chapters we will draw a conclusion and give thought for further research on this topic.

2. Background

2.1 Pitfalls of parallelism

Adding parallelism to a program can also introduce problems. We will have to try to avoid the biggest pitfalls with our solution.

2.1.1 Shared Memory Problems

When sequential code is transformed into parallel code, some memory locations could be used in more than one simultaneous thread and thus be shared between them. Because it is unpredictable and inconsistent which thread will run first, the value of the variable at any given state is also unpredictable and inconsistent. This can cause data races and race conditions, which are explained in the section below.

```
S1 bool Transfer (int amount, Account
        accountFrom, Account accountTo){
S2.  if (accountFrom.Balance<amount)
S3.     return false;
S4.  accountTo.Balance += amount;
S2.  accountFrom.Balance -= amount;
S6.  return true;
S7.}
```

Code block 2

Data races and Race Conditions

A data race can occur when multiple threads try to access a shared variable at the same time. There are three cases in which a data race can occur:

1. When a thread writes to the variable while another thread reads it.
2. When more than one thread writes to the variable.
3. When more than one thread reads the same variable.

In the first case, the result can be two different states. The thread that is reading variable can either get the value from before the write of the second thread or from after the write. The result depends on which thread accesses the variable first.

In the second case, the result can again be two different states (assuming there are two threads). One of the threads will be the last to perform the write action. The result will be that the value of the variable will have the value of this action.

For example, consider code block 2. If S4 would be executed twice and in parallel, both executions could first read the value of accountTo.Balance; perform the addition operation; and finally write the new value to accountTo.Balance. In this example, the execution that would happen to perform its write

operation last will define the value of accountTo.Balance, and also overwrite the result of the other execution. This means the resulting values on '`accountTo.Balance`' and '`accountFrom.Balance`' are unpredictable and can be different every time.

In the third case there is no danger of corrupting the variable. It does not matter which of either threads reads the variable first, the result will be the same.

Race Conditions can occur when a variable is shared between threads. Race Conditions occur when a variable is changed between a 'check' and an 'act'. For example, in code block 2, S2 would be the check, and S3 or S4, S5 and S6 would be the acts. If the value of '`accountFrom.balance`' is changed after the check is executed, a corrupted state can be the result because the following statements can assume that the check (still) holds. In this case, the value of '`accountFrom.Balance`' can become negative.

Because we want to avoid this pitfall, we have to get insight on where the dependencies are in the code. If the generated opportunities for parallelism do not compromise the dependencies, data races and race conditions will not occur if they would be implemented correctly.

2.1.2 Over-parallelization

By introducing parallelism, you create an overhead to manage the threads. This could actually make your application slower than without parallelism. It is recommended that you only parallelize expensive operations, and make sure that the target machine contains more than one processor.

Another look at this pitfall is described in Amdahl's law[2], the maximum performance increase of a program when implementing parallelism is limited by the time needed for the sequential fraction of the program. In other words, there are always parts of a program that cannot be executed in parallel. Examples of these parts are:

- Introduced overhead by adding parallelism
- Code inside a lock
- Code with too many dependencies

Improving the performance of the program by implementing parallelism cannot be increased past the time needed to execute these sequential parts.

2.1.3 Cross-thread operations

Some parts of .NET C# can only be accessed from the thread in which they were created. For example this holds for all form-controls in Windows Forms applications. When parallelism is added and these controls are accessed from a new thread, runtime exceptions can be generated. This pitfall can be seen as a specific case of a Shared Memory Problem.

2.1.4 Deadlocks

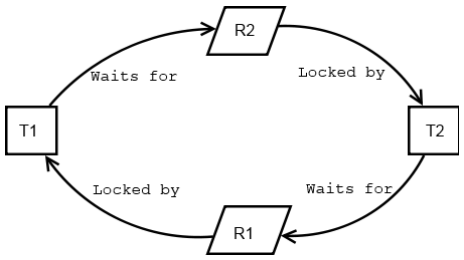

Figure 2: Deadlock
T1 & T2 are threads,
R1 & R2 are resources.

To enable parallelism, locks can be used. A lock can be made on a resource by a thread to make sure other threads will not access it. A Deadlock is a state in a parallel program involving at least 2 threads and 2 resources. Both of the threads have locked one of the resources, and are waiting on the other resource to be freed from lock.

Consider Figure 2, T1 has a lock on R1 and is waiting on R2 to be freed from lock, which is locked by T2 and is waiting on R1 to be freed from lock. It can also involve more threads and resources, the threads are then waiting in a circular pattern. Deadlocks do not only arise in parallel programs, deadlocks can also arise in the real world. For example, in traffic.

Because we will not be implementing or give options on how to implement the found opportunities for parallelism, we will not look into the possibility that the opportunities for parallelism can introduce deadlocks. This study will show the programmers where parallelism can be implemented, but not how. However, to implement the found opportunities for parallelism locks will not be needed. The blocks of code from the opportunities will be completely independent from each other. In other words, because locks are not required, implementing the found opportunities will not cause deadlocks.

2.2 Existing Solutions

This chapter will go into past research which relates to this study. First we will discuss the various research topics on dependencies and parallelism. Then we will discuss how they relate to this study.

2.2.1 Dependencies

This section contains past research on dependency analysis.

Dependencies in loops

Banerjee et al. [4] provide an overview paper on automatic program parallelization techniques. They also cover dependence analysis techniques, straight line code parallelization, do loop transformations, and parallelization of recursive routines.

Dependency analysis types

There are two kinds of dependency analysis: Dynamic and Static. Dynamic dependency analysis is done by executing the code and analyzing how the code is run. Static dependency analysis is done by analyzing the code without executing it. Ronen et al. [18] show research on the combination of Dynamic and Static dependency analysis.

Program Dependence Graph

We have identified 3 types of dependencies: Call, Control and Data dependencies. J. Ferrante et al. [23] have shown that Data and Control Dependencies can be combined to create a Program Dependence Graph(PDG). They describe the PDG as follows: "The program dependence graph explicitly represents both the essential data relationships, as present in the data dependence graph, and the essential control relationships, without the unnecessary sequencing present in the control flow graph.". They show that the PDG can be used to perform transformations on the code. For instance, constant expression folding can be applied using the PDG. Constant expression folding is a technique in which an operator having a constant result can be replaced with a constant value. This could help reducing dependencies in a program and possibly creating more opportunities for parallelism.

Tools to extract dependencies

There are a few tools already developed that can extract a number of dependencies from programs. An example of such a tools is NDepend[2]. NDepend can calculate many code metrics for .NET programs, and generate dependency graphs and dependency matrixes. Unfortunately NDepend, like most other tools, is not open source.

Program slices

Program slices, coined by Mark Weiser[19], have a lot in common with our goal. A program slice can be defined as follows: A program slice is the computation of the set of programs statements, the program slice, that may affect the values at some point of interest, referred to as a slicing criterion.

An example of a program slicer is shown by Ricky E. Sward and A.T. Chamillard[7]. Program slices should be able to run without complications. This means that the program slices are independent of all other statements in the program. So in order to obtain the slices, the dependencies have to be known.

Compiler

Eva Burrows and Magne Haveraaen[5] illustrate that automatic dependency analysis is proven to be too complex for the general case, and that a parallelizing compiler cannot help parallelizing every dependency pattern. Instead they show a framework from which data dependency information and placement information is created, which could be utilized by a compiler to create parallel code. This placement information has to be provided by the programmers to indicate to the compiler which parts of the program could be made parallel. An parallelizing compiler can than process this and generate the parallel code.

[2] http://www.ndepend.com/

2.2.2 Parallelization

This section goes into past research on parallelization.

Eliminate dependencies

Eliminating dependencies can be seen at part of parallelization. If dependencies can be eliminated in a program, parallelization can become easier. An example of this is shown by Morita et al. [8] for divide-and-conquer programs. In divide and conquer programs there are a lot of recursive functions. They show that it is possible to transform the source code and eliminates the dependencies between the recursive executions of the function so that parallelism can be added.

Parallelization of loops

Parallelization of loops is probably the oldest technique of automatic parallelization, because loops have the most potential for performance gain. Goumas et al. [9] provide research on parallel code generation for tiled nested loops. They created an experimental tool that transforms code into using parallelization. Arquimedes Canedo et al. [11] introduce a technique for automatically parallelizing loops which have a data dependencies. By breaking these dependencies without changing the semantics of the loop.

Compiler parallelization

When it comes to automating ways to achieve parallelization, the most well-known (and applied) kind is probably compiler parallelization[20]. However the ways that parallelism is added is limited and might require the programmer to refactor his code[21].

Guided Manual parallelization

Parallelization can also be done manual. Although, as mentioned before, this can be a time-consuming and error-prone task. Diego Huang and J. Gregory Steffan [10] provide a middle way between automatic parallelization and manual parallelization. They suggest interaction between the compiler and the programmer, so the compiler can give feedback to the programmer while applying parallelism. This feedback would be statements that are currently blocking the parallelization of certain parts of the code.

Hardware parallelization

Another way to improve performance is called hardware parallelization. By improving the hardware to enable even more parallel execution performance can be improved. An example to achieve this is called MapReduce [12]. MapReduce is a solution for parallelization across multiple servers. MapReduce programs divide the workload of a software program and distribute it across multiple servers, and have one main process manage the results. Not every software program can be used for MapReduce, it is required that the workload can be split into many pieces so different servers can execute it in parallel. So in order to use MapReduce, the program has to be transformed to break all dependencies between the tasks in the workload.

2.2.3 Comparison with this study

Transformations

Some of the various solutions that we have just noted use code transformation to achieve parallelism. For this study we have chosen to scope down to only analyzing the given source code for opportunities for parallelism. We will not transform the source code to enable parallelism, so none of the algorithms will change. The only part that would have to be added in order to introduce parallelism for the found opportunities is the mechanisms to control the threads.

Loops

A loop executes the same piece of code multiple times, so this can have a high probability for parallelism. However, most of the program is not executed inside of a loop, and thus will not be checked. In this study we will not look for opportunities for parallelism in loops in particular. Instead, in this study we look in all of the source code and produce opportunities for parallelism that will help programmers to win performance gains. This also means we will not focus on loops in particular to be sure we have extracted all dependencies.

Dependency analysis types

In this study we can settle for the use of static dependency analysis, because .NET C# is a highly structured language. Unlike languages like JavaScript or PHP, in .NET C# we can determine the dependencies in a static environment. Unlike with dynamic analysis, all source code can be analyzed. This is because in dynamic analysis a program could be run without all of the code being executed. And if some code is not executed, dynamic analysis does not analyze all of the code.

Program Dependence Graph

Like the PDG, we will also use Data and Control dependencies to determine the opportunities for parallelism. We will have to analyze the code for Data, Control and Call dependencies to be able to know if an opportunity for parallelism will not change the output of the program.

Program Slices

Like with program slices, we have to determine the dependencies of the program. With program slicing, a criterion is chosen and from there on everything that the criterion is dependent on is added to the slice. In this study, we will be calculating the dependencies just like with program slicing, only we do not have a criterion and will calculate the dependencies of all the statements in the program. Another property of the program slices is that they are independent from each other. This means that the program slices could be run in parallel. Even though this sounds like it could be a solution for this study, there is a down side. Because the program slices contain all dependencies from the chosen criterion, some statements could be found in multiple program slices. If program slices are run in parallel but contain many of the same statements, meaning these statements are executed multiple times, performance increase could turn out to be very limited. Instead of using program slices, we will look for independent blocks of code that can be executed in parallel.

Guided Manual parallelization

The research performed by Diego Huang and J. Gregory Steffan has similarities with this study. They also give information to the programmer about where their attention should be directed to. However, in their research the aim is on compiler parallelization. And as described earlier, this can be limited. In comparison, our approach is not limited to compiler parallelization and thus has a wider scope. The downside of our approach is that the implementation of parallelization is not done automatically yet. However, this could be added in a future research.

Compiler

Compiler parallelization analyzes the code for certain patterns (e.g. loops) and try to make them parallel. In this study we look for opportunities for parallelism by analyzing all of the source code. Improvement can be made on compiler parallelization by enabling feedback from the compiler to the programmer about aspects in the code that need refactoring. This has some similarity with our study, as we will be providing opportunities for parallelism that the programmer has to process in the code.

Hardware parallelization

The aim of this study will be applications that would be run on typical servers or PC's. We will not be aiming for applications that would run on supercomputers. The difference is that on supercomputers any small parallelization could blow up into a huge gain because of the volume of CPU's and number of times the application is executed. Instead we will focus on making applications perform faster by finding high-level opportunities for parallelism which on itself could cause serious performance increase.

2.3 Extracting dependencies from source code

To find the opportunities for parallelism we need the dependencies. However there is no static list of the existing dependencies between the elements in the .NET Framework. To find the dependencies, we have to analyze the source code and keep track of the dependencies when we locate them by reasoning about the source code. Analyzing the source code will be done by using an AST. By visiting the AST, we can analyze the source code and infer the dependencies from it.

2.4 Sequential Consistency

There are several levels of consistency that can be applied when introducing parallelism. In this study, we have chosen to produce opportunities that would not change the output of the program, and try to ensure sequential consistency. Sequential consistency is defined as follows:

"The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."[16]

In sequential consistency, transformations are allowed as long as the output of the program remains the same as the original program, and the operations executed on individual processors are in the same sequence as defined in the program. The proposed solution of this study first calculates the

dependencies of all the statements in the program. Then it analyzes the dependencies for opportunities for parallelism that do not break any of these dependencies. Breaking a dependency means that a statement is executed before all of its dependencies have been executed. Breaking dependencies can cause the output of the program to change or even cause exceptions to occur. Every opportunity that does not break any dependency does not change the output of the program, and implementing the opportunity would mean that 2 blocks of code would be run in parallel without changing the sequence of the statements of those blocks. This means the proposed solution ensures sequential consistency is achieved. However, this also means that even if there is an opportunity that would not change the output of the program, but it does break a dependency, it will be excluded from the generated opportunity lists. In other words, the proposed solution is stricter than sequential consistency.

2.5 Rascal

The main part of the proposed solution is done in Rascal[22], a domain specific language. Rascal is an open source meta-programming language, it's domain is programming languages. Rascal can easily be extended to analyze new languages by introducing a language syntax for the new language. This is what was done for the .NET C# language. This syntax was then used to analyze the language for dependencies in the source code and to find opportunities for parallelism.

2.6 Dependencies in .NET C#

In this study we are interested in dependencies between all possible structures of statements in .NET C#. In this section we will sum up which dependencies have been handled in this study. We will also sum up which dependencies were found but were not handled. It is important to note that this does not mean that this is a complete set of all dependencies in .NET C#. There are probably many more undiscovered dependencies to be found. However, to give an insight on what is handled in the proposed solution, they are summed up. We will also discuss some expectations about the dependencies which were not handled.

2.6.1 Handled dependencies

Kind	Element	Depends on	Reason
Call	Routine	Routine	In the body of one routine the other is called.
Call	Class	Class	Class calls a routine of other class
Call	Object Creation Statement	Class	Constructor depends on class
Control	If-Else statement	Condition	Assignments of used variables
Control	If-Else statement	Branch Statements	Dependencies from the branch statements have to be executed first
Control	Child statement	If-Else statement	Branch statements depend on parent condition
Control	Do-While statement	Condition	Assignments of used variables
Control	Child statement	Do-While statement	Branch statements depend on parent condition
Control	While	Condition	Assignments of used variables
Control	While	Statement	Dependencies from the branch statements have to be executed first
Control	Child statement	While statement	Branch statements depend on parent condition
Control	For statement	Condition	Assignments of used variables
Control	For statement	Statement	Dependencies from the branch statements have to be executed first
Control	Child statement	For statement	Branch statements depend on parent condition
Control	For Each statement	Condition	Assignments of used variables
Control	For Each statement	Statement	Dependencies from the branch statements have to be executed first
Control	Child statement	For Each statement	Branch statements depend on parent condition
Data	Routine	Property/field	Routine uses property
Data	Property/field	Routine	Routine sets property
Data	Assignment	Last Read	Any assignment has to be executed after the last read of the variable, the order of read and writes on a variable has to remain the same.
Data	Read	Last assignment	Any read of a variable is dependent on the last assignment of that variable.
Data	Assignment	Assignment	Any assignment is depended on the assignment before it, so the order remains.

Data	Assignment statement and variable initialization	Last assignment of all right-hand side variables, or the definition of the variable (e.g. parameters or properties)	An assignment depends on the values of all variables used in the right hand side of the assignment/initialization. So this means it depends on the last assignment of these variables.
Data	Return statement	Last assignments of all variables	A return statement depends on all values of its used variables.
Data	Unary statement	Last assignment of variable	The unary statement depends on the value of its variable, and thus on the last assignment of this value.

Table 1: An overview of all found and handled dependencies.

Order of assignments

To maintain the values of variables on any given moment during execution, the order of assignments on a given variable have to remain the same. If the order is changed, the output of the program could be changed.

Optional paths

The basic data dependency could be described as follows: A statement is dependent on the last assignment of all used variables. A variation to this is found when the last assignment of a variable happens to be inside an optional path. An optional path is a block of code with a number of statements that will not always be executed. For example, the body of an If-statement. The body will only be executed if the condition is met.

This means we do not know which assignment is the last assignment. So in order to be sure the output of the program will remain the same, we have to add a dependence to both the assignment inside the optional path and the assignment before this optional path. This will continue until an assignment is found that is not inside an optional path.

However, this only holds if the usage of the variable is not in the same optional path as the last assignment. If they are in the same optional path they will always either both be executed or not.

2.6.2 Unhandled dependencies

This chapter sums up the dependencies that were found during this study but were not implemented into the solution yet.

- Object creation statement depends on one constructor

Currently, when an Object creation statement is encountered a dependency between the statement and the class is added. However, this dependency should be between the statement and the corresponding constructor instead of the class itself. This could introduce problems, for example when the constructor takes ref or out-parameters and assigns the values of these parameters.

- Await and async keywords

Await and async keywords are recently added keywords to the .NET C# language and could introduce more dependencies, or take them away. The async-keyword is used to mark a routine as asynchronous, meaning execution does not have to wait when invocating the routine. The await-keyword causes the execution to wait on an async-marked routine. This could mean that an async routine that is not awaited on does not introduce any dependencies, because it is executed in a separate thread. More research is required to be able to extract any dependencies and ensure the output of the program remains the same when await and async are used in a program.

- Delegate/anonymous/lambda functions

Currently the bodies of delegate, anonymous and lambda functions are not yet analyzed. The Dependency Extractor should be extended to analyze these bodies and keep track of the dependencies.

- Linq introduces more dependencies, for example in "Select"-bodies.

Currently, the operations that can be performed by using Linq are not analyzed. However, many dependencies could be introduces when using Linq. For example, in a "`Select`"-operation any number of statements can be performed and result in a value which is returned by the "`Select`"-`operation`.

- Struct

Support for usage of "Struct" has to be added. Because structs are a lightweight equivalent of classes, the impact of adding support for Structs is probably limited.

- Loop dependencies

Inside loops a special kind of dependency can arise, a dependency to the previous execution of the loop. This is currently not implemented in The Dependency Extractor and needs more research to be able to add this expansion.

- Throw, return and continue

Keywords like throw, return and continue should not be made parallel because these keywords can exclude other code from being executed. This means all the code after these keywords depend on them. The proposed solution does not yet handle this problem. A solution would be to prevent that any of the found opportunities for parallelism would contain return statements.

3. Architecture

In this chapter we will discuss the architecture that was created in order to obtain our goal. We will look at the requirements that the architecture has to fulfill and then we will look at how the architecture turned out.

3.1 Requirements

3.1.1 Requirement 1

If the generated opportunities would be implemented in the code, it will not change the output of the program.

Rationale

To maintain the original workings of the analyzed program, applying parallelism for the generated opportunities should not change the output of the program. This could make the program unreliable and unusable.

Challenge

This requirement is hard to achieve and hard to evaluate. To be able to be sure that the output does not change for a given found opportunity, we would have to test this opportunity. However, there is not just one test. Applying parallelism to an opportunity means there will be an astronomical amount of possible execution scenarios how the threads could be executed. To illustrate why this amount of possibilities can become astronomical, consider the following enumeration:

1. Every block of code contains multiple statements.
2. Every statement contains multiple instructions.
3. Instructions could be run:
    - In parallel with any other instruction;
    - Before every other instruction;
    - After every other instruction;
4. Every instruction can run on multiple threads.
5. Every thread can run on multiple CPU's.

In this enumeration, the blocks of code are the opportunities for parallelism we are trying to find. Instructions are the (small) tasks that the CPU can execute. The main reason why the number of possible execution scenarios is astronomical is the third in our enumeration. Consider that the number of instructions per block of code can easily become very large. And in addition those instructions can be executed in many different scenarios. Every other of the mentioned elements multiplies the possible execution sequences even more.

This is why we chose to calculate the dependencies for the software programs. If none of the dependencies are broken, the opportunities will be valid. This means we have to evaluate if our dependencies are correct and complete. This results into the next requirement.

3.1.2 Requirement 2

The architecture can calculate and output the dependencies for all statements in the source code.

Rationale

If all the dependencies between statements are calculated, the opportunities can be checked for not breaking these dependencies. If no dependencies are broken in applying an opportunity, the output of the program will not change.

Challenge

To achieve this requirement we have to cover every possible structure in the .NET C# language. To do this in the given time is a serious challenge and it is likely we will not be able to cover all of the language.

The second challenge is that we will probably be calculating more dependencies then needed. There are scenarios in which dependencies could be broken, while the output of the program remains the same. For example, if two additions are made on the same variable, the sequence does not matter. They will always result in the same value, as long as they are not executed at the same time. Executing them at the same time could cause data races, as noted in 2.1.1 under pitfalls of parallelism.

For our definition of an opportunity, this kind of dependency should not be a dependency at all, because these dependencies eliminate potential opportunities for parallelism.

3.1.3 Requirement 3

The Architecture can implement the generated opportunities for parallelism by applying transformations to the source code.

Rationale

When requirement 1 and 2 have been fulfilled, we have a list of opportunities for parallelism. The next step is to implement these opportunities by applying transformations to the source code.

Challenge

Being the final and last to achieve requirement, it is likely that we will not have the time to achieved this requirement, but would still be a great addition.

3.2 The Architecture

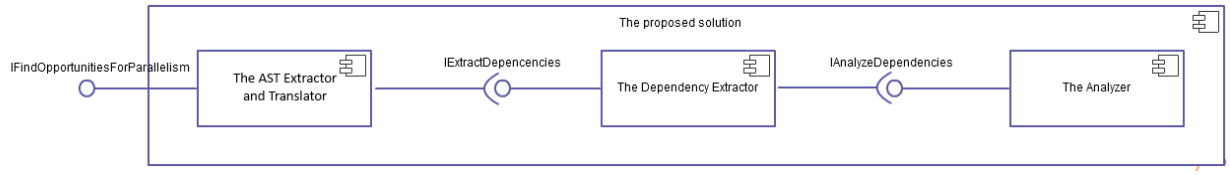

Figure 3: Component view of The Architecture

The pipelined-architecture(visualized in figure 3) that was created during this study is described in this section.

3.2.1 The AST Extractor & Translator

The AST Extractor & Translator(AST E&T) is the start of the architecture. The AST E&T uses NRefactory[3] to get the AST from the target program. Then the ASTs are translated into a format that The Dependency Extractor can comprehend, and then written into text files. This translation is done by visiting the entire AST and translating every node into the format.

NRefactory

NRefactory was chosen to use in The AST E&T because of its functionality and maturity. NRefactory is an open source library to perform refactoring operations to .NET C# code. Another AST Extractor that was considered was Roslyn[4]. However, while working with Roslyn several problems emerged, and some essential functions were missing. NRefactory was chosen in favor of Roslyn because NRefactory did not show any of the problems that Roslyn did. Also, because NRefactory is open source, if problems would emerge there is a possibility to fix these problems independently of the manufacturers.

Separation

The AST E&T is separated from the rest of the architecture. This is because it uses an external library to perform its main task. By separating it from the rest this dependency in the architecture is isolated.

Text files

The translated ASTs are written into files that The Dependency Extractor can comprehend. This is because The AST E&T is written in a separate .NET C# program, and The Dependency Extractor is written in Rascal. So text files are chosen to be the means for communication between them.

AST

Using AST's to handle the source code was chosen because of the Rascal benefits. Rascal has many implemented functions to deal with AST's. For example pattern matching, which allows to visit a tree and look for a specific pattern in all of the child nodes.

[3] http://www.codeproject.com/Articles/408663/Using-NRefactory-for-analyzing-Csharp-code
[4] http://msdn.microsoft.com/en-us/roslyn

3.2.3 The Dependency Extractor

The Dependency Extractor uses the produced text files and parses them into a Rascal-AST. The Dependency Extractor will then visit the AST and analyze it to determine the dependencies and maps them as they are found. The map has the form of: Node X depends on Node Y, in a 1:N relation. A node can be any .NET C# element, for example a statement, property or routine.

Separation

The Dependency Extractor is separated from the AST Extractor & Translator, as mentioned before. But it is also separated from the Analyzer. In this study we will be using the dependencies to look for opportunities for parallelism. However, the dependencies could also be used for different goals. If the Dependency Extractor and the Analyzer were merged, the dependencies could not be used for different goals.

Rascal

The Dependency Extractor is created in Rascal. As mentioned before, Rascal is a meta-programming language, designed to deal with other programming languages. Rascal was chosen because of the many build in functions that help with analyzing source code. The recommendation of Jurgen Vinju and my past experience with Rascal also contributed in the decision for Rascal.

Agile

During this study a point was reached where a pro-agile decision was made. We decided to go into the depth instead of expanding over the width. In other words, the Dependency Extractor was not fully completed before work started on the Analyzer. This decision had to be made because we realized time was too limited to cover all possibilities in the .NET C# language, and we still wanted to obtain some sort of result

The upside of this decision is that we were able to create a front to back study, able to generate results and able to validate these results.

The downside of this decision is that the Dependency Extractor is not fully completed and does not take all possible dependencies in the .NET C# language into account. This also means that the results we are able to generate may not be valid and may break dependencies that are not mapped by the Dependency Extractor.

3.2.4 The Analyzer

```
     public rel[tuple[AstNode, loc],tuple[AstNode, loc]] GetAllIndependentStatements()
     {
S1     relDependence = StartProcessing();
S2     relDeps = relDependence*;
S3     setAll = carrier(relDependence);
S4     relAll = (setAll * setAll);
S5     relPossible = (relAll - relDeps);
S6     rel[tuple[AstNode, loc],tuple[AstNode, loc]] relIndependentStatements = {};
S7     visit (Project) {
S8       case b:blockStatement(stats): {
S9         for([*_,x,*_,y,*_] := stats) {
S10          if(<<statement(x),x@location>,<statement(y),y@location>> in relPossible) {
S11            relIndependentStatements +=
                 <<statement(x),x@location>,<statement(y),y@location>>;
S12          }
S13        }
S14      }
S15    }
S16    return relIndependentStatements;
     }
```

Code block 3

The Analyzer uses the resulting dependency map from the Dependency Extractor and analyzes this map to look for opportunities for parallelism. In the code block 3, the core functionality from the Analyzer is shown: the "`GetAllIndependentStatements`" function. This function returns a relationship, from statement to statement, containing all the statements that are independent of each other. Each statement of this core function of the analyzer is described in table 2.

S1	Get the map of dependencies from the Dependency Extractor.
S2	Create a new relationship of statement to statement that contains the dependencies extended with the transitive closure.
S3	Creates a set containing all statements that are involved in any dependency.
S4	Creates relationship of statement to statement, containing all possible sequences of statements.
S5	Creates a new relationship of statement to statement by deducting the dependencies from the relationship containing all possible sequences.
S6	Declares "relIndependentStatements" as a relationship between two tuples, containing an AstNode and a loc(location). An AstNode can be any element found in the source code AST.
S7-S8	Visits root("`Project`") of the AST for all "`blockStatements`". A blockStatements-object is a node that has child statements. For example, an If-Else statement has 2 child blockStatement-nodes. One for the if-body and one for the else-body.
S9-S10	Loops over all the combinations of all found statements in the block, checking if the combinations are present in the relationship "`relPossible`".
S11	If a combination holds the check, it is added to the "`relIndependentStatements`", which is returned in the end.

Table 2: Description per statement for code block 3.

The generated relationship called relIndependentStatements is the result of the Analyzer. This relationship contains every statement, linked to the statements of which it is not dependent. These statements are always in the same block of code.

Transitive closure

The Analyzer extends the dependencies by doing a transitive closure on the map that is given from the Dependency Extractor. To explain why this is needed, consider code block 4. If we follow the dependencies, S3 depends on S2, which in turn depends on S1. Since a statement cannot execute before its dependencies are executed, S1 has to be executed before S3 can be executed. This is also known as a transitive closure, and is needed to ensure we fulfill requirement 2. If S1 is not executed before S2 the output of the program would change, in this case an exception would be thrown.

```
    | void Foo() {
S1  |  int i = 1;
S2  |  i += 2;
S3  |  int j = i + 3;
    | }
```

Code block 4

Algorithm

The Analyzer goes through a few steps in order to generate the independent statements. To explain these steps, let's first take a look at the input of the Analyzer: the dependency map. The dependency map can be compared to a mandatory sequence of execution; since, as noted in the transitive closure section, it is only valid for a statement to executed after its dependencies have executed. In other words, the dependency map can be compared to a **blacklist** of opportunities for parallelism. It states which statements cannot be executed in parallel and must be executed in sequence. So in order to produce the opportunities for parallelism, or **whitelist**, we need all the opportunities for parallelism deducted by the blacklisted opportunities. Also consider the Example 1 for an example of this process.

```
Format:
Statement: Dependencies statements

Blacklist/Dependencies
S2: S1
S4: S3
S5: S3, S4

All opportunities
S2: S1
S3: S1, S2
S4: S1, S2, S3
S5: S1, S2, S3, S4

Whitelist
S3: S1, S2
S4: S1, S2
S5: S1, S2
```

Example 1: Construction of the Whitelist

Doing this produces the whitelist for opportunities. However, taking all opportunities for parallelism means taking all combinations of any 2 statements. So this is not yet limited to statements that are in the same code block. To do this we visit all the blocks of code in the source code and see which combination of statements in our whitelist are in the same block of code. This results in the list of opportunities for parallelism.

```
var a = 1;
var b = 1;
if (a == 1)
    DoSomething();
if (a == 1 && b == 2)
    DoSomethingElse();
```

Code block 5

Sub paths

As a bit of a side track, there has been some research into sub paths. A sub path is an another type of opportunity for parallelism. The algorithm shown above only results in totally independent statements, while opportunities for parallelism could also be found in not totally independent statements.

Consider code block 5. In this code block, both if-statements depend on variable “a” and thus are not considered an opportunity for parallelism by the previous algorithm. However, they do not depend on each other(assuming both methods “DoSomething” and “DoSomethingElse” share no dependency).

To find these sub-paths, all blocks of code were re-analyzed. These blocks of code are looped through in execution order to find the sub-paths. In this context, a sub-path is defined as follows:

“A sub-path is a set of statements with one statement as the key. The key-statement is not included in the set. The key represents the statement after which the set of statements can be executed (in parallel).”

The statements were divided into sub paths following these rules while looping through all of the statements:

- If there are no sub paths yet, create a new sub path with this statement as key.
- If the statement has no dependencies, create a new sub path with this statement as key.
- If none of the dependencies of the statement are contained in a sub path, create a new sub path with this statement as key.
- If one or more of the dependencies of the statement are all contained in one other sub path, add the statement to that sub path.
- If dependencies of the statement are contained in more than one sub path, create a new sub path with the last dependency(execution order) as key.

This will result in a map of sub paths, and each has a ‘Key’ statement after which the sub path can be executed in parallel.

Working Example Case

To show what the results could be of the sub-path algorithm, an example is given. Consider code block 6, there are many dependencies to be found. The dependencies are summed up in table 2. Following the rules of the algorithm, a list of sub-paths is found, displayed in the table 3.

Overview of the example case:

```
class SubPathExample
{
    public SubPathExample()
    {
S1      var a = 1;
S2      var b = 2;
S3      var c = 3;
S4      var d = 4;
S5      if (b == 2)
S5.1    { d = 5; }
S6      if (a == 1 && b == 2)
S6.1    { c = 4; }
S7      var f = c;
S8      var g = d;
S9      var h = b;
S10     var i = c + d;
S11     var j = h + 1;
S12     var k = h + 10;
S13     var l = k + h;
    }
}
```

Code Block 6

Statement	Depends on
S1	-
S2	-
S3	-
S4	-
S5	S2, S4, S5.1
S5.1	S2, S4, S5
S6	S1, S2, S3, S6.1
S6.1	S1, S2, S3, S6
S7	S1, S2, S3, S6, S6.1
S8	S2, S4, S5, S5.1
S9	S2
S10	S1, S2, S3, S4, S5, S5.1, S6,S6.1
S11	S2, S9
S12	S2, S9
S13	S2, S9, S12

Table 2: Dependencies

Key statement	Contents
S1	-
S2	S9, S11, S,12, S13
S3	-
S3	S6
S4	-
S4	S5
S5	S8
S6	S7
S6	S10

Table 3: Sub-paths

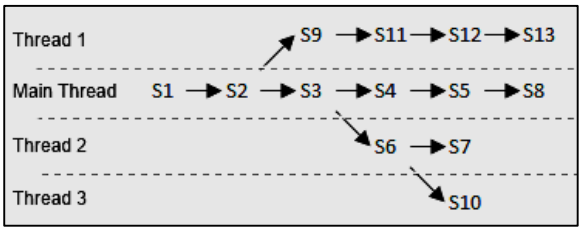

Figure 4: Visualization of applied threading

The Sub-paths are opportunities for parallelism. The statements should be executed in order, and every time a Key statement is executed, the contents of the Sub-path can be executed in a new thread. Figure 4 illustrates how this would look. After statements S2, S3 and S6 new threads are created. By executing the statements like this, the 13 step long function can be executed in 6 steps. Of course an overhead will be added by the implementation of threads which will make it bigger than 6 steps, but that is the trade-off.

```
    class ExampleField
    {
F1      int counter = 0;
        public ExampleField()
        {
S1          var a = counter;
S2          Process();
S3          var b = counter;
        }

        public void Process()
        {
S4          counter += 1;
        }
    }
```

Code Block 7

A Problem

After some experimenting, a problem emerged. If properties or fields were used throughout the body of a code block, it causes this algorithm to be far less effective. This is because fields and properties introduce new dependencies. For example, see code block 7.

When field “counter” is read for variable “a”, it will return 0. When field “counter” is read for variable “b”, it will return 1 because “Process” has incremented it. To ensure this behavior, this routine cannot be made parallel. This means there is probably a dependency. There is no direct dependency between S1 and S2. However, “counter” is assigned inside the body of “Process”. If we add a dependency between “counter” and “Process”, any usage of “counter” can never be made parallel with the invocation of “Process”, which is correct behavior. This also goes the other way around, “Process” uses the value of “counter” and thus it is dependent on it. So no assignment of “counter” can be made parallel with “Process”, which is again correct behavior.

Because of these dependencies, many of the identified the sub-paths from the algorithm became connected by dependencies. And there for were not detected by the algorithm.

3.3 Conclusion

By creating the proposed solution, we have achieved most of our goals. We have found a way to analyze .NET C# code by extracting an AST and translating the AST into Rascal format text files, which are then picked up by Rascal for analysis. We have extracted a list of dependencies between various elements in the source code. By analyzing the dependency list we were able to create a relationship between statements representing statements that are independent of each other.

Even though the independent-statements relationship is close our next goal; generating a list of opportunities for parallelism, it is not quite the same. The proposed solution does not yet combine the independent-statements into code blocks that represent opportunities for parallelism. However, transforming the independent-statement relationship to blocks of code is not expected to be hard to achieve, but it will take some time.

Instead of doing this, we would rather validate the dependencies and independent statements. We are not sure that the found dependencies are correct, it could be possible that some dependencies are not dependencies at all. Also we do not know if the independent statements that resulted from the analysis of the dependencies are in fact independent from each other. We still have to validate both of them.

4. Evaluation

4.1 Evaluation questions

In this chapter we will first look at the goals we have set in chapter 2.2. Then we will look at the pitfalls of parallelism that have been summed up in chapter 2.1. And finally we will try to achieve our final 2 sub goals:

- Validate if the found opportunities can improve performance.
- Validate that the found opportunities do not change the output of the program.

We will do this by answering the following evaluation questions.

1. Can the proposed solution generate opportunities that would improve performance?
2. Are the found independent statements really independent?
3. Are the found dependencies real dependencies?

4.2 Goals

In chapter 2.2 we have set a number of goals to achieve with this study. In this chapter we will validate if these goals are achieved. We will first look at the sub goals before we look at the main goal.

4.2.1 Sub goals

These are the sub goals we have set up to work towards the main goal.

- Find a way to analyze .NET C# source code.

We achieved this sub goal by using NRefactory and creating The AST Extractor and Translator.

- Determine the dependencies between all elements in the source code.

The Dependency Extractor was created to achieve this sub goal, although we still have to validate if the results are correct.

- Analyze the dependencies and determine which elements in the source code are independent.

This sub goal was achieved by creating The Analyzer. The results of the Analyzer will be validated in this chapter.

- Generate a list of opportunities for parallelism based on the independent elements.

This sub goal was not achieved. However, to achieve this goal from the results of The Analyzer should not be too hard.

The following two goals will be handled in this chapter:

- Validate if the found opportunities can improve performance.
- Validate that the found opportunities do not change the output of the program.

4.2.2 Main goal

Our main goal for this study has been described as follows:

"Generate a list of valid opportunities for parallelism, that could improve performance, to guide programmers while implementing parallelism."

As noted in the previous section, not all sub goals have (yet) been achieved. And one of the sub goals will not be achieved during this study. However, if the last 2 sub goals will be achieved in this chapter, we will have achieved most of our main goal. Only leaving the transformation from independent statements to independent code blocks, in order to meet our definition of an opportunity as described in chapter 4, section 3.

4.3 Pitfalls of parallelism

In chapter 2.1 we have summed up the pitfalls of parallelism. In this section we will look at each of these and evaluate if we have avoided them.

4.3.1 Shared Memory Problems

To avoid this pitfall we have analyzed the source code for dependencies, and excluded opportunities for parallelism that would break one of these dependencies.

4.3.2 Over-parallelization

We have not avoided this pitfall with our proposed solution. Some opportunities for parallelism could lower performance by implementing them. However, in section 4.4, we will show that the proposed solution can also improve performance. A fairly simple solution to partly avoid this pitfall is to order the opportunities for parallelism by the total lines of code that would be run in parallel. Opportunities with bigger chances for performance increase would probably end up in the top.

4.3.3 Cross-thread operations

We have been unable to look into avoiding this pitfall. This means that the found opportunities for parallelism could introduce runtime exceptions. Programmers that will implement the opportunities will have to look out for cross-thread operations on .NET controls that do not support this.

4.3.4 Deadlocks

This pitfall for parallelism has to be avoided by the programmers that will implement the found opportunities for parallelism. However, the found opportunities for parallelism are completely independent of each other. To create deadlocks, there have to be shared resources amongst the threads. Since they share no dependency, deadlocks cannot be the result of implementing the

opportunities. Aside from this, the opportunities still have to be implemented by a programmer. The programmer could of course still chose to implement an opportunity by using locks, and in turn cause deadlocks.

4.4 The optimal case

To answer the first evaluation question, we will handle a self-constructed case. In this case we expect performance gain equal to almost twice the normal performance, because the overhead has to be deducted.

4.4.1 The Code

```
class OptimalParalellizable
{
    public void OptimalParalellizableFunction() {
        FunctionOne();
        FunctionTwo();
    }
    private void FunctionOne () {
        for (int i = 0; i < 10; i++) {
            //Heavy load function
            Thread.Sleep(50);
        }
    }
    private void FunctionTwo () {
        for (int i = 0; i < 10; i++) {
            //Heavy load function
            Thread.Sleep(50);
        }
    }
}
```

Code Block 8

Code block 8 has been written to be optimal for parallelization. The results of The Analyzer is that the following two lines could be parallelized:

- FunctionOne();
- FunctionTwo();

For this example, we will add the parallelization and validate if the performance will indeed go up for our function called:

"`OptimalParalellizableFunction`".

4.4.2 Results

```
public void OptimalParalellizableFunctionThreading()
{
    var thread1 = new Thread(new ThreadStart(FunctionOne));
    var thread2 = new Thread(new ThreadStart(FunctionTwo));

    thread1.Start();
    thread2.Start();

    while (thread1.ThreadState == 0 || thread2.ThreadState == 0) //0 = running
    {
        Thread.Sleep(1);
    }
}
```

Code block 9

Code block 8 shows the modified function so it uses parallelization. In comparison to the function that doesn't use parallelism, it is easy to see how some pitfalls could be introduced. The modified function is a lot bigger. It introduces overhead and increased the complexity of the function.

The results of the performance test are shown in table 4, and match our expectations. The function is almost twice as fast with an performance increase of 49.5%. The overhead introduces a delay of 5 milliseconds but also a performance increase of 501 milliseconds. The overhead cost is nothing in comparison with the performance gain for this case.

Number of runs	2008
Average time with parallelization	511 milliseconds
Average time without parallelization	1012 milliseconds

Table 4: The performance results of the applied opportunity.

4.5 Validating (in)dependencies

We will first try to validate the independencies and then do the same for the dependencies.

4.5.1 Applying parallelism

The obvious method to test if the found opportunities are valid is to apply parallelism for the given opportunities and validate if the program still functions correctly. This method has been applied in section 4.4, The optimal case, and has shown that the implemented opportunity has executed

successfully 2008 times. This has shown us that implementing an opportunity can introduce performance increase.

However, as mentioned before, applying parallelism to an opportunity means there will be an astronomical amount of possible execution scenarios how the threads could be executed. So by applying parallelism and testing it a few times does not validate the results. For example, the same scenario could occur run every time.

This method is a form of Random Testing[13]. The basics of Random Testing is generating a set amount of random tests and validating them on a program. Applying Random Testing can only give an indication based on the number of performed tests and, if possible, the number of possible tests.

To be able to gain a better indication if the found opportunities are valid opportunities, we will explore another validation method.

4.5.2 Swapping

Thread 1	Thread 2	Thread 1	Thread 2	Thread 1	Thread 2
1		1			3
2			3	1	
	3		4		4
	4	2		2	
1			3		3
	3	1			4
2		2		1	
	4		4	2	

Table 3: Possible sequences for 2 threads with 2 statements run in parallel. The numbers indicate the execution order.

Another way to test if statements could be run in parallel is to swap them around. When 2 blocks of code are run in parallel, the execution sequence is unpredictable. There for, the program should produce the same result in any possible order of execution of the 2 blocks of code. We can simulate this behavior by swapping statements in the source code. This method is a form of Mutation Testing[15]. We will be creating mutations of the code and testing these mutations. Normally, these mutations should be caught and rejected by tests. Indicating a level of test-coverage. For this study, the mutations should not be caught. This is because we are trying to find out if there are dependencies between the parts of mutated source code. For example, the 2 blocks of code could both contain 2 statements. If they would be run in parallel. there are 6 possible sequences in which they can be executed. See table 3 for these sequences. By using this method it is testable when the sets of statements that are run in parallel are not too big. If they become too big, the astronomical amount of possibilities come back into play. However, if there is a way to automate this test it could be plausible to use this method to test all the possible sequences and thus prove that the opportunities are valid. Another advantage of this method is

that it does not require any overhead, unlike with applying parallelism. Also when applying the needed overhead, errors could be introduced.

Condition

If 2 statements are independent of each other, it does not mean they can be swapped without changing the output of the program. We have to check if the independent statements meet the following condition:

"Two independent statements can be swapped if none of the statements between these two statements depend on either of the statements, nor do the two statements depend on any of these statements."

This condition makes sure that a statement is not swapped past or in front of a dependency.

4.5.3 Applying the methods

The aim of applying our solution was on a complete software program, that included Unit testing. However we stumbled upon a problem. Because the Dependency Extractor does not take all of the .NET C# framework into account, it could not handle the complete software program that used unhandled structures from the .NET C# language. The cause of this problem lies in our previous decision of not fully completing the Dependency Extractor. Because the Dependency Extractor is not fully completed, our solution cannot handle full software programs. Instead we will apply our solution on smaller cases and analyze them for opportunities for parallelism.

4.5.4 Swapping independencies

Test 1

In this test we will start by swapping single statements with each other.

Code:

```
1   int field1 = 1;
2   int field2 = 2;
3   int field3 = 3;
4   public int Test1()
5   {
6       var a = 1;
7       var b = 2;
8       if (b == 2)
9           field2 = 2;
10      if (a == 1 && b == 2)
11          field1 = 2;
12      var c = field1;
13      var d = field2;
14      var e = b;
15      var f = field1 + field2;
16      var g = field3;
17      var h = f + g;
18      return a + b + c + d + e + f + g + h + field1 + field2 + field3;
19  }
```

Output: 30

Dependencies

The following dependencies are found in the code.

Statement	Depends on statement(s)
1	4, 11
2	4, 9
4	1, 2, 3
8	2, 7
10	1, 6, 7
12	10
13	8
14	7
15	8, 10
16	3
17	15, 16
18	3,6,7,8,10,12,13,14,15,16,17

These dependencies are visualized in the following figure. In this figure, every number represents a statement, and an arrow from statement 1 to statement 2 means statement 1 depends on statement 2.

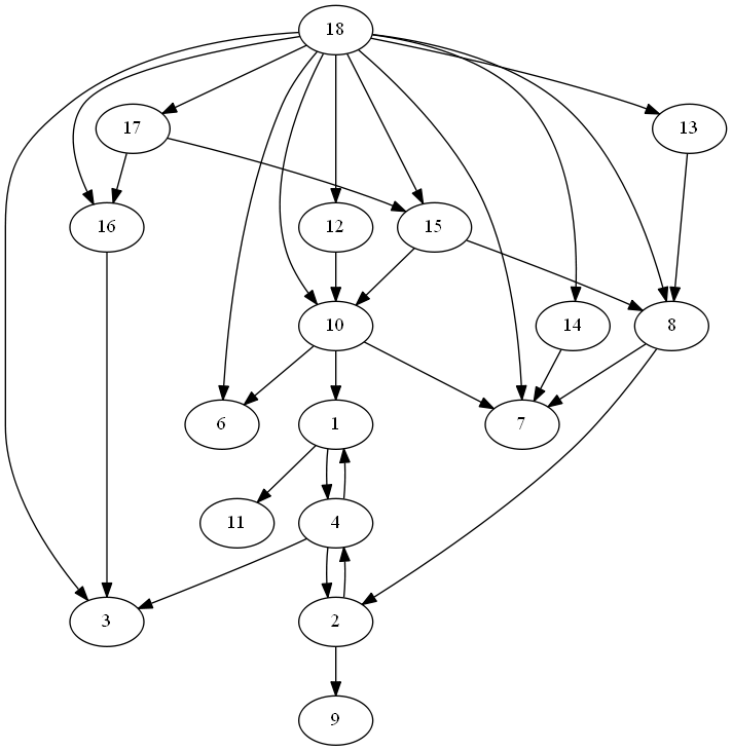

Figure 5: Dependencies between statements from Test 1.

Independent statements

The following independent statements are found in the code.

Statement	Independent from statement(s)
6	7, 14, 16
7	16
8	14, 16
10	14, 16
12	13, 14, 15, 16, 17
13	14, 15, 16, 17, 18
14	15, 16, 17

Swappable statements

After applying the condition on the independent statements, the following statements have been found suitable for swapping.

Statement	Swapped with statement	Output unchanged after swapping
6	7	Yes
12	13, 14, 15, 16	Yes
13	14, 15, 16	Yes
14	15, 16	Yes
15	16	Yes

Test 2

Code

```
1   public int Test2()
2   {
3       var result1 = 1;
4       while (result1 < 100)
5           result1 += 15;
6       var a = result1 / 2;
7       if (a > 50)
8           result1 += a;
9       else
10          result1 -= a;
11
12      var result2 = 1;
13      while (result2 < 100)
14          result2 += 30;
15      var b = result2 / 2;
16      if (b < 50)
17          result2 += b;
18      else
19          result2 -= b;
20
21      return result1 + result2;
22  }

    Output: 220
```

Dependencies

Statement	Depends on statement(s)
4	3, 5
5	3, 4
6	3, 4
7	6, 8, 10
8	3, 4, 6, 7
10	3, 4, 6, 7
13	12, 14
14	12, 13
15	12, 13
16	15, 17, 19
17	12, 13, 15, 16
19	12, 13, 15, 16
21	3, 4, 7, 12, 13, 16

Figure 6 visualizes these dependencies. The same rules apply as the figure from Test 1.

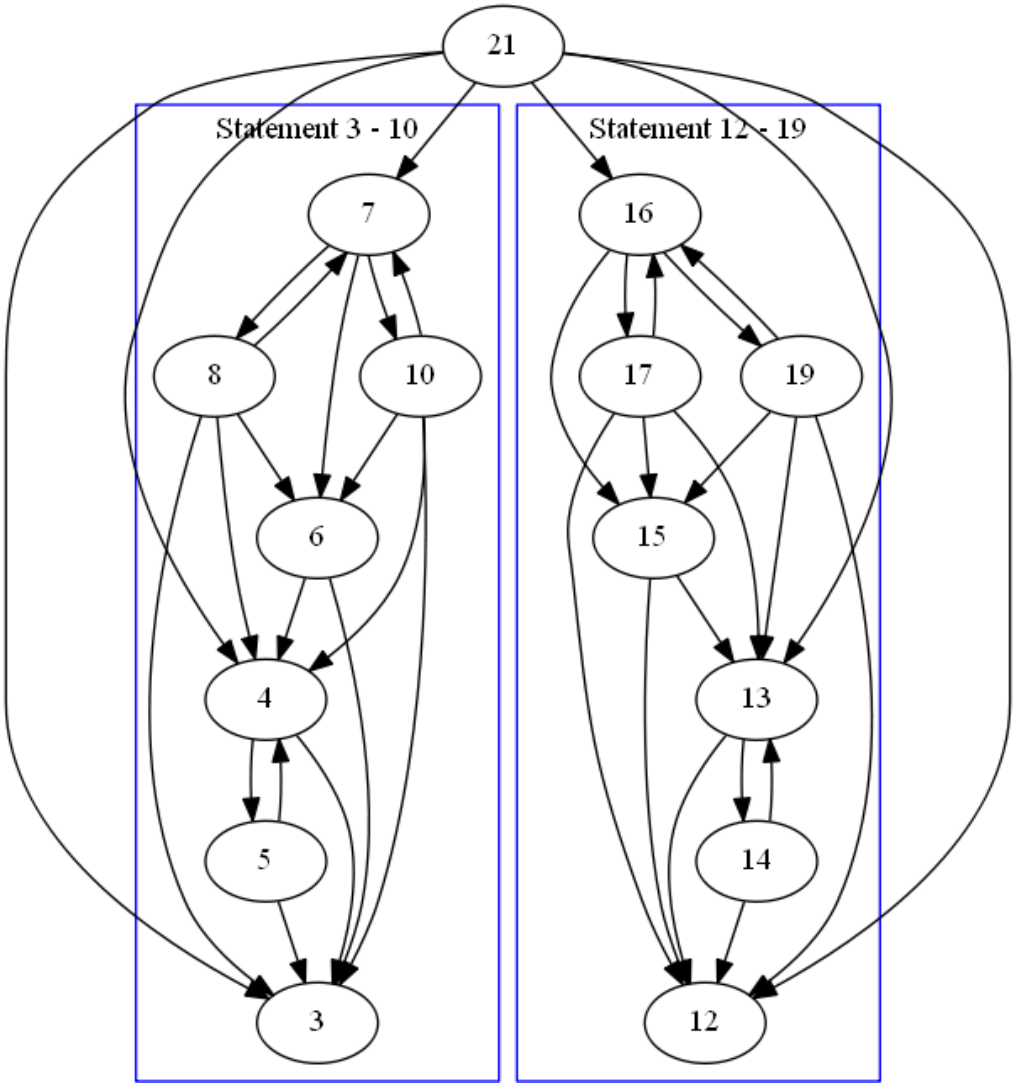

Figure 6: Dependencies between statements of Test 2.
The blue boxes represent the separated statement groups.

Independent statements[5]

Statement	Independent from statement(s)
3	12, 13->14, 15, 16->19
4->5	12, 13->14, 15, 16->19
6	12, 13->14, 15, 16->19
7->10	12, 13->14, 15, 16->19

Swappable statements

Statement	Swapped with statement	Output unchanged after swapping
7->10	12	Yes

[5] In this table, the '->' sign in "1->5" means statements 1, 2, 3, 4 and 5.

Mid-Conclusion

For Test 1 and 2, we have validated the independent statements by applying the swapping method. However, the algorithm for finding swappable statements turned out to be lacking. Resulting in only 11 out of 21 independent statements were able to swap for test 1. For test 2 this was even worse: 1 out of 16.

To increase the number of swappable statements, we will extend our algorithm. When we find 2 statements suitable for swapping, we will keep checking the following statements until one is found that is not suitable. This addition could help validate more independent statements, because a statement will now be swapped with more than one other statement.

Test 3

For Test 3, we will retest the code of Test 2 with the new addition to our swapping method. The code of Test 2 will remain the same, along with the dependencies and independent statements.

Swappable statements

Statement	Swapped with statements	Output unchanged after swapping
7->10	12, 13->14, 15, 16->19	Yes

This time the swappable statements resulted in a different result. Statement 7 was swapped with statements 12, 13, 15 and 16. Meaning the If-statement of statement 7 was swapped to below the If-statement of statement 16.

Since he output was not changed, this test proves that statement 7 is indeed independent from statement 12, 13, 15 and 16.

Test 4

So we know that the proposed solution does not yet work on bigger programs. However, to see how it would perform with larger amounts of code, we have created a function that would not break on analyzing its dependencies. This function is shown in code block 12.

The function is called ‘LiveDay ‘ and represents how a typical day for ‘joe’ would go. This function is bigger and more complex than the code blocks from the previous tests. It contains various structures that the proposed solution can handle. It also contains deeper structures.

Code

```
1   public static void LiveDay(ref Person joe)
2   {
3       var isAlive = joe.IsAlive;
4
5       while ( joe.SleptForHours <= 8 ) {
6           joe.SleepForHour();
7       }
8
9       if ( joe.IsHungry ) {
10          var refrigerator = joe.Refrigerator;
11          try {
12              var foods = refrigerator.GetFavoriteFood(joe);
13              foreach ( var food in foods ) {
14                  joe.Eat(food);
15              }
16          }
17          catch ( Exception ex ) {
18              if ( refrigerator.IsEmpty )
19                  refrigerator.Fill(joe.DoGroceries(new Supermarket()));
20              else //joe is no expert
21                  joe.CallMechanic();
22          }
23      }
24
25      var work = joe.Work;
26      if ( work.HasToGoToWork(joe) ) {
27          var car = joe.Car;
28          joe.Driving = joe.Position != work.Position;
29          while ( joe.Driving ) {
30              joe.Drive(car, work);
31              joe.Driving = joe.Position != work.Position;
32              var accident = joe.InCarAccident;
33              if ( accident == "deadly" ) {
34                  isAlive = false;
35                  joe.Driving = false;
36              }
37          }
38          if ( joe.IsAlive )
39              joe.DoWork(work);
40      }
41      if ( isAlive ) {
42          joe.DaysAlive += 1;
43          if ( joe.DaysAlive > 9000 )
44              isAlive = false;
45      }
46      joe.IsAlive = isAlive;
47  }
```

Dependencies

The amount of dependencies in this function is quite big. To illustrate how quickly the dependencies can grow and become quite complicated, we will add the visualization of the dependencies that exist within this function. Any dependencies to other parts in the code have been excluded.

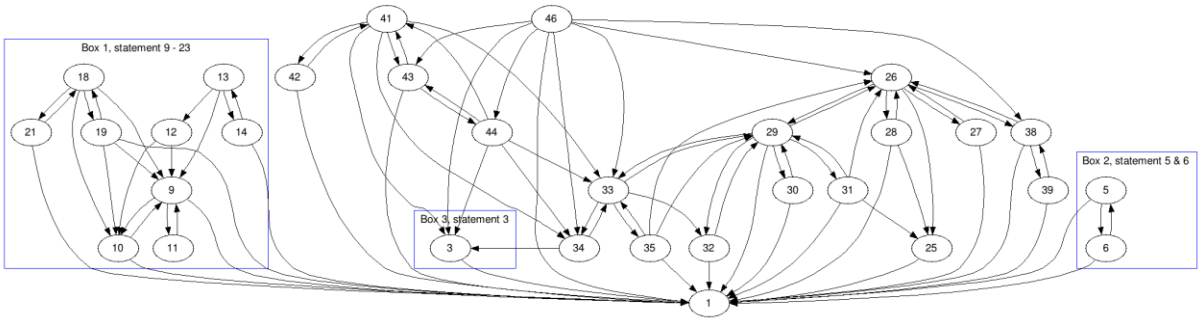

Figure 7: Dependency visualization for Test 4.

In figure 7, the blue boxes will be the swappable statements. A quick look at Figure 7 reveals a lot of information. If we look at box 1 we can see that it has no dependencies to any statements outside the box, except for statement 1. The same applies for box 2. If there are no dependencies between the "blocks of statements", they could be valid opportunities for parallelism. If we look at box 3, we can see there are a few statements dependent on box 3. But if we look closely, we can see that these dependencies are all from statement 34 and beyond. In other words, there are no dependencies between box 3 and any statement until statement 34. Meaning box 3 can be executed in parallel until statement 34 is executed.

The big knot at the bottom is statement 1. It could seem that this would get in the way of finding the opportunities for parallelism. However, statement 1 is executed before all other statements. Meaning dependencies to statement 1 from all other statements will not prevent valid opportunities for parallelism.

Analyzing the dependencies like this is exactly what The Analyzer can do.

Swappable statements

Statement	Swapped with statements	Output unchanged after swapping
3	5->26	Yes
3	9->25	Yes
5->7	9->46	Yes
5->7	26->46	Yes
9->23	26 ->46	Yes

This function was tested by calling the mutated versions with different instances of "Person" with different values for the properties. By doing this we achieved full code coverage.

Even though there appeared to be some problems with the dependencies of the code, the given results turned out to be independent. This means that the proposed solution can indeed detect valid opportunities for parallelism for more complex code.

4.5.6 Validating dependencies

The third evaluation question is hard to answer. We could use the swapping method in the same way we used it to validate the independent statements, but there is a difference between dependencies and independencies. As we have noted before, some dependencies can be broken without breaking the program or changing the output (chapter 6, section 1.3). An example of one of these dependencies is when 2 statements perform an addition on the same variable. A change in the order of the additions does not matter for the output nor does it break the program. And still, there are dependencies between them. Another problem with testing dependencies with the swapping method is that dependencies are not always between swappable statements. Dependencies can also be to routines, or to properties. These structures cannot be swapped, and thus would not be able to validate using the swapping method.

This means there might be some false positives in the list of dependencies, and would mean that the list of independent statements is smaller than it could be. Fortunately, we have been able to validate the independent statements. So the list of independent statements has a high precision, and will not contain a many false positives.

5. Conclusion

We started this thesis with the following problem: How can we quickly determine if a (large complex) .NET C# program could be transformed to use parallelism. To find a solution we have set a main goal and sub goals. The first sub goal we achieved was to extract a dependency model from .NET C# software programs. We were able to achieve this by creating an AST of the code in Rascal using The AST Extractor & Translator. Then we consumed the AST and created a dependency model with The Dependency Extractor. The next sub goal was to analyze the dependency model and determine the independent statements in the source code, which was achieved by creating The Analyzer. The next sub goal was to generate a list of opportunities for parallelism based on the results of The Analyzer, but instead we chose to validate the results we found so far(which are the final two sub goals). By validating the results, we found that we can indeed find opportunities for parallelism that improve performance (up to at least 49.5%). And in all of our tests we have not been able to detect false positives in the found independent statements, this means we have achieved high precision in the generated list of independent statements.

Our main goal was to generate a list of valid opportunities for parallelism, that could improve performance, to guide programmers while implementing parallelism. We have achieved most of our main goal, we are able to generate a list of valid independent statements that can lead to improved performance.

However, there are a few limitations to the proposed solution. Transforming the list of independent statements into our definition of an opportunity for parallelism has not yet been achieved, even though it is not expected to be hard to achieve if time is available. Unfortunately, time was short and lead to another limitation: The proposed solution cannot handle the full .NET C# language, and thus it cannot handle complete software programs. However, because we chose to start on the creation of The Analyzer, we have been able to get and validate our results.

By using the technique presented in this thesis, opportunities for parallelism can be found in the source code that can guide programmers to increase performance by implementing parallelism into software programs. However, because this technique cannot handle the full .NET C# language (yet) we have not been able to find a complete solution to the problem. The proposed technique can handle small .NET C# programs. To be able to handle large complex programs, the proposed solution has to be expanded to handle the full .NET C# language.

6. Future work & Discussion

First of all, as future work I would recommend extending the Dependency Extractor to cover the full .NET C# language. This will make the proposed solution able to be applied to complete programs. Chapter 2, section 6.2: Unhandled dependencies, can be used as a start for this extension.

Secondly, completing the last step of transforming the list of independent statements into a list of opportunities for parallelism would, in combination with the first recommendation, complete the solution for the problem.

When the solution is completed, a great addition to this study would be to fulfill the third requirement(stated in Chapter 3, section 1.3): adding a new module to consume the generated opportunities to transform the code to apply the generated opportunities for parallelism.

Another topic that needs more research is sub path detection, discussed in Chapter 3, section 2.4. There are probably a lot more opportunities for parallelism to be found in the source code than what we are currently able to find. The sub path algorithm is an example of this. If this could be extended it would be another great addition.

For this study we have chosen to create a list of opportunities with a high precision. This has the advantage of providing accurate information about the analyzed program, and the results actually tell the programmers where to implement parallelism. However, this could also be seen from another point of view. If we would generate as many opportunities as possible, and maybe give the opportunities a score to indicate the reliability, we would be helping the programmers even more. If the programmer has more leads on where parallelism could be added, the program could gain more performance. Even if not all the opportunities can actually be implemented, they might still be a valid opportunity after some refactoring is done by the programmer.

7. References

[1] A. Telea, H. Hoogendorp, O. Ersoy and D. Reniers. Extraction and visualization of call dependencies for large C/C++ code bases: A comparative study. In Visualizing Software for Understanding and Analysis, pages 81-88, 2009.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In Proceedings of the spring joint computer conference (AFIPS '67 (Spring)), pages 483-485, ACM, New York, NY, USA, 1967.

[3] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker and J. Dongarra. MPI: The Complete Reference, volume 2, The MPI Extensions. MIT Press, Cambridge, MA, USA, 1998.

[4] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. In proceedings of the IEEE, volume 81, issue 2, pages 211-243, 1993

[5] E. Burrows and M. Haveraaen. Programmable data dependencies and placements. In Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming (DAMP '12), pages 31-40, ACM, New York, NY, USA, 2012.

[6] G. Goff, K. Kennedy, and C. Tseng. Practical Dependence Testing. In Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI '91), pages 15-29, ACM, New York, NY, USA, 1991.

[7] R. E. Sward and A.T. Chamillard. AdaSlicer: an Ada program slicer. In Proceedings of the annual ACM SIGAda international conference on Ada (SigAda '03), pages 10-16, ACM, New York, NY, USA, 2003.

[8] K. Morita, A. Morihata, K. Matsuzaki, Z. Hu and Masato Takeichi. Automatic inversion generates divide-and-conquer parallel programs. In proceedings of the ACM SIGPLAN conference on Programming language design and implementation, volume 42, issue 6, pages 146-155, 2007.

[9] G. Goumas, N. Drosinos, M. Athanasaki and N. Koziris. Automatic parallel code generation for tiled nested loops. In Proceedings of the ACM symposium on Applied computing (SAC '04), pages 1412-1419, ACM, New York, NY, USA, 2004.

[10] D. Huang and J. G. Steffan. Programmer-assisted automatic parallelization. In Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research (CASCON '11), pages 84-98, IBM Corp., Riverton, NJ, USA, 2011.

[11] A. Canedo, T. Yoshizawa and H. Komatsu. Automatic parallelization of simulink applications. In Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization (CGO '10), pages 151-159, ACM, New York, NY, USA, 2010.

[12] G. Mackey, S. Sehrish, J. Bent, J. Lopez, S. Habib and J. Wang. Introducing map-reduce to high end computing. Petascale Data Storage Workshop. PDSW '08, volume 3, pages 1-6, 2008.

[13] T.Y. Chen. Adaptive Random Testing. Quality Software QSIC '08, The Eighth International Conference, page 443, 2008.

[14] N. Sangal, E. Jordan, V. Sinha and D. Jackson. Using dependency models to manage complex software architecture. In Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages and applications (OOPSLA '05), pages 167-176, ACM, New York, NY, USA, 2005.

[15] M. R. Woodward. Mutation testing-an evolving technique. In Software Testing for Critical Systems, IEE Colloquium, pages 3/1-3/6, 1990.

[16] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. Computers, IEEE Transactions, volume C-28, pages 690-691, 1979.

[17] E. G. Coffman, M. Elphick and A. Shoshani. System Deadlocks. ACM Computing Surveys, volume 3, issue 2, pages 67-78, 1971.

[18] I. Ronen, N. Dor, S. Porat and Y. Dubinsky. Combined static and dynamic analysis for inferring program dependencies using a pattern language. In Proceedings of the conference of the Center for Advanced Studies on Collaborative research (CASCON '06), Hakan Erdogmus, Eleni Stroulia, and Darlene Stewart (Eds.), article 3, IBM Corp., Riverton, NJ, USA, 2006.

[19] M. Weiser. Program slicing. In Proceedings of the 5th international conference on Software engineering (ICSE '81), pages 439-449, IEEE Press, Piscataway, NJ, USA, 1981.

[20] B. Franke and M.F.P. Oboyle. Compiler parallelization of C programs for multi-core DSPs with multiple address spaces. Hardware/Software Codesign and System Synthesis, First IEEE/ACM/IFIP International Conference, pages 219-224, 2003.

[21] P. Larsen, R. Ladelsky, J. Lidman, S.A. McKee, S. Karlsson and A. Zaks. Parallelizing more Loops with Compiler Guided Refactoring. Parallel Processing (ICPP), 2012 41st International Conference, pages 410-419, 2012.

[22] P. Klint, T. van der Storm and J. Vinju. EASY meta-programming with Rascal. In Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III (GTTSE'09), pages 222-289, Springer-Verlag, Berlin, Heidelberg, 2009.

[23] J. Ferrante, K. J. Ottenstein and J. D. Warren. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems (TOPLAS), volume 9, issue 3, pages 319-349, 1987.

[24] A. Aho, R. Sethi and J. Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.